Javelin's Guide for PennMUSH Gods

by Alan Schwartz
Javelin@Belgariad and elsewhere
Paul@DuneMUSH and Dune II

Note: This document is available as both a single document (suitable for printing) and a multi-part document (more appropriate to hypertext). These multiple views are automatically generated with a Perl script called multiview, by James "Eric" Tilton, 3/29/94 (jtilton@willamette.edu), heavily modified by me. This document is text-browser friendly.

The current edition of this document is available online at http://pennmush.tinymush.org/~alansz/guide.html.

Here are the diffs since the last revision of the Guide, and here's the guide's log of changes.

You can search the Guide.

Please leave your comments about this Guide!

Introduction and Thanks

This document is an attempt to set down the fruits of my experience as a PennMUSH God, site admin, and source code hack. It should be particularly useful to the new PennMUSH God who's having trouble or wants to get a handle on where to start hacking in new features.

Documentation is never a solo project. I am indebted first and foremost to Amberyl (aka Lydia Leong) not only for maintaining, updating, and generally rewriting the PennMUSH code into the form I received it in, not only for teaching me a great deal about how to build, run, hack, and debug a MUSH, not only for writing an excellent MUSH user and coder manual herself, but for that first roybit which got me started on the path to MUSH administration!

I also thank all the MUSH gods who've contributed questions, problems, suggestions, patches, and ideas to the various MUSH mailing lists and newsgroups. As well as the players and admin of the various MUSHes which I've been involved in as site/source hack or consultant (Pandemonium, Mua'kaar, Gohs, Dune II, Princess Bride, Dorsai, and especially DuneMUSH) for their patience. :)

Table of Contents

Choosing a MUSH server

Contents

MUSH servers

There used to be basically two choices available to the God who wants to run a MUSH server, rather than a MUCK, MUSE, MOO, TinyMUD, TeenyMUD, or any of the other flavors of MU* out there: PennMUSH 1.6 or TinyMUSH 2.0.

Now there are five MUSH servers I'm aware of:

PennMUSH 1.6
: The subject of this guide, a memory-based MUSH server. PennMUSH tends to suffer from creeping featurism, which may be a good or bad thing depending on your outlook. :) It's developing toward having a superset of the TinyMUSH features.

TinyMUSH 2.0

The last developed version is 2.0.10p6, I believe. This has been the standard disk-based MUSH server, though TinyMUSH 2.2 seems to be supplanting it, as there doesn't seem to be much development on 2.0 now.

TinyMUSH 2.2
: A rewrite of much of TinyMUSH 2.0, with a similar feature set. The current standard for disk-based MUSH servers.

AlloyMUSH 1.0
: Based on a beta version of TinyMUSH 2.2, with various PennMUSH features added like @mail, powers, and zones.

TinyMUX 1.0
: Based on TinyMUSH 2.0.10p6, with various PennMUSH features added, like @mail, powers, zones, all the Penn functions, and new channel and macro systems.

Which server you should choose depends primarily on the features of your site and secondarily on the features of the server. The big first decision is if you want a memory-based server (PennMUSH) or a disk-based server (the others). If you choose disk-based, you'll get to make other choices which are not covered in this document.

A very good list of servers that's often more up-to-date than the one above is maintained by T. Alexander Popiel at http://www.tinymush.org/~popiel/mushinfo.html.

Gathering system info

Ideally you'll want to know the following information about the computer system on which you're going to run your MUSH:

1. What is the brand and model of the hardware? For example: Sun Sparcstation 2, DEC Decstation 3100S, etc. The "uname -a" command often will give you this information.
2. What is the name and revision level of the operating system? For example, SunOS 4.1.3_u1, Ultrix 4.2, etc. "uname -a" gives this information. It may also be mentioned in the banner you see when you log into the system or in the file /etc/motd. It's useful to know if the operating system is basically BSD (SunOS 4, Ultrix, HP-UX, Linux, NeXT) or SysV (Solaris 2, Irix)
3. How much physical memory is in the machine? For example: 8Mb (typical of a 386/486 machine), 16Mb, 64Mb, etc. Physical memory is ram chips on the machine's motherboard. Figuring this out can be tricky. You can usually find this out if you reboot the machine, or can read its boot logs, which are /usr/adm/messages or /var/adm/messages on most BSD systems.
4. How much virtual memory (swap space) is the machine configured for? Unix systems can use disk space as virtual memory - it's slower than physical memory, but much cheaper and more available. Virtual memory is used when processes need to be paged (portions of their memory sent to swap space) or swapped (the whole process sent to swap space) to free more physical memory. The command /etc/pstat -s reports total memory space (physical + swap) on BSD systems. Solaris uses swap -l. For others, check the man pages for 'swap' or 'virtual'. Typically, machines have around twice as much swap as physical memory, but this varies.
5. How much disk space will the MUSH have available to it? If you're running the game on someone else's machine (and of course you have permission, right? Illegal MUSHes are easily detected and often result in nasty consequences and give MU*'s a bad name), how much disk space can your game account use without causing problems? The upper limit is usually the size of the disk partition on which the game account lives. The "df" command shows the size of disk partitions, and the directories which are housed by those partitions.
6. What is the typical user load of the machine, and what is important to the users? If you own the machine and it's the workstation on your desk that only you use, there's no problem. If the machine is used by others however, it's important to consider what kind of other software is likely to be run. If people use it for email, news, and word-processing, they'll want quick keyboard response, but won't be taking up much CPU or memory. If people use it to compile large software packages or run statistical analyses, you'll be sharing much more CPU and memory.
7. How is the machine connected to the internet? Most university sites don't have to worry about the bandwidth of their connections, but if you're connected via a 14.4k SLIP line, expect some trouble. :)

Penn and Tiny compared

The main difference between PennMUSH and TinyMUSH is where the database is stored while the game is running. In PennMUSH, the database is loaded entirely into memory while the game is running (text on objects and in mail messages is compressed), and a copy of the database is dumped onto disk periodically. In TinyMUSH, the database resides on the disk, and as portions are needed, they are loaded into memory and cached (kept around for a while in memory). Additional memory is required for player connections, so the more players the game typically has, the more memory is typically required.

Typically, this means that you want to run PennMUSH when you have plenty of memory to spare, and TinyMUSH when memory is scarce but you have plenty of disk space (and fast disks).

Actually, things are a bit more complicated on modern workstations, since parts of the PennMUSH process that aren't being used will generally be paged out of memory to disk by the operating system. In some cases, this can be more efficient than having the MUSH server do the paging, but generally, it results in more use of physical memory.

Here are some typical usage statistics. The "memory" column refers to the game's total size (SZ on ps listings). The "disk" column refers only to the uncompressed size of the database - not the source code, executables, backup databases, etc. (TinyMUSH entries include the size of the .gdbm.{db,pag,dir} and the .db file). Typically for PennMUSH, you have 2 databases: the database from which the process was started (indb) and the database which the process is currently using/writing (outdb). It's also possible to use compressed databases, which save considerable disk space at the cost of longer startup times and dumps.

```
Type  OS           Name        # Objects  Memory  Disk
Penn  Sun4.1.3u1   DuneMUSH    19000      28Mb    14Mb uncompressed
Penn  Ultrix4.2    DuneII      2200       8.7Mb   3.8Mb uncompressed
Penn  Linux1.0.8   Gohs        2500       5.5Mb   1.6Mb uncompressed
Tiny  Sun4.1.3     TwoMoons    9350       5.5Mb   30Mb               (1)
```

(1) Info from elfchief@lupine.org. TwoMoons is based on Tiny 2.0.10p6

If you still can't decide between PennMUSH and TinyMUSH, or are fortunate enough not to care about either disk or memory space, here are the primary differences in features between Penn and Tiny 2.x. Thanks to T. Alexander Popiel for some of these.

What Penn has that Tiny does not:

- The Royalty flag, and @powers
- Multiple control and local master rooms via Zones
- A built-in mail system
- A built-in chat channel system
- A built-in building/topology checking system
- Object creation/modification times
- More extensive help and other documentations systems (news, rules, events, index)

What Tiny has that Penn does not:

- The @robot command
- The @forwardlist command, which makes creating puppets which report to multiple players a whole lot easier
- Exits within things and players
- The very handy @attribute and @list commands
- Exits are inherited from parent rooms

Of course, if you use TinyMUX or AlloyMUSH, you get the Tiny 2.x memory setup with many Penn-like features. The advantage of using TinyMUSH 2.2, however, is that its a current standard, and being actively developed - you get excellent support.

Compiling PennMUSH

Contents

Getting PennMUSH

PennMUSH up to 1.50 pl10 was developed by Amberyl. PennMUSH since then is being maintained by Alan Schwartz <dunemush@mellers1.psych.berkeley.edu>, and developed by Alan Schwartz, T. Alexander Popiel, and Ralph Melton.

PennMUSH now uses the following version-numbering scheme:

```
                          1.6.0 patchlevel 0
                          | | |            |
Code base version -----/  | |            |
Major version -----------/  |            |
Minor version --------------/            |
Patchlevel ------------------------------/
```

The patchlevel is updated when new bugs are fixed by patches. The minor version is updated when a new version of the tar file is posted; it may include bugfixes and new features. The major version is updated when major internals are changed.

You can find the most recent PennMUSH code at pennmush.tinymush.org in /pub/PennMUSH/Source.

It will have a filename like "pennmush-1.6.8.p0.tar.gz". Once you've got the file, you'll need to uncompress it (using gunzip or uncompress if the file ends in .gz or .Z, respectively), move it to an empty directory, and untar it ("tar xvf pennmush-1.6.8.p0.tar"). PennMUSH does not create its own top-level directory when untarred - be sure you untar it in an empty directory!

There may also be patches in that directory, which should be applied to the code once you've uncompressed and untarred it. They have names like 1.6.8-patch01.

PennMUSH releases are signed with Javelin's PGP public key, which is also available in the ftp directory.

Older versions of the PennMUSH source code can be found in the "oldsrc" subdirectory.

Files explained

The files below are listed as they appear in the most recent PennMUSH releases. Releases from pl10 and before have all the src/ and hdrs/ files together in the top-level directory. Filenames were shortened to 8.3 format in PennMUSH 1.6.0.

The PennMUSH source includes the following files:

BUGS
: A list of known bugs in the current release. If you fix one, let me know!

CHANGES-#
: A set of files describing changes to the code from patchlevel to patchlevel

Configure
: A program which automatically configures MUSH for your system and builds you a Makefile and some other important files.

FAQ
: Some frequently asked questions

Makefile.dist
: The distributed Makefile for compiling the MUSH (pl10 and before). You'd copy this to Makefile and modify that.

Patchlevel
: A file which tracks official patches to prevent them from being applied out of order. Don't mess with it.

README
: Compilation instructions

options.h.dist
: A file which contains compile-time options that you'll want to set. Copy this to options.h, and modify that.

dune.h.dist
: More compile-time options. Again, copy this to dune.h and modify that. (post-pl10)

src/announce.c
: The source code for a standalone port announcer. You can use this code when your MUSH is down to provide a simple message when people try to connect. But it's a bit buggy, so consider using portmsg.c, instead, described in the Appendix on Programs and Scripts.

hdrs/ansi.h
: The header file defining ANSI codes

src/atr_tab.c
: Attribute table stuff. This code handles the hashed attribute table, and contains the attribute alias table.

hdrs/atr_tab.h
: The table of standard attributes. All standard MUSH attribs are listed here. The format is described below.

src/attrib.c
: Attribute setting and getting

hdrs/attrib.h
: The header file which lays out the actual attribute (ATTR) structure

src/boolexp.c
: Working out boolean expressions (like locks!)

src/bsd.c
: The tcp/ip socket interface to the MUSH. Handles player connection to the game, low-level input/output from players, WHO and QUIT. This is where the main() function is and where the main input/output loop is.

src/chat.c
: The code for the @chat/@channel system. The list of chat channels is at the very end of this file.

hdrs/chat.h
: Header file for @chat/@channel system

src/compress.c
: A wrapper for the routines to compress and uncompress the text of attributes. This keeps down MUSH memory requirements. It either defines things for no compress, or includes compress_h.c or compress_b.c, depending on the setting of the COMPRESSION_TYPE option.

src/comp_b.c
: MUSH before 1.50pl10 used a "bigram" compression method, which is simple and unlikely to go very wrong, but doesn't give very good memory savings. This file contains that algorithm.

src/comp_h.c
: MUSH after 1.50pl10 introduced a Huffman compression method which should reduce memory requirements but be slightly slower.

src/comp_w.c
: Nick Gammon's word-based compression, required for Win32 machines. Unix sites can also use this method, which is more efficient for large databases than Huffman, but is not 8-bit clean.

src/conf.c
: Code to read your mush.cnf file and set up the MUSH's configuration

hdrs/conf.h
: Important MUSH configuration constants that are rarely changed.

src/connect.c
: A file which does nothing. Ignore it, but don't delete, since the Makefile expects to find it. This is ancient stuff.

src/convdb.c
: Code which does db conversions for db's which don't contain a Powers field and the like.

hdrs/copyrite.h
: The copyright statement

src/cque.c
: Code to maintain the MUSH queue

src/create.c
: Object creation code

src/db.c
: Code to implement the MUSH database, let you access objects, etc.

hdrs/db.h
: Definitions of objects in the database, macros

hdrs/dbdefs.h
: Definitions of macros about db objects

src/destroy.c
: Object destruction code

src/dump.c
: Code for a miniprogram which dumps all of a given user's objects out of a db. Generally ignored. Build with 'make dump'.

src/extchat.c
: This will be the source code for the extended chat system. Currently, in progress.

hdrs/extchat.h
: Headers for extended chat.

hdrs/externs.h
: Header file of externally defined stuff

src/extmail.c
: The source code for the EXTENDED_MAIL mailer, included by src/mail.c when defined.

hdrs/extmail.h
: Header file for extended mail system.

src/extract.c
: Code for a miniprogram to extract a subsection of a db. Generally ignored. Build with 'make extract'.

src/flags.c
: Code for setting, checking, and using MUSH flags is here. Also, the flag table and powers table are here.

hdrs/flags.h
: Constants representing MUSH flags and powers are defined here

src/function.c
: The function table and code for dealing with it.

hdrs/function.h

Header file for the function parser and function table.

src/fun*.c
: The src/fun*.c files (src/funlist.c, src/funmath.c, etc.) contain the definitions of PennMUSH functions of various types. If you add functions to PennMUSH, add them to src/funlocal.c.

src/game.c
: Another biggie. The MUSH command parser lives here. All commands (by players or objects) are parsed here, and the appropriate functions are called elsewhere.

hdrs/game.h
: Header file for game.c

hdrs/getpgsiz.h
: Header file to simulate the getpagesize() system call if your system doesn't have it. This is used only with the gmalloc package, and in fact behaves incorrectly on at least Ultrix 4.2.

hdrs/globals.h
: Global macros. A very small file.

src/gmalloc.c
: The source code for the GNU malloc package, one of the 3 basic malloc packages. Discussed below.

hdrs/help.h
: Header containing the structure of a help index entry.

src/intrface.c
: This file really doesn't do much more than #include bsd.c. If more kinds of interfaces were written (or if the port concentrator worked), this file would include the appropriate code.

hdrs/intrface.h
: Definitions for the interface code.

src/lock.c
: Code for the extended @lock system introduced in pl13. To add new locks, you'll modify the top of this file.

hdrs/lock.h
: Header file for the extended @lock system. To add new locks, you'll modify the bottom of this file.

src/log.c
: Code to do logging of things to the logfiles

src/look.c
: Code for look and examine

src/mail.c
: Code for the MUSH @mailer. In the post-1.50pl10 releases, this is really just a stub that selects either src/origmail.c or src/extmail.c, based on whether EXTENDED_MAIL is defined in dune.h.

src/match.c
: Code to determine if an object matches a name, with various constraints. This is where the MUSH figures out if, say, there's an exit in the room called 'out'.

hdrs/match.h
: Header file containing the match.c prototypes, which is included in other modules which intend to use matching functions.

src/memcheck.c
: Code to do some simple checking on the allocation and deallocation of memory

hdrs/memcheck.h
: Header file for mem_check.c

src/mkindx.c
: The standalone mkindx program, which creates indexes for the help.txt, news.txt, and events.txt files.

src/move.c
: Code to move objects from place to place. Commands like @tel, enter, and moving through exits.

hdrs/oldattr.h
: The old header file attrib.h, used in some db conversions

src/origmail.c
: The source code for the original (pre-1.50pl10) MUSH mailer.

src/parse.c
: The function parser

src/player.c
: Code to handle players, including creation, password-checking, names and aliases, etc.

src/plyrlist.c
: This maintains the list of connected players and aliases.

src/predicat.c
: A bunch of standard functions to check various conditions. One of the most important source code functions, controls(), is here (which determines when one thing controls another!)

src/rob.c
: Code to manage MUSH money, include give and rob.

src/rpage.c
: Code for remote-paging of other MUSHes

src/set.c
: Code for setting object attributes.

src/smalloc.c
: The Satoria malloc package, one of the 3 standard ones.

src/speech.c
: Code for messages, including say, pose, page, whisper, @emit, etc. (Doesn't include @mail or @chat)

src/strutil.c
: Utility functions for doing string comparisons and similar kinds of things.

src/timer.c
: The game timer, which determines when a dump should take place, when objects from the wait queue should be triggered, etc.

src/unparse.c
: Unparsing an object is correctly displaying its brief examine information (flags, locks, etc.) Some routines for that are here.

src/utils.c
: Various utility functions.

src/version.c
: Sounrce file which implements the @version command, reporting build time, date, flags, and MUSH version. This file is rebuilt at every compile.

hdrs/version.h
: Header telling you the version of the code

src/warning.c
: Code which implements the building warning system if USE_WARNINGS is defined in dune.h. (post-1.50pl10)

hdrs/warning.h
: Header file for the warning system (post-1.50pl10).

src/wild.c
: Code for wildcard matching

src/wiz.c
: Wizard functions and commands are defined here

win32/
: Directory containing support files for porting PennMUSH to Windows NT or Windows 95. This port was done by Nick Gammon.

RWHO/
: Directory containing the code for the RWHO library, needed if your MUSH is going to send or receive RWHO information. Just go in here and type 'make' to make the library (librwho.a).

IDENT/
: Directory containing the ident library, which allows your MUSH to get user account information for connections from sites running the ident daemon. (post-1.50pl10)

game/
: The directory for run-time game stuff, including:

game/access.cnf The access.cnf file controls which sites can access the MUSH in a various ways. You can prevent sites from connecting, creating players, and connecting to guests. You can also enforce registration and automatically SUSPECT players from certain sites. Details about access.cnf can be found in the section on Security.

game/mush.cnf
: The run-time configuration file for the MUSH. Discussed below. Many people copy this file to once called <mushname>.cnf, and modify the restart script to use that file instead.

game/names.cnf
: A list of names which players aren't allowed to use. Obscenities typically, and/or out-of-theme names.

game/restart
: The MUSH startup/restart shell script. You'll want to edit this to reflect your setup. Discussed below.

game/data
: The directory for game databases in use

game/data/minimal.db.Z
: The minimal database, containing 3 objects: #0 (Limbo, the start room), #1 (One aka God, the SuperWizard), and #2 (the Master Room)

game/log
: The directory where MUSH log files are created

game/save

The directory where backup databases are usually kept

game/txt

The directory for text files used by the MUSH

game/txt/connect.txt

Text file shown when a player connects to the MUSH

game/txt/create_reg.txt

Text file shown when a player tries to create a character from a site where creation is not allowed.

game/txt/down.txt

Text file shown when the MUSH is set to disallow non-admin logins and any player tries to connect

game/txt/events.txt

Text file containing 'events' entries. Starting with pl13, this file is built for you from files in the game/txt/evt directory. See game/txt/README for information.

game/txt/full.txt

Text file shown when a player tries to connect to the MUSH and the MUSH is full (at the player login limit)

game/txt/guest.txt

Text file shown to guests that log in. (post-1.50pl10)

game/txt/help.txt

Text file containing 'help' entries. Starting with pl13, this file is built for you from files in the game/txt/hlp directory. See game/txt/README for information.

game/txt/motd.txt

Text file shown to all players on connection.

game/txt/news.txt

Text file containing 'news' entries. Starting with pl13, this file is built for you from files in the game/txt/nws directory. See game/txt/README for information.

game/txt/newuser.txt

Text file shown when a player successfully creates a new character

game/txt/quit.txt

Text file shown when a player quits

game/txt/wizmotd.txt

Text file shown when an admin connects

Options explained

The options.h (all releases) and dune.h (post-1.50pl10) files list all of the compile-time options which can be turned on or off. Whenever you change one of these files, you should recompile the MUSH.

Options which are unclear or which deserve special mention are discussed in this section.

The EXTENDED_ANSI define turns on the ansi() function which allows players to use color and other ansi attributes in their messages. As the file notes, however, in order to protect non-ANSI players, if you use EXTENDED_ANSI, the game must filter every message output to players, and remove ansi characters from messages to non-ANSI players. This is a significant increase in CPU time required (since passing messages to players is what the MUSH spends a lot of time doing), but shouldn't be a problem for most modern machines that aren't heavily loaded.

FUNCTION_SIDE_EFFECTS are nifty, and make possible some very queue-light (albeit often CPU-intensive) code. They also make possible a lot of security problems. In general, if your coders aren't demanding these, avoid them.

PLAYER_NAME_SPACES works fine in post-1.50pl10 releases, but has some small (but annoying) bugs in releases up to 1.50pl10. If you use this, remind your players that names with spaces should always be enclosed in quotes, like "Paul Atreides".

The DBCOMP option offers essentially a tradeoff between disk usage and speed of MUSH startups and dumps. If you define DBCOMP, the MUSH will pipe its dump through a compression program of your choice (usually "compress" or "gzip"), resulting in a smaller db file and a slower dump (and restart, when it first reads the db). Compress is usually faster than gzip, but gzip gives better compression. If you have the disk to burn, I personally recommend not using DBCOMP at all, just to reduce baggage.

The NO_FORK option prevents the MUSH from forking when it dumps. Details about this are in the MUSH management section on dumps. If you define it, be sure you define a DUMP_NOFORK_MESSAGE to show players. If you don't define NO_FORK, you can choose to have the MUSH use vfork() instead of fork(), but many many systems have broken vfork()'s, so it's rarely worth it - better not to fork at all.

If you're using a version before pl13, and your system doesn't have crypt(), you should define NOCRYPT, and you'll have to uncompress and hand-edit minimal.db.Z to replace the encrypted password to #1 with an unencrypted password of your choice. The password, by the way, is stored in the XYXXY attribute. pl13 and later versions can figure this out themselves.

While the SINGLE_LOGFILE option can save some file descriptors and possibly allow more players to log in to your MUSH, it should be avoided in general as it makes dealing with your MUSH's error, warning, and other output more difficult. However, if you do want to use it, you can often separate out the individual components by using the 'grep' command and searching for NET, RPT, ERR, WIZ, or LOG.

You can set CHAT_SYSTEM to 0 (no chat), 3 (the original MUSH chat system), or 4 (the extended MUSH chat system, which doesn't require server hacking in order to preserve channels across shutdowns). If you're using an old version of PennMUSH (1.50pl10 or earlier), the MUSH can't automatically convert databases to add and remove chat channels; to add chat channels to such a database (which includes minimal.db) you must set CHAT_SYSTEM to 2, load the database, @shutdown, and recompile the MUSH with CHAT_SYSTEM set to 3 (the db has chat, and you want to use it) or, if you're feeling silly, 1 (the db has chat, but don't use it.)

As options.h says, defining MEM_CHECK allows you to test for memory leaks if you know what you're doing. When MEM_CHECK is defined, a count is kept of each memory allocation of various types which is incremented when more memory is allocated and decremented when memory is freed. You can see the list of active (allocated) memory counters by logging in as #1 and typing @stat. If an allocation from a certain function builds up and never decreases, the function may be leaking memory and you can examine it. Rhyanna provides more detailed information about how to use MEM_CHECK. Leave MEM_CHECK undefined for a normally running MUSH, as it's more efficient without.

BOOLEXP_DEBUGGING does not appear to function. Don't define it.

The DEBUG define adds info to the log file whenever a block of text that's been created to send to a player's socket (in bsd.c) is freed. If you're using smalloc.c, this also causes smalloc to try to verify the free-memory table and report the results when #1 does an @stat.

One day I'll have a working port concentrator again (I once had the old one hacked up to work right under SunOS, but not Ultrix, when I lost the code.) Until then, you can't define CONCENTRATOR unless you want to hack it into shape yourself. That requires getting a copy of conc.c and bsdconc.c from PennMUSH 1.50 pl4 or before.

Always define LOCKOUT and WCREAT. There's no reason not to, and the added security is well worth it. If you want everybody on registration status, you can just define WCREAT (and not LOCKOUT), but defining both gives you maximum flexibility.

For discussion of LOGIN_LIMIT, see the section in MUSH management on file descriptors and login limits.

Always define INHERIT_FLAG - it increases security greatly.

Many people misunderstand the ADD_NO_COMMAND_FLAG. In pl10, the flag no_command prevents an object from being check for $commands. This can improve the server's efficiency, and you get this flag no matter how you define ADD_NO_COMMAND_FLAG. What defining it does is cause all rooms and players in your database to be set no_command whenever you restart the MUSH. This is fine if you're converting an older database over, but not what you want if you're starting up from minimal.db, since any time someone chooses to make themselves (or a room) !no_command, the server will change it back to no_command next restart if this is defined. (What you *do* want to do, however, is arrange for all newly created rooms and players to be set no_command, which is handled in the mush.cnf file.)

Before pl10, the old method of dealing with newlines discussed in the OLD_NEWLINES define was not to allow \n's in attributes in the database (a \n was considered to mark the end of an attrib.) The current method still uses \n to end attributes, but allows \r\n's (carriage-return/linefeed, which appears as ^M in editors) to appear in attribs, since \r\n is what %r translates to.

The later #defines are less important constants, and generally you can follow the comments.

Making PennMUSH (post-pl10)

Always read the README file for detailed instructions. The short list is:

1. Type 'sh Configure' to autobuild a Makefile
2. Either:
    - Copy options.h.dist to option.h and edit it
    - Copy dune.h.dist to dune.h and edit it
3. Or (pl13 and later):
    - Do a 'make update' which will walk you interactively through all the options. Even better, if you're upgrading to a new

patchlevel, copy your old options.h and dune.h to your new directory, and then 'make update' - your old settings will be read and incorporated into a new options.h/dune.h. You'll be prompted for new options.

4. Edit src/Makefile and select MALLOC, RWHO, and IDENT options. Information is in src/Makefile, or read the next section.
5. Do a 'make install' to build the files and set up symbolic links
6. Do a 'make customize' (pl12 and later) if you want to customize a directory for your mush (very handy if you run multiple MUSHes). This requires perl. For example, if you customize for 'mymush', a duplicate game/ subdirectory called mymush/ will be created, containing mymush.restart and mymush.cnf already set up correctly. The databases will be named indb.mymush.Z, outdb.mymush.Z, and maildb.mymush.Z, and will live in mymush/data.

Making PennMUSH - pl10 and earlier

If you chose to use RWHO, you'll have to compile the RWHO stuff first. This is automatic in post-pl10 releases. For pl10 and before, cd to RWHO and type make to build the RWHO library.

Copy the Makefile.dist to Makefile, and edit it, following the instructions for specific systems. If you have gcc available, it's highly recommended as the compiler, as it allows you to use both -g (debugging) and -O (optimizing) flags together. If not, I strongly advise that you use your compiler's debugging flag (-g, typically) rather than the optimizing (-O) flag, so that when your MUSH crashes (and it will sooner or later :), you can determine where the problem lies.

The malloc options are not very well-detailed in the makefile. "Malloc" (memory allocation) is the function which is used when a program wants to get some memory from the operating system. It's a function which is built into all Unix operating systems, but is often inefficient and sometimes buggy. Therefore, MUSH allows you to use one of two alternate malloc packages which do more efficient memory allocation (typically by asking the system malloc for big huge chunks now and then, and then managing the doling out and recovery of memory from those chunks itself). I've used both smalloc and gmalloc successfully on Sun and Dec systems - try both and see which performs better for you. That may change as you MUSH grows. Some odder systems (NeXT, in particular) won't behave well with either of these, and require the system malloc.

If you use smalloc, you might as well define SLOW_STATISTICS, too. If you use gmalloc, and your system has the getpagesize() function, you may want to add -DHAVE_GETPAGESIZE to the SYSDEFS in the Makefile, since gmalloc only detects this function correctly if your system defines BSD, which at least Ultrix 4.2 doesn't. To build pl10 PennMUSH, type 'make install'. People often just 'make' and forget the install step, but it creates crucial symbolic links, so do it!

Troubleshooting

If PennMUSH fails to compile on your system, double check the Makefile to be sure that you've set the right SYSDEF's for your system. If you're porting to a new system, of course, you can't do that.

If you need outside help, you could post your problem on the rec.games.mud.admin newsgroup. If you do, be sure you include complete information about the hardware, operating system, compiler used, options set in the makefile, etc. It's a good idea to include a copy of the make error. You can save a copy of your make output by using a command like:

```
make |& tee make.log        (csh, tcsh, etc)
make 2>&1 | tee make.log        (sh, ksh, bash, etc)
```

Or you can use the UNIX 'script' command, which makes a copy of everything you type or see:

```
script make.log
make
(type 'exit' or ctrl-D to stop logging to the file)
```

Starting up

Contents:

The game directory

The game/ subdirectory houses the files that the MUSH uses when it's running, including database, text files, and logs. If you use the 'make customize' feature of PennMUSH pl12 and later, your "game" directory will have a customized name instead of "game", and your mush.cnf, restart, and database files may have different names as well.

The MUSH game subdirectory contains the following files and directories:

mush.cnf
: The config file for your mush (make a copy of this using the name of your mush, e.g., dune.cnf)

restart
: The restart script for starting up the MUSH

maildb(.Z)
: The @mail database - created at first startup

data/
: Directory containing your game's database(s)

save/
: Directory for old game databases

txt/
: Directory for .txt files (help, events, news, and various message files)

log/
: Directory for MUSH log files

The .cnf file

The mush.cnf file lists a set of run-time options that can be changed without having to recompile the MUSH. In the post-pl10 releases of PennMUSH, changes to the .cnf file can be made while the MUSH is running. You must send a signal 1 (HUP) to the MUSH process for the changes to take effect. Use ps to find the PID# of the MUSH process (netmush or netmud) and then kill -1 PID#.

In PennMUSH versions up to pl10, you must shutdown and restart the MUSH for changes to take effect. The non-obvious options you can set in the .cnf file are:

input_database
: The location of the input database the MUSH should read from. If you have DBCOMP defined, this is usually data/indb.Z or data/indb.gz, otherwise data/indb. If you've used 'make customize', this will have a name like data/indb.mushname.Z

output_database
: The output database. As above, with DBCOMP, usually data/outdb.Z, or data/outgb.gz, otherwise data/outdb. If you've used 'make customize', this will have a name like data/outdb.mushname.Z

crash_database
: Panic databases are never compressed. Keep the default data/PANIC.db

mail_database
: As above, this is where the mail database is located. With DBCOMP, maildb.Z (or .gz) is the default. Without, use maildb. Actually, there's an argument to be made for using data/maildb.Z and thus keeping all your databases together in the data subdirectory, rather than having maildb's cluttering your game directory. If you've used 'make customize', this will have a name like data/maildb.mushname.Z

compress_program
: What program to use for compression, usually compress or gzip. Be sure this program is somewhere on your path.

uncompress_program
: What program to use for uncompression, usually uncompress or gunzip.

player_start
: This is the room number where newly created players connect to. It defaults to 0, but you will probably want to change it for a non-social MUSH, since #0 is always a good building nexus (because rooms which can be reached from #0 are "connected" and don't give the owner the disconnected room message).

master_room
: If DO_GLOBALS is defined, then the Master Room's exits are global throughout the MUSH, and any $commands on objects in the Master Room are global. By default, this is room 2, which comes with minimal.db.

dump_interval
: The time in seconds between MUSH checkpoint dumps. For reference, 3600 seconds is an hour, 7200 = 2 hours, 10800 = 3 hours, 14400 = 4 hours, 21600 = 6 hours, 28800 = 8 hours, and 43200 = 12 hours. If you don't do dumps at least every 12 hours, you're playing with fire. If you do any kind of development or hacking on your MUSH, dump at least every 3 hours.

max_logins
: This is where you set the maximum number of players who can log in, if you have LOGIN_LIMITS defined. See the section on file descriptors and login limits for more information.

player_queue_limit
: Mess with the default of 100 at your risk. If this number is too low, players won't be able to queue enough commands to make some of their objects work. If it's too high, runaway objects may eat up too much queue space and lag your MUSH. This makes you vulnerable to a denial of service attack by a twink as well as to coder mistakes.

queue_chunk
: When the MUSH is not processing input from the net, it takes some commands off the queue and runs them. This parameter controls how many commands are run. The higher the number, the less your queue will build up, but the slower the game's net response will be. The default of 3 is usually fine, since by the time the queue's a serious issue, there's often nearly constant net activity anyway. See below.

active_queue_chunk
: When there is net activity, the MUSH still can take some commands off the queue for processing. By default, it takes off a single command, giving the queue the same priority as a single player. If you get a lot of players and find your queue building up unacceptable, raising this (start by raising it to 2) can reduce queue build-up (and therefore queue lag) at the expense of slower command response. It's a tradeoff, but it often pays when there's a slew of players and queued commands.

function_recursion_limit
: Mucking with this can allow code to waste CPU by recursing (calling itself) too many times. Encourage your coders not to write recursive code, and keep this at the default of 50.

function_invocation_limit
: The maximum number of functions that can be invoked in a single command. This can be a hassle if you try to do some kind of @dolist on all the objects in your database, but it pays to keep it down to the default of 2500.

rwho_*
: The values for the rwho parameters can be obtained by emailing jds@moebius.math.okstate.edu and asking for an RWHO password, as discussed in the README file.

log_*
: These parameters control what the MUSH will log. log_commands and log_huhs will both produce huge spammy logs - don't use them unless you're trying to identify what command is causing a crash or something. I recommend log_forces for security reasons unless you have a global which does a lot of @forcing (like some versions of Dynamic Space). I don't use log_walls, myself, but if you like to watch your wizards, you could. These can also be turned on and off with @enable/@disable, as can logins and daytime.

daytime
: The "computationally expensive" commands are: @find, @entrances, @mail/stats, @search, and the lsearch() function.

room_flags, player_flags, exit_flags, thing_flags
: These very handle parameters allow you to set up flags which should be automatically set on any object of a given type that's created on the MUSH. Having rooms and players (and possibly things) start out no_command is a good way to save CPU time that would be spent searching things for $commands. Starting players enter_ok makes a lot of sense. Some MUSHes may want to start exits transparent or players opaque or whatever. On DuneMUSH, we had a flag called "unregistered", and players started with this flag set, allowing us to do on-line registration.

The restart script

When you run the restart script, here's what it does:

1. Sets some variables to use later. You must define GAMEDIR to point to your game directory and CONF_FILE to point to your .cnf filename. If you've used 'make customize' these will be set for you.
2. Scans your .cnf file and figures out the name of your incoming and outgoing databases, and the compression scheme you use.
3. The "mush" variable is set to 2 if there's already a MUSH running, or 1 otherwise, by using a tricky little ps/egrep command line. If you run multiple MUSHes, change the line to check for the full name of the conf file of this mush, rather than just the string "conf" (again, 'make customize' does this automatically.) If the MUSH is already running, restart reports that, and quits.
4. The help.txt, news.txt, and events.txt files are re-mkindx'd. In pl13, this is handled by running a 'make' in the txt directory.
5. If there's a PANIC.db, and it's complete (i.e., the game was able to finish the whole panic dump), we turn it into an outdb.
6. All the log files are erased. You may want to change the script to preserve some of these.
7. Tries to find a database to start up, in this order:
    - If there's an outdb, indb is moved to save/indb.old, and outdb is moved to indb.
    - If not, and if there's an indb, we use that.
    - If not, and if there's a save/indb.old, copy that to indb and use that.
    - Finally, if we can find minimal.db.Z, we'll use that.
8. The MUSH is run.

The txt/*.txt files, @readcache, and mkindx

PennMUSH uses two kinds of text files, both of which live in the game/txt subdirectory: cached text files and indexed text files.

Cached text files

The cached text files contain various messages which are displayed to players under different circumstances. The MUSH reads these (typically short) files into memory when it starts up ("caching" them), and when you change these files, you must use the @readcache command on the MUSH to update the memory versions. These files include:

connect.txt
: The connection screen. This file should give the name of the MUSH (and usually the source code version), an email address for the God or contact person, and character connection/creation information. It should also include the "Type WHO to see who is currently connected" line that's in the default connect.txt - MUSH clients like tinyfugue use this line to detect when to start displaying text when they're doing quiet logins.

quit.txt
: Shown to players when they QUIT or LOGOUT

down.txt
: Shown to players when logins are @disabled (i.e., when only admin can log in). Often unused in favor of @motd/nologin, which doesn't require site access.

newuser.txt
: Shown to players when they create a new character.

motd.txt
: Shown to players after they've connected to their character. Often unused in favor of @motd, which doesn't require site access.

wizmotd.txt
: Shown to admin when they connect. Often unused in favor of @motd/wiz, which doesn't require site access.

In the post-pl10 releases of PennMUSH, sending a signal 1 (SIGHUP) to the MUSH process (see .cnf file above) also causes the MUSH to re-cache the cached .txt files, just like using @readcache.

Indexed text files

The indexed text files are help.txt, news.txt, events.txt, rules.txt, and index.txt, depending on what you've got defined in options.h and dune.h These files are often long, so to speed up access, the MUSH uses index files which help it quickly find where in the file a given topic is. The index files, help.indx, news.indx, and events.indx, are built with the mkindx program (e.g., mkindx news.txt news.indx), and must be rebuilt whenever you change one of these files.

In pl13 and later versions, the help.txt, news.txt, and events.txt files themselves should not be edited, because they are built from files residing in the txt/help, txt/news, and txt/events directories. This allows you to keep your local help files separate from those distributed with PennMUSH, and to manage them as a set of files rather than a single large file.

Here's how it works:

1. The source files for help.txt, news.txt, and events.txt are kept in directories called help, news, and events respectively.
2. Files in those directories which end in .<directoryname> are considered to be part of the text. That is, files in help/ which end in .help, and files in news/ which end in .news will be merged into the final help.txt, news.txt, or whatever.
3. When the MUSH is restarted, a 'make' is run in this directory. For if a file in help/ is newer than help.txt, it calls compose.csh to rebuild help.txt. If you've got perl on your system, compose.csh will also call index-files.pl to make a 'help index' entry which lists all your help entries.

So, if you want to add your own news entries, make a file called news/local.news and put 'em there. Or maybe organize it into parts: news/theme.news, news/code.news, etc.

The directions which follow about formatting these files still applies to the files in the help/, news/, etc. subdirectories.

The format for the indexed text files is a topic line (or set of lines) followed by the text of the topic. Topic lines begin with "&". The topic "& help" (usually first in each file) is shown if the player types "help", "news", or "events" without any topic.

Topic names with themselves beging with an & denote admin help or news topics, accessible with the ahelp or anews commands.

Here's what a piece of a news file might look like:

```
& rpg           <-- note that both topics will work for this entry
& game
```

```
You've typed 'news rpg' or 'news game'.
This is the player information for the role-playing game...

& &rpg
You've typed 'anews rpg', and you must be an admin.
This is the admin information for the role-playing game.
```

Caution: Only topic lines should start with &'s. Lines of the help entry which need to start with &'s (like programming examples of the form: &test me=....) should have at least 1 space before the & so that it's not the first character on the line.

Minimal.db and first startup

Now you're ready for that first startup. Go to the game directory and type 'restart'.

To monitor the restart, type: "tail -f log/netmush.log", which will show you the progress of the startup. When it reaches "MUSH startup completed" (or an error message!), hit ctrl-C to kill the tail process. (If you got an error, see Troubleshooting, below).

In general, you can check to see if the mush is running by using "ps -ux" (on BSD systems) or "ps -lf" (on SysV systems), and looking for the process named "netmush" or "netmud".

Assuming you've succeeded in getting the MUSH up and running on your chosen port, here are some things you should do immediately:

1. Log in as One (password 'one') and change One's password. (Many people also rename One to God or some other appropriate name). This is very important, as #1 is the only player which can confer wizbits on other players, and needs to be protected. @dig a floating room for God to live in, and @tel and @link #1 there.
2. Make a wizard character for yourself. You can do it while logged into #1 by using @pcreate name=password, @set *name=wizard. Once you've done this, you can log out of #1 and into your wizard character. You'll need #1 later, but it's best not to use God unless necessary.
3. Create some guest characters, if you like. You can do this with @pcreate as above, and @power *name=guest. I typically use the following kind of code:

   ```
   @dol Guest Guest2 Guest3 <etc> =
           {@pcreate ##=guest; @power *##=guest;
            @lock *##=##; @elock *##=##; @desc *##=A guest. Be kind.;
            @atrlock *##/describe=on}
   ```

4. If you're bringing up minimal.db, and you had CHAT_SYSTEM defined as 2, do an @shutdown, recompile the MUSH with CHAT_SYSTEM defined as 3, and restart again.

Troubleshooting

The MUSH crashes

If the MUSH actually crashes (leaving behind a file called "core" in the game directory), see the section on MUSH crashes. The problems addressed here result from the MUSH failing to load the db, and exiting cleanly.

Problems loading minimal.db

Most problems loading minimal.db are caused by misconfiguring one of 3 of the options.h compile-time options: CHAT_SYSTEM, ADD_POWERS, and DBCOMP. If you've got one of these problems, you'll see a message like:

```
Bad character 0 on object 0
Bad attribute list object 0
```

The CHAT_SYSTEM problem applies only to releases before "dune-2". As discussed above, minimal.db as distributed does not have the chat field set up. It therefore will not load unless CHAT_SYSTEM is defined as 0 or 2. If you set CHAT_SYSTEM to 2, and convert minimal.db to use chat channels (or if you get the minimal.chat.db.Z which is available from the archive), future startups should be done with CHAT_SYSTEM set to 3 (that is, after shutting down the first time, set CHAT_SYSTEM to 3, and recompile the MUSH).

The minimal.db distributed with pl10 already has the @power field added, and should not be run with ADD_POWERS defined. ADD_POWERS adds the @power field to older dbs.

If you have DBCOMP defined, be sure that minimal.db is appropriately compressed or gzip'd, depending on how you have the compression_program defined in mush.cnf. If you don't use DBCOMP, be sure that your startup db isn't compressed. :)

In some cases, the problem might also be that the symbolic link from hdrs/dune.h to dune.h in the top-level directory (or from hdrs/options.h to options.h in the top-level directory) has been broken. In that case, you'll want to do something like this:

```
cd hdrs
rm dune.h options.h
ln -s ../dune.h dune.h
ln -s ../options.h options.h
cd ..
make
```

Problems loading your db

"Bad character on object xxx" - the bane of all MUSH Gods. If you've had this db working before, and you haven't modified the source code, this indicates there is some corruption of your database. To recover, you'll want to locate the corruption by hand-editing the db. Here are the steps:

1. MAKE A BACKUP OF THE DB! Do this every time. It's very very easy to make a mistake. I put mine in game/save, with names like indb.(date).corrupt
2. If your db is compressed, uncompress it.
3. Load your db into your favorite editor. Some editors can not handle some db's, usually because the lines of text in the db are too long for the editor's buffer (Sun's vi can have this problem), and sometimes because the db itself is too large (20Mb is a big file :). GNU emacs version 19 works pretty reliably. If the problem is the size of the file as a whole (not line length), you can use the unix 'split' (split indb) command to split up the db into 5000-line chunks which you can paste back together when you're done using 'cat x?? > indb.new'.
4. Find the offending object. The beginnings of objects are indicated by lines which start with !object# (for example, "!512"). The next set of lines which contain numbers represent the object's type, flags, powers, chat channels, locks, etc.
5. If you're lucky enough to know the attribute on the object that's causing your trouble, you can find it. Attributes start with a "]" symbol (for example: "]STARTUP"). If you just know the bad character, start searching for that character in the object. The attribute name is followed by some owner and flag information (separated by ^'s) and then on the next line, the text of the attribute itself, which is a single line but may contain literal carriage returns (^M's).
6. When you find the problem character, try to fix it. The most reliable fix tends to be deleting that attribute (the]attrib line, and the line following it which contains the character), but if you see something strange (control characters besides ^M, hard returns in attributes, etc.) you could try just fixing that and see if it helps.
7. Quit the editor, saving the new db, recompress it if necessary, and try starting up under it. Repeat as above until it works. :P

MUSH Management

Contents

Using #1

God, player #1, is a very special Wizard, and a very useful tool. God can do some things that no other Wizard can, and some commands function differently for God. Because of these powers, it's wise to safegaurd God's password at least as well as you would the MUSH account's password. While the same can be said about any Wizard password, it goes doubly for God.

Making wizards

God is the only thing in the game which can set a WIZARD bit on a player. When you recruit Wizards for your MUSH, you'll have to

log in as God to grant them the bit. For that matter, only God can remove a wizard bit or @nuke a wizard player, a definite safety feature.

Malloc stats

If you have MEM_CHECK defined, or are using smalloc.c and have DEBUG defined, running @stat as God will give you a dump of memory allocation information internal to the game. This can be useful in debugging memory leaks and the like.

@startup for @functions

When the MUSH starts up, the @startup attribs on all objects are triggered, in ascending order of db#. This means that God's @startup is the first one triggered, which makes it ideal for doing @function commands, since later @startup's may rely on your @functions being installed.

Schmidt@Dune (and elsewhere) developed this code for #1's startup to make automatic @function setup easier:

```
@STARTUP #1=@dolist lattr(#191/FN_*)=@function [after(##,FN_)]=#191,##
```

(In this case, object #191 had all the global functions on attribs call FN_FOOFUNC, FN_BARFUNC, etc.)

Miscellany

- Only God can use the @poor and @allquota commands.
- Only God can use @log/wipe to erase a game log, and this additionally requires the password for the game account.
- God may @chown a player, which should only be used if, due to some corruption, a player ends up not owning themselves. God can also @boot players who the game lists as not connected. Don't do it.
- God can @wipe wizard-only attributes on things. Even wizards can not (using @wipe - they can clear them manually).
- Only God may @set, @edit, @tel, @force, or @trigger God.
- In the post-pl10 extended mailer, only God can do @mail/nuke and wipe all @mail. And only God may use the mail() function to read other players' @mail.
- In the post-pl10 releases, God may change a player's name (using @name) without giving the player's password.

Shutdowns, dumps, and paranoia

What's a dump?

A database dump is the process of making a copy of the MUSH's database (which is in memory when the MUSH is running) on the machine's disk. This is absolutely necessary when the MUSH is shut down, so that it can be restarted in the same state it was in when it was shut down. In addition, "checkpoint dumps" are performed by the MUSH at regular intervals (e.g. every hour, every 3 hours, every day, or however you configure it in mush.cnf). If the MUSH should crash, it can be restarted from a checkpoint dump, and data loss will be restricted to changes made in the last interval. Finally, a dump can be performed by a wizard using the @dump command.

What's a forking dump?

In options.h, you can decide if your MUSH should fork when it dumps. "Forking" means that the MUSH makes an identical copy of itself in memory, and that copy runs the dump and then exits, while the original copy continues doing MUSH things. This means that players won't even notice when your MUSH dumps, which is nice.

There's a downside, however. When the MUSH forks, there are basically two MUSH processes running. That means durnig the dump, your MUSH is using twice the CPU (roughly) and twice the memory that it normally would. Unless your machine has a lot of memory (or your MUSH is very small), this can cause the MUSH to swap, and lag everyone badly.

The alternative is to dump without forking. While the MUSH is dumping, it can do nothing else in this case, so the game will pause until the dump is complete. It gives the players a message to let them know that there will be a pause.

What happens on a shutdown dump?

A shutdown dump is just like any other dump, but it occurs at shutdown. :)

What's a paranoid dump?

A paranoid dump is performed by a Wizard typing @dump/paranoid. A paranoid dump differs from a normal dump in that it automatically corrects certain kinds of database corruption, writing out a non-corrupt database to disk. This is often useful when your database is corrupted in such a way that normal dumps cause a crash. You can @dump/paranoid, kill -9 the MUSH process (DON'T @shutdown, which will overwrite the paranoid dump with a corrupt one), and then restart from the paranoid dump.

What exactly does a paranoid dump try to fix?

- Bad attribute names: Unprintable characters or spaces in names of attributes will be replaced with a '!' character.
- Bad attribute owners: If the attribute is owned by an invalid db#, its owner will be set to God.
- Bad attribute text: If the attribute's text contains unprintable characters or hard newlines (\r's and \n's), they will be replaced with '!' characters. The fixing of hard newlines used to be important, but is actually now more of a hassle since MUSH can now deal with \r\n's in dbs.

In the post-pl10 releases, if DB_FIXING is defined, @dump/paranoid attempts to correct not only the disk dump of the database, but the memory copy as well, hopefully repairing things enough that you can dump or @shutdown normally. It can only correct bad attribute owners so far, but that sort of corruption is the most likely to cause an actual crash while dumping.

What's a panic dump?

A panic dump occurs when the MUSH process is terminated with a signal 15 (SIGTERM). This is a pretty unusual thing, but if someone accidentally kills your MUSH process somehow, it'll attempt to save a copy of the database before it dies. The copy will be stored in game/data/PANIC.db, and will be uncompressed. You can tell if the dump was successful by examining the last line of that file. If it's ***END OF DUMP***, it was successful. The restart script knows about panic dumps, and will try to restart from a panic dump if it can find a successful one.

File descriptors and login limits

How many players can be connected to a MUSH at once? There are 3 factors which affect the answer to this question.

The operating system

Each player connection comes in over a TCP/IP socket, and each socket require that the operating system allocate it a "file descriptor", a number which refers to that socket. In addition, each file which is opened by the MUSH process (standard input, standard output, standard error, one for each log file, one reserved for reading help/news/events files, one to accept connections) requires a file descriptor.

Most operating systems have limits on how many file descriptors a user's process can open. Some typical limits are 64, 128, or 256. Sometimes there will be a 'soft limit', which is lower than the maximum, and which you can increase (via the 'unlimit' command in the csh) up to the 'hard limit', or system maximum. The hard limit can only be raised by the system adminstrator recompiling the system's kernel. This is not too bad to do under SunOS 4.1.x (NOFILE in sys/param.h), Ultrix 4.2 and later (NOFILE in h/param.h and max_nofile in conf/{mips,vax}/param.c), and Linux, where you always have kernel source. Note: SunOS 4.1.x systems have (by default) a limit of 256 descriptors, but there's a bug in the stdio code which prevents descriptors after the first 128 from working correctly. :(

When you run out of descriptors and the MUSH tries to open another one, it will likely either crash or hang. Since the MUSH requires about 10 descriptors for itself (6 if you define SINGLE_LOGFILE), subtract that from the hard limit to figure out how many players you can support.

CPU/memory/network bandwidth

There's also a practical limit. The more players you have connected, the more work the CPU has to do to service the process, the more memory will be used (for player objects and for all those queued commands), and the more load will be placed on your network connection. At some point, your game will begin to lag increasingly. Once again, little can be done unless you have a budget for better hardware.

Setting login limits

Finally, you can set a limit on how many players your MUSH will accept in your mush.cnf file, if you've defined LOGIN_LIMIT in your options.h file when you compiled. Usually, you just want to set your login limit at the highest practical value, taking the above things into consideration. So, if you've got a max of 64 file descriptors, and you don't define SINGLE_LOGFILE, figure you can have

54 players connected, and set login_limit accordingly. Remember, however, that your admin and players with the login @power can connect even over your login_limit, and if they run out your file descriptors, expect bad things to happen.

Wizard commands in detail

Most of the Wizard commands are covered in the Wizard's supplement to Amberyl's MUSH Manual, but I though I'd mention a few that deserve special notice.

@comment
: The COMMENT attrib can't be seen by mortals, so using @comment to set comments on suspicious players is a common thing to do. To maximize usefulness, consider a policy whereby whenever a player is set suspect, they must be commented, and comments must include the name of the commenting Wizard. If you let any admin set SUSPECT, consider a +suspect player=comment command which sets the flag and the comment at the same time.

@disable/@enable
: The options which you can enable and disable are: *logins* (when disabled, only admin/login-@power can log in), *daytime* (when enabled, cpu-intensive commands can't be used), *command_log* (when enabled, all commands are logged), *force_log* (when enabled, all @forces are logged), *huh_log* (when enabled, all commands resulting in Huh? are logged), *wall_log* (when enabled, all @walls and @wizwalls are logged).

@fixdb
: This very dangerous command allows you to directly set the contents, location, exits, and next elements of a database object. No error-checking is performed, so be sure you know what you're doing. To learn more about these elements, read the source code and play around with the examine/debug command which displays values from a db object's structure which are not normally accessible in raw form.

@kick
: As the help says, @kick causes the immediate execution of a specified number of commands from the queue. The real question is when to use it. It can help if your queue is clogged with a non-infinite loop, like a large @dolist. But what often happens is that while the game is busy executing the @kicked commands, commands from connected players are building up, resulting in a large queue clog even after the @kick. I use it as a last resort, and rarely when many players are connected.

@log
: This works just like the help says it does, and is really handy if you want to keep track of things (like when players die by your combat system or use certain commands). The alternative, logging such things to some db object, has all sorts of problems as the log gets bigger and bigger. If you use @log, classify everything you log with some unique KEYWORD so you can get it from your logs with a simple grep. I used to have an object which would @log the number of connected players and admin, and how many were IC, every 30 minutes, and I'd make histograms out of the data in the log. :)

@pcreate
: This is how you create characters for people from sites on registration status.

@readcache
: Don't forget to run this command if you change the cached .txt files and want to update them (or, in the post-pl10 releases, kill -HUP the MUSH).

@sitelock site-pattern[= options]
: Adds the site-pattern directly to the access.cnf file with the specified options. Wizards can use this to lock out sites. Later uses of @sitelock override earlier ones.

@squota
: In the post-pl10 releases, you can @squota player=+number or -number, to adjust their quota up or down from its current level.

@uptime
: On some operating systems, this command's functionality is limited. On others, the command works, but seriously lags the MUSH while it's working. Test it once and see.

@toad
: I'm a fan of disabling this. Wizards shouldn't nuke players, they should boot and newpassword (and maybe sitelock), all of which are reversible. Only God should nuke players, and God should be secure enough to do it with the simple @nuke, not the flashy @toad (which leaves a slimy toad you often nuke anyway). Rhyanna@Castle D'Image notes, however, that @toad can be a convenient way for RP MUSHes to make corpses after IC deaths.

MUSH crashes

Ok, your MUSH crashed. It happens. The good news is that if you compiled with debugging enabled (-g), you can usually figure out *where* in the code it crashed and *why*. Even if you personally can't correct the problem, such information is exactly what you'd want to post/mail if you were asking for help.

In order to get debugging information, your MUSH crash must have produced a core dump, which is a file called 'core' in your game directory. This file is usually quite large. On some systems, you must be sure that your MUSH restart script does an 'unlimit

coredumpsize' or you won't get complete core files.

You also need a debugger. Most systems have one of the following debugging programs: gdb (ideal), dbx (a good second choice), sdb (ick), adb (very ick). You (or your sysadmin) can ftp the source code for gdb from the GNU ftp site.

Using the gdb or dbx debugger

gdb and dbx are similar, so in the following examples I'll use them interchangably. If you'll be needing a copy of your debugging session to mail out, use the unix 'script' command to log everything ('script buglog', do your debugger stuff, then 'exit').

To run the debugger, go into your source directory and type:

gdb netmud game/core

The debugger will tell you how the process died, and what function it was executing at the time.

Getting and using a stack trace

That alone can be useful, but far more useful is a stack trace, which shows not only the function it was executing that caused it to crash, but which function called that fatal function (and with what parameters), which function called that, and so forth, back up to the very top function, main().

The command 'where' will print out a stack trace. The commands 'up' and 'down' will walk you up and down through the stack, showing the values of parameters that are being passed. You can also use the 'p' (print) command to print the values of other variables in each function.

Running the MUSH under gdb

If the crash occurs during the MUSH startup process (when analyzing or loading the db, for example), and doesn't leave a core dump, you have another option if you have gdb. You can actually run the MUSH from within the gdb debugger by going to the game directory and doing:

```
gdb netmush
run mush.cnf
```

(According to Rhyanna, on some operating systems, typing 'handle SIGPIPE nostop' before running the mud will prevent the debugger from reporting SIGPIPE signals when players connect.)

When the MUSH crashes, you'll be told how and where, and be returned to gdb to analyze the crash. Gdb and dbx can do much more - you can tell them to run the MUSH until they hit a certain function and then wait for further instructions, step through code one line at a time, etc. Read the manual pages if you need to do these things.

Trashed stacks

Some crashes, alas, overwrite the memory containing the function stack, and produce a totally unhelpful core file. The only help I've found for this is to cross my fingers and run the MUSH again, hoping that the next crash will pick a different memory location to overwrite. In some cases, 'touch'ing all the MUSH source files (touch *.c) and recompiling can help, but it's mostly superstition.

Lagging

One of the most pernicious problems common to MUSHes is lag, the condition in which the MUSH feels slow in responding to player input. There are a number of things which contribute to lag, and once you identify the culprit, you can decide if you can improve the situation.

Network lag

Network lag is caused by difficulties in the network connection between the player and the MUSH host machine. For example, a router on the internet between the two might be dropping packets, or a segment of the network might be overloaded with packets.

The characteristic property of netlag is that you won't experience it if you're connecting to the MUSH from the MUSH machine itself. If you can get a lagless connection by doing a 'telnet localhost <port#>', netlag is responsible.

If your host has the *ping* command (most do), you can test how long it takes a packet to travel between your host and another machine, and try to identify how slow things are going to be. If you happen to have the *traceroute* command, you can see exactly where (between which routers) the network is lagged.

Unless you happen to be a network administrator of the problem stretch of network (or if it's just the local connection to your machine), there's not much you can do. If the problem is your particular net connection, you can probably spend more money and get one with a higher bandwidth, or reduce other things your machine does that requires the net (email, etc.), but neither of these are really worth it for a MUSH, usually. :(

DNS and IDENT lag

Another network-related source of lag involves domain name service lookups. When a player connects to the MUSH, the MUSH knows the player's IP address, and queries the DNS to get the player's hostname. This is called a "reverse hostname lookup".

Some hosts, however, have very slow nameservers. Sometimes this is because the nameservers are behind slow internet connections with heavy traffic. The MUSH stops while a reverse hostname lookup is going on, so if it takes more than a second, you will feel lag.

If you've got your MUSH configured to use IDENT lookups, the same kind of problem applies. In addition, IDENT lookups of non-unix systems can hang until the lookup times out.

There are a few ways you might deal with this problem:

- Turn off ident or reduce the ident_timeout value in mush.cnf
- If you don't need hostnames, set use_dns to "no" in mush.cnf (PennMUSH 1.6.x) and no reverse hostname lookups will be performed.
- If there are particular sites that are causing you trouble, and if you have access to your system's hosts file (/etc/hosts on most Unix systems), you could try making an entry for the troublesome system in the hosts file. Many DNS setups will check the hosts file before asking the nameserver.

A future PennMUSH release may avoid this problem by having a separate process handle DNS lookups (and IDENT lookups, for that matter).

CPU lag

CPU lag is caused when the MUSH machine is having to split its time doing many tasks or tasks which require a lot of running time spent in the CPU. If your MUSH is on a machine which has a lot of users, this is more likely. If the users are programmers who run things like compilers regularly, this becomes much more likely.

You can examine the CPU load in a few ways. The *uptime* command will display an interesting, if non-objective statistic called "load average", measured over the last minute, 5 minutes, and 15 minutes. If you know what typical load average looks like, you'll be able to recognize abnormally high load. Loads over 3, especially in the 5/15 minute entries, tend to make for a slow game.

But what's causing the load? Here you can use *ps -aux* (BSD) or *ps -elf* (SysV) to see all the running processes and how much CPU time they're getting at that moment. This static picture can be deceiving, but is a good start. Read the man page for details.

If you're responsible for the load due to your own compiling and such (or if you need to decrease the CPU load your MUSH puts on the machine), read the man page for the *nice* and *renice* commands, which let you tell the system that your compilation (or MUSH process) should be nice about using CPU, and the CPU should give it lower priority.

Disk swapping

Unix systems have a limited amount of memory in which to run their programs. Memory is also expensive. So unix systems use a part of the disk as "virtual memory" or "swap space". Processes send parts of their memory that haven't been accessed in a while off to virtual memory, a process called paging. Paging helps make all the programs work together gracefully.

MUSHes can be pretty big programs, though, and sometimes another program (some compilers and editors, as well as statistical software and other such things, for example) has to be granted more memory than can be recovered even with paging. In these situations, the whole MUSH process may be "swapped out" to the disk, temporarily put on hold until it can be swapped back into memory. While the MUSH is swapped out, the game is frozen, and if it stays swapped long enough and often enough, you experience lag.

Dealing with swapping and paging is beyond the scope of this guide. Read the man pages for *ps*, *vmstat*, *pstat*, and *iostat*, or pick up a book on Unix System Administration or Performance Tuning (say, the O'Reilly Handbooks, which are great) if you're the system administrator of your MUSH machine. If not, purge unused objects from your database and hope.

Queue lag

Queue lag occurs when the MUSH's queue becomes clogged, and the MUSH can no longer keep up with servicing the queue and player input. Players get priority, so keyboard response will be good, but if they try to use $commands, which go on the object queue, they won't get response for some time.

@kick can be used as a temporary fix for this, though see the concerns above in the section on Wizcommands. A more permanent solution might be to adjust the values of queue_chunk and (especially) active_queue_chunk in the mush.cnf file. This will make keyboard response slighly worse, but will usually fix the queue clogging.

Security

Security on a MUSH (which many feel is an oxymoron already) addresses the needs of the game to provide four or five crucial factors:

Stable service
: Players should be able to play the game - it should be up reliably.

Database control
: Players should not be able to modify objects belonging to other players.

Privacy
: Private player information (email addresses, sites, private conversations) should be inaccessible by unprivileged players.

Freedom from harassment
: Players should be free of harassment from other players (gross spamming, for example.)

A fair game
: Players should not be able to use MUSH knowledge to cheat in any game run on the MUSH, particularly if the game is the point of the MUSH.

Unwelcome players can generally be broken down into two major categories: crackers, who threaten stable service and database control, and twinks, who threaten privacy, freedom from harassment, and game fairness. A cracker seeks to destroy the integrity of the game by making it unusable. A twink seeks to destroy the integrity of the game by making it unenjoyable or unfair.

A cracker philosophy

Some people who try to compromise MUSH security claim that they're doing you a service, by showing you where you MUSH needs shoring up. As Amberyl once put it, this is akin to someone breaking into your house (unasked, of course) to show you that your locks should be repaired.

These folks not only put your game and database at risk, not only might become privy to sensitive information (player email addresses, etc), but destroy the trust which generally exists within the internet and MUSH community. It's no service.

Unfortunately, the crackers usually win, because they have a lot of time to spend trying to compromise your security, while you and your Wizards have more to do than just continually improve it. The goal here, therefore, is to emphasis some easy measures that you can take to deal with attacks on your MUSH.

A twink philosophy

While the goal in dealing with crackers is to keep them away from your MUSH or minimize the damage they can do, twinks come in many flavors and require individual response. Many twinks are misguided newbies who, once set straight, become valuable players. Others merely seek to disrupt the game, creating dozens of characters and paging regular players with annoying messages. As Gilbert and Sullivan put it, with twinks it's often best to "let the punishment fit the crime."

SUSPECTs

The suspect flag makes an excellent first line of defense against suspected problem players. In addition to allowing you to tell when a SUSPECT player connects, disconnects, or changes their name, all actions by SUSPECT players are logged in the MUSH's command.log file, so you can decide if they really pose a threat.

There's a script by Talek in the Appendix which I've used to process command.log and create individual suspect files, one per player (by db#), which can be read or searched for keywords like "hack" or whatnot.

By the way, I think it's good policy for every MUSH to place a statement in news ('news policy' perhaps) noting that while every attempt is made to preserve privacy, you reserve the right to log commands executed in order to maintain MUSH security and/or to fix bugs.

Registration and lockout

Site registration, and site lockout, are strong measures which can be very effective in preventing twink invasions. The key to site-based access control is the *access.cnf* file.

You can control the following aspects of access:

- Whether or not a site can connect to guests
- Whether or not a site can connect to non-guest characters
- Whether or not a site can create characters at the login screen
- Whether or not a site can use the "register" command at the login screen, to request that a character be created and its password be emailed to the player.
- Whether or not a site is suspect. Players who connect from suspect sites are automatically set SUSPECT.
- Which of the above your Wizards can override using @sitelock, and which are set in stone.

access.cnf (in the game directory) is a list of sites and their access control options. When someone connects, the file is read through in order, and the first line which matches their site is used to determine their access options. Order of the file is thus critical.

A line in the file looks like this:

```
   [
...] [# Comment]
```

Host patterns may contain the wildcard "*" which will match any number of characters. Patterns can match individual users from sites which run ident servers, but because ident relies on trusting the server and a reasonably speedy network, it's best avoided. Using "*" as the host pattern matches all hosts, and really only makes sense as the last line of the file.

A host which does not match any line in the file is allowed complete access. A host which matches a line with no options listed is banned completely; connections will dropped before the login screen.

There is also a special line in the file:

```
  @sitelock
```

The "@sitelock" line marks where new additions to the file via the @sitelock command will be added (if your file doesn't contain one, one will be added at the end of the file when someone uses @sitelock). Any lines listed above the @sitelock line can not be overriddent by @sitelock; those listed below can. This allows you to decide which access control rules your Wizards should be able to override.

The available options include:

create
: allow players from this site to use the 'create' command

!create
: don't allow players from this site to use the 'create' command

connect
: allow players from this site to connect to non-guest characters

!connect
: don't allow players from this site to connect to non-guest characters

guest
: allow players from this site to connect to guests

!guest
: don't allow players from this site to connect to guests.

register
: enable the 'register' command at the connection screen. Players may type 'register ' to create a new character and have its (randomly generated) password emailed to the player. Note that this does *not* disable 'create', so you'll probably want to use this option along with the !create option.

suspect
: cause all players connecting from this site to be set SUSPECT. Unless you want SUSPECT guests, you'll probably want to include !guest as well.

The @sitelock command sets the optional comment to inform you of who performed the @sitelock and when.

access.cnf is re-read when the MUSH receives a HUP signal. It is updated whenever someone uses @sitelock.

The file *game/access.README* contains examples of using access.cnf to enforce a variety of different access control policies.

Gagging, booting, guesting, newpasswording and nuking

There are a wide variety of things you can do to discipline a twink player. Amberyl's wiz-ethics lecture discusses many of these, and I, like her, believe that one should "let the punishment fit the crime" which choosing which command to use:

The GAG flag
: Prevents the player from speaking, a handy response to spammers and those who page obscenities.

The Guest power
: Restricts a number of commands players can use, especially if you're running the post-pl10 releases with HARSH_GUEST defined, in which case they can't do any building or attribute-setting.

The FIXED flag
: In the post-pl10 releases, this flag prevents the player from using the @tel and home commands. Handy if they keep @tel'ing where they're not wanted.

@boot
: Disconnects the player from the MUSH. Nothing, of course, prevents them from reconnecting, unless you lock their site or install a global @aconnect to listen for them connecting and boot them again.

@newpassword
: Changes a player's password. This is the proper way for non-Gods to "nuke" a player, as it doesn't actually result in any data loss, and can be reversed if the twink makes amends. This does no good against twinks who are just creating scratch characters they don't care about. In the post-pl10 releases, God can also re-name a player without knowing/changing their password, which can have similar effects.

@nuke
: This is how God cleans players from the database. It's also an appropriate response (along with a sitelock) against crackers.

Coded attacks on the MUSH (privacy, denial of service)

Most twinks just want to rise to ultimate power in your game. But some crackers are interested only in compromising your security in order to destroy and deny the game to others.

Admin passwords **must** be protected. A cracker who can log in as a Wizard can destroy your database in one command.

Players should not be given wizard-bitted objects if at all possible. Wizard-bitted global commands should be carefully checked for security. One important check is to try calling the command with arguments like "[set(me,visual)]" and "\[set(me,visual)\]" - if the function results in the wizobject becoming visual, it's not secure! (And Wizard globals shouldn't be visual anyway, just in case.) Best is if you can run without FUNCTION_SIDE_EFFECTS.

Be very careful of zones. My policy is that nothing owned by an admin should ever be zoned to a ZMO which has non-admin on the elock. Instead, it should be @chowned to a mortal builder-character and then zoned. You can find ZMO's without elocks by logging in as a mortal and running an @find to see what you control.

In the worst-case scenario, if you somehow lost your entire database, you'd still be okay if you've been making backups of your database every now and then (how often and how many depend on your disk space.) Always have at least one working backup somewhere. And chmod a=r it, too, so you don't accidentally overwrite it! If you really want disaster-recovery, you should keep a database backup on *another* machine as well, in case your host should crash.

Making Guests

If you want to encourage people to visit your MUSH and try it out, make some Guest characters available. Guests are severely limited in the commands they can use, especially if the (post-pl10) HARSH_GUEST define is enabled.

Post-pl10 PennMUSHes can have as many Guest characters as you see fit. Each guest should be set @power Guest. When a player logs

into a Guest-@powered character, the following happens:

1. If there's no one else connected to that Guest, the player is connected.
2. Otherwise, the server searches through the database for another Guest-powered character which isn't in use. If it finds one, the player is connected to that Guest.
3. Otherwise, the server gives up, and connects the player to the original Guest (resulting in multiple players controlling one character)

A typical setup is to create 5 or so Guests with names like "Guest", "Guest2", "Guest3", etc., and ask players to connect to simply "Guest", and let the server assign them to an unused one.

PennMUSH pl10 supports setting the Guest power on multiple players, but doesn't do the autoconnection trick. Pl9 and earlier allow only a single Guest, whose db# is specified in the mush.cnf file.

Everyday stuff

Whenever you log into the MUSH or the MUSH account, there are a few things that are worth doing as part of your regular routine. This is just about making sure that things look fairly normal.

@stat

One simple thing that's work a minute every time you connect is to do an @stat command to see how many objects are in the database. If you find that it's jumped 2000 in a day, you might suspect somebody's been mucking around. If you find a lot of garbage, somebody's been nuking a lot of things. If you find a lot of players and know you don't have that many active players, maybe it's time for a player purge (discussed below.)

@uptime

If the @uptime command doesn't lag your MUSH, running it can give you some useful information. In addition to the load average (discussed above under lag, it reports the PID# of the MUSH process, a bunch of memory statistics, of which the most useful is probably Max res mem (maximum resident memory used by the game, in bytes), a lot of i/o information, and the head of the object free list, the object db# which will be used for the next object that's @created. If it's #-1, the free list is empty (you have 0 garbage) and new objects will not be using recycled space.

Keep an eye on max res mem, to get a feel for how it grows as your MUSH does.

Player purges

After a while, you may want to remove old players and their objects from your game. I typically use tinyfugue macros, rather than MUSHcode to do this. First, I define a macro /nuke <player>, which nukes a player and everything they own, providing they're not currently connected:

```
/def nuke = @switch hasflag(pmatch(%1),connected)=1,{think %1 is
connected!}, {@dol lsearch(%1,none,none)=@nuke ##}
```

The connected check prevents you from accidentally nuking yourself.

Then, I define a player purge macro like this:

```
/def ppurge = @dol lsearch(all,type,player)=@switch
[and(not(orflags(##,Wr)),lt(convtime(xget(##,last)),sub(convtime(time()),
2419200)))]=1, {think /nuke [name(##)] ## [lstats(##)] }
```

This macro searches the db for all players and lists those who aren't admin and whose last connection was longer than 28 days ago.

Actually doing a purge, then, requires these steps:

1. /log purge, to start logging output to a file called 'purge'
2. /ppurge, to list players who are candidates for purging
3. /log off, to stop logging
4. /sh vi purge, to edit the purge file and see who'll be purged. Delete lines in the file for players who you don't want to see purged. What's left will be a list of /nuke lines which will purge the players you want purged.
5. /quote 'purge, to load the commands in the purge file and do the actual purging.

6. /sh rm purge, to remove the purge file

Useful globals

The motd-maintainer

It's really convenient to be able to set the MUSH motd, wizmotd, and nologinmotd from within the MUSH. But when you @shutdown, those messages are lost. This code provides a way to preserve the motds across shutdown. It works like this:

Instead of directly setting the motd's with @motd, Wizards should set the motd's as attributes on this object (&MOTD, &WIZMOTD, &NOLOGINMOTD), and then type 'setmotd', which will post them via @motd. If the game is shut down, the attribs will continue to store the motd's, and on restart, the motd's will be reposted via the object's @startup code. This code actually builds the motd out of any attribs starting with MOTD, so you can maintain a set of motd's more easily.

Here's the important code:

```
@create MOTD Device
@link MOTD Device = #2
@lock/use MOTD Device = iswizard/1
@set MOTD Device = STARTUP
@set MOTD Device = !NO_COMMAND
@set MOTD Device = SAFE
@set MOTD Device = WIZARD
&DO_SETMOTD MOTD Device=$setmotd:@tr me/U_SETMOTD=%#
&U_SETMOTD MOTD Device=@motd [iter(sort(lattr(me/MOTD*)),%r[eval(me,##)])];@wizmotd [eval(me,WIZMOTD)];@pemit %0=
@STARTUP MOTD Device=@tr me/U_SETMOTD
&ISWIZARD MOTD Device=[hasflag(%#,W)]
@DESCRIBE MOTD Device=Motd is stored in attributes of the form MOTD*. Type 'setmotd' to set the motd.
&MOTD-1 MOTD Device=First motd message
&MOTD-2 MOTD Device=Second motd message
&WIZMOTD MOTD Device=Admin motd message
@tel MOTD Device = #2
```

The MUSHclock (autoshutdown, announcements)

Sometimes, it's useful to have a MUSH object which performs a command at regular intervals (for example, every hour on the hour). This object is a MUSHclock which, upon startup, figures out how long to wait until the hour starts, and then performs its function, @wait's an hour, and repeats. In this example, the only function it performs to to make an announcement if there's one set in the ANNOUNCEMENT attrib:

```
@create MUSHclock
@set MUSHclock = STARTUP
@set Announce = SAFE
@power MUSHclock = announce
&ANNOUNCEMENT MUSHclock = Your messages here
&ANNOUNCE MUSHclock=@switch [eval(me,announcement)]=,,{@wall [eval(me,announcement)]}
&LOOP MUSHclock = @tr me/announce; @wait 3600=@tr me/loop
@STARTUP MUSHclock=@wait [sub(3600,mod(secs(),3600))]=@tr me/loop
```

While this kind of code has many uses (weather and ecology systems often work on similar loops), two I've seen are (1) hourly announcements via @wall, and (2) causing the MUSH to automatically @shutdown at a specified time (9 am). The latter code is combined with an automatic restart script on the machine that restarts the MUSH each day at 5 pm, allowing a MUSH to operate only during non-prime hours.

Miscellany

The port announcer (announce.c)

MUSH is distributed with the source code for a port announcer, a small program which listens on a port for a connection, spits out a text file, and closes the connection. A useful thing when your MUSH is down indefinitely and you want to keep people posted on its status.

The distributed announce.c code, unfortunately, has a number of problems which keep it from running correctly in many cases, and may also cause it to crash. I use an alternative port announcer which works just great, called portmsg.c. Read the Appendix on Scripts and Programs for information about how to get portmsg.c.

When you've got it, compile the program with: cc (or gcc) -o announce announce.c

Run it with: portmsg textfile port#

You'll have to kill it by hand when you're done with it.

Hacking PennMUSH

Contents:

When to hack

What? There's a command that your MUSH really needs, a function not in standard PennMUSH, a bug you've discovered? In cases like this, the fact that you can "Use the Source, Luke" can greatly enhance your enjoyment of MUSH, your understanding of MUSH, and your MUSH. :)

I should probably insert here the standard cautions about not adding loads of feeping creatures ("feeps") and kludgy hacks to your MUSH, but I won't. Feeps are fun and usually harmless, and as long as you're happy with your code, don't let anyone tell you otherwise.

While there are no rules, then, there are a few guidelines I try to follow when deciding what *not* to hack:

- Avoid hacks which permanently alter the db or maildb structure, unless you know that you'll always be willing to support the code and cope with possible future PennMUSH upgrades, or if you don't care about ever upgrading. The sort of thing I'm talking about here is adding a couple new variables to the db structure. If you think your hack is of general interest, let me know and if I agree, you'll be assigned an official db-flag for your hack, and we'll arrange for built-in db conversion code to turn your changes on or off (the way USE_WARNINGS works, for example).
- Avoid hacks which require the MUSH to filter all outgoing text. EXTENDED_ANSI (and its relative ANSI_COLOR) is bad enough. While I've seen some beautiful commands like @wrap (which word-wraps output to players), each time you add a filter to all the MUSH's output, you significantly increase CPU requirements. Leave client-functions to clients, if possible.
- Avoid hacks with very limited functionality. If you've got spare time to hack, try to rewrite the chat system to improve its efficiency and give us more than 32 channels, or something.
- If the hack is something that seems very useful, email the idea and/or the source code (a context diff) to dunemush@mellers1.psych.berkeley.edu, the current PennMUSH maintainer (me). It may well appear in the next release!

Source code control, patches, and #ifdef

Source code control

If you hack at the PennMUSH source code, you will eventually make a mistake, and want to go back to an earlier version of your work. Or if you make a really good change, you may want to distribute it to others. For this reason, you should always use some form of "source code control" or "revision management" when hacking source. There are 3 common revision management systems:

1. *Backups*. Make a directory under your source directory called "oldsrc" or whatever, and put a copy of your source code into it. When you make changes, you can recover your old files from oldsrc, use it to produce patches, and eventually copy your new files into oldsrc when you're sure they work. Many people keep a "clean" source directory containing the original stock pl10 code, in case they need it. Of course, if you need to go back more than one revision, you're in trouble unless you clutter your disk

with many many oldsrc directories. A variant of this strategy involves storing older versions as compressed tar files.
2. *SCCS*. SCCS (source code control system) is a more sophisticated way to manage source code. It stores changes from version to version in a subdirectory. You "check out" files to work on them, and "check in" files that you've hacked. You can revert to any revision at any time. This is good. Many major unix systems (Ultrix, SunOS, HP-UX) come with sccs installed. Read the man pages for info.
3. *RCS*. RCS (revision control system) is the GNU project's free replacement for SCCS, available from ftp.gnu.ai.mit.edu in /pub/gnu. The commands are different from SCCS, and some things are easier to do. RCS is standard with Linux. There's a front-end program by GNU called CVS, too. RCS can also be used to ease upgrading to a new patchlevel by preserving your hacks to the older patchlevel (details below).

If you choose to use SCCS or RCS (and I can't recommend it highly enough), discipline yourself to *always* check in code after each revision, so that you can undo each step. I personally use RCS, and check in my files with "ci -l <filename>" (which checks the file in and then back out for further editing) since I'm the only person who modifies my code. If you have multiple people hacking (especially from different accounts on the machine), take advantage of the fact that RCS and SCCS will "lock" revisions so that only the person who checked it out can modify it and check it back in, preventing two people from making inconsistent changes.

When I start out with a new pennmush distribution, the first thing I do is to check in the entire source directory to RCS so I'll have a clean copy of the original distribution. I usually do it like this:

```
cp options.h.dist options.h
cp dune.h.dist dune.h
echo . | ci -n dist -l src/*.c hdrs/*.h
```

Applying patches with patch

Changes and bugfixes to the MUSH code are often distributed as "patches" or "context diffs", which are files which describe how the source code should be changed. The program "patch" (by Larry Wall, distributed by the GNU project, see above for ftp site) automatically reads these files and makes the changes to your source code. If you don't have the patch program, ask your system administrator to get it!

Typically, if you receive a patch from a mailing list or newsgroup, you simply save it to a file, and, from within your source directory, type:

```
patch < patchfile
```

in order to process the patch.

When patch fails - reading context diffs by hand

Here's a guide to reading diffs (and rejected parts of patches, commonly files ending in .rej after you try to apply a patch.) It's based on an email message by T. Alexander Popiel.

- If the diff begins with the line "Prereq: <somestring>", then it means that in the file that follows, the string should be present within the first few lines, or the diff should not be applied. Patch checks for prerequisites to help insure that you're patching the right version of the source code.
- For each file in the diff, there are two header lines, which look like this:
  *** filename1 date1
  --- filename2 date2
  This indicates that the diff is the difference between filename1 and filename2, and should usually be applied to filename2.
- After the header lines, the diff will indicate which line numbers in filename1 (the source) are to be examined, what should be changed or deleted, what the resulting line numbers in filename2 (the destination) are, and what should be changed or added:
  - A '-' in the first column of the source part of the patch indicates a line deletion.
  - A '+' in the first column of the destination part of the patch indicates a line addition.
  - A set of '!'s in the first column in both the source and the destination indicate a line-group replacement; a group of consecutive lines in the source are replaced with a corresponding group of lines in the destination.

Making patches with diff -c or rcsdiff

But what if you want to make a patchfile of changes you've made, to give to someone else? The "diff" program (a standard unix utility) can produce the patchfile, given the original and revised source code files. For example, if you revise player.c, and save the older version as player.c.orig, you could make a patchfile like this:

```
diff -c player.c.orig player.c > patchfile
```

The "-c" switch indicates that you want a context diff, which is more detailed than an ordinary diff and better for patches. If you're going to publicly distribute the patch, be sure it's a context diff! The order of the files is important: diff -c originalfile newfile.

If there's more than one source file changed, you can do this:

```
diff -c player.c.orig player.c > patchfile
diff -c game.c.orig game.c >> patchfile
```

If you use RCS, there's an even easier way to make patchfiles. Before you check in your revisions, type (for example):

```
rcsdiff -c player.c > patchfile
rcsdiff -c game.c >> patchfile
```

Rcsdiff makes a diff from the last checked-in revision to the current version of the file. If you read the man page, you'll see that it can also make diffs between checked-in revisions.

#ifdef and #define

You can save yourself a lot of hassle if you're careful in how you hack the PennMUSH code. When you decide to add new code, or change old code, add an #define into options.h which will turn your code change on or off. For example, if you're adding a new flag called NOMAIL, put something like this into options.h:

```
/* If defined, the NOMAIL flag (a toggle on players) will be in the game,
 * which prevents players from using @mail commands.
 */
#define NOMAIL_FLAG
```

Then, surround your additions with #ifdef NOMAIL_FLAG...#endif pairs. For changes, use #ifdef NOMAIL_FLAG...#else...#endif. This allows you to preserve the original PennMUSH coding, should you ever need to refer back to it (if, for example, you're trying to apply someone else's patch to something you've already changed), and allows you to turn on and off your feature as necessary.

How-to

This sections contains information on how to do the most common kinds of hacks that people want added to their MUSH: new flags, new powers, new attributes, new chat channels, new functions, and new commands.

Add flags

Unbeknownst to most players and admin, the MUSH actually has 5 different sets of flags. Technically, there is 1 set of true generic flags, which can be set on any type of object in the database (thing, player, exit, room). For example, the OPAQUE flag can be set on any object, though it's not meaningful for rooms. The same is true for CHOWN_OK, which is not meaningful for players.

The other 4 sets of "flags" are called toggles, and each set can only be used with one of the types of object.

There are a limited number of flags/toggles which can be created in each set, because internally, flags are represented as bits in a 32-bit integer. Thirty-two bits means 32 possible flags in each set. Flag space is scarce for generic flags, since most of the MUSH flags are of this type. There's plenty of room to add new toggles, however, and in general, if your new flag idea only applies to a single kind of object, you should add it as a toggle.

As an example, we'll add a (silly) toggle for players called "NOMAIL" (letter 'i') which prevents them from using the @mail commands.

Adding a flag is a 3-step process. First, you must edit flags.h and #define a name and bit pattern for the flags. You should be able to follow the way it's already done in the file. One hint is to try to use bit patterns from the highest possible (0x80000000) and work downward, since new flags added by Amberyl work their way up, and upgrading to a future patchlevel will be more difficult if your flags overlap new ones. If you're adding a toggle, you don't need to start all that high, since it's unlikely that large numbers of toggles will be added to future releases, but better safe than sorry.

In our example, we add these lines:

```
#ifdef NOMAIL_FLAG
#define PLAYER_NOMAIL   0x80000000
#endif
```

Next you must edit flags.c, and find the flag table. Usually, I just search for a nearby toggle, like PLAYER_TERSE. The flag table is where you list the name of the flag as it appears when you examine someone, the one-character abbreviation, its definition, and who can set it. Our added lines to the table looks like this:

```
#ifdef NOMAIL_FLAG
    { "NOMAIL", 'i', TYPE_PLAYER, PLAYER_NOMAIL, F_WIZARD },
#endif
```

That means that the flag will appear as "NOMAIL", abbreviated 'i', is a toggle on players (TYPE_PLAYER - a generic flag is NOTYPE), is defined by the PLAYER_NOMAIL #define, and can only be set by wizards.

NOTE: In post-pl10 releases, flags have separate permissions for who can turn them on and off. This flag would be written like this in those releases:

```
#ifdef NOMAIL_FLAG
    { "NOMAIL", 'i', TYPE_PLAYER, PLAYER_NOMAIL, F_WIZARD, F_WIZARD },
#endif
```

The valid flag-setting values are:

F_ANY
: anybody can set this on something they control

F_OWNED
: you must own the object to set this

F_INHERIT
: only inherit objects can set this on things they control

F_ROYAL
: royalty/wizards can set this on things they control - note that royalty typically don't control anything more than normal players!

F_WIZARD
: wizards can set this on things they control, which is basically everything

F_GOD
: only God (#1) can set this

F_INTERNAL
: nobody can set this - it's used internally by the game - used for flags like MARKED

F_DARK (post-pl10)
: Only God can see if this flag is set or not.

F_MDARK (post-pl10)
: Only admin can see if this flag is set or not.

F_ODARK (post-pl10)
: Only admin and the owner of the object can see if this flag is set or not.

If you want to require that only inherit wizards can set a flag, you can use F_INHERIT | F_WIZARD.

While you're in flags.c, also edit the flag alias table further down, which indicates what partial names and aliases can be used to refer to a flag when @set'ing it. You might add entries that look like this:

```
#ifdef NOMAIL_FLAG
    { "NO_MAIL", "NOMAIL" },
    { "NOMAI",   "NOMAIL" },
    { "NOMA",    "NOMAIL" },
    { "NOM",     "NOMAIL" },
#endif
```

Basically, add all the unique abbreviations of the flag's name.

Finally, you need to figure out what (if anything) your flag is supposed to affect in the code, and add in code that checks for the presence of the flag. For example, we could check in game.c when someone tries to use an @mail command. A common way to handle this is to edit dbdefs.h and define a macro like:

```
#ifdef NOMAIL_FLAG
#define NoMail(x)       (IS(x,TYPE_PLAYER,PLAYER_NOMAIL))
#endif
```

And then in game.c, you can deny the player the @mail commands if NoMail(player).

When you're defining macros for toggles, define them using the IS() macro above. When you're defining macros for generic flags, you can define them as (Flags(x) & FLAGNAME). For example, the definition of Visual(x) is:

```
#define Visual(x)       (Flags(x) & VISUAL)
```

That's it! Recompile and you've got a new flag.

Add powers

Adding powers is much like adding flags, but involves the flags.h, wiz.c, and look.c files.

In flags.h, you can define the power's bit pattern, just as you would with a flag (the powers are at the end of the file.) The same guidelines apply, except that powers are always generic and can always only be granted by Wizards.

In look.c, you must define how the power is displayed when someone examines the player, in the function powers_description. Find it, and the addition you'll need to make is obvious.

In wiz.c, you'll need to modify the function find_power, which looks up a power's bit pattern, given its name. Again, it's obvious.

And, as with flags, you can now define a macro so you can test for your power. For powers, however, this is done in db.h. For example, the login power's macro looks like this:

```
#define Can_Login(x)    (Hasprivs(x) || (Powers(x) & LOGIN_ANYTIME))
```

which is to say that the ability to log in goes with either being a royalty/wizard (which is what Hasprivs() tests for) or having the LOGIN_ANYTIME power. The Wizard() macro tests for Wizardhood and God() for Godhood, in case you need 'em.

Add chat channels

Adding new chat channels is easy. While the @channel/add command can add channels on-the-fly, those channels don't remain after a shutdown. To add a channel permamently, modify chat.c instead.

Edit the file, and go to the last function. You'll see a list of all the currently defined chat channels. Just follow along and add yours. For example:

```
   add_channel("Theme", CHP_PUBLIC, 0, 0x100000);
```

The first argument is the channel's name. The second is its privilege level. The third is 0 for a noisy channel (which announces connections and disconnections) and 1 for a quiet channel, and the last is the channel's bit pattern, which should be unique among channels.

The allowable privileges are:

CHP_PUBLIC
: Public channel which anyone may join

CHP_ADMIN
: Only admin (royalty/wizards) may use this channel

CHP_WIZARD
: Only wizards may use this channel

CHP_FORBID
: No one may use this channel - a flag used internally when you delete a channel.

CHP_OBJECT (post-pl10)
: If you've got OBJECT_CHAT turned on, only non-player objects and wizards may join this channel. Can be used for radios and the like.

If you plan to remove a chat channel, first be sure that no player is connected to it. While on your MUSH, use @channel/delete to delete it (which takes all players off it), and then shutdown, edit chat.c, recompile, and restart (or, if you're like me, do the edit and recompile while the MUSH is up, then shutdown and restart.)

Add attributes

Adding a new attribute is similar to working with flags and powers. You'll edit the attribute tables in the files atr_tab.h and atr_tab.c, and optionally modify player.c.

In atr_tab.h, you'll find the attribute table. Entries look like this:

```
  { (char *) "ACLONE",  AF_ODARK | AF_NOPROG,           NULL,   0 },
```

The entries in the table include the name of the attribute (casted to char *), its attribute flags (multiple flags separated with the | operator), its value (NULL in the table), and its owner (0 in the table).

The allowable attribute flags are defined in attrib.h to be:

AF_ODARK
: Only the owner (of the object or attribute) can see the attribute. If you *don't* include this attribute flag, the attribute will be a public one, like SEX and DESCRIBE.

AF_MDARK
: Only a Wizard can see the attribute, it's invisible to mortals.

AF_DARK
: No one can see the attribute, ever. Used mostly for internally used attributes like XYXXY (the player's encrypted password).

AF_LOCKED
: Only the creator of the attribute can change it. This is the flag set with @atrlock.

AF_WIZARD
: Only a Wizard can change the attribute.

AF_NOPROG
: The attribute won't be searched for $commands.

AF_NOCOPY
: The attribute won't be copied when the object is @clone'd.

AF_PRIVATE
: The attribute won't be inherited if the object is used as an @parent.

AF_NUKED
: The attribute is marked to be deleted. This is an internal flag that you never want to set on an attribute you're defining.

After making your entry in atr_tab.h, edit atr_tab.c and find the atr_alias_tab, the table of attribute aliases. Attribute aliases are just like flag aliases, added in this format:

```
        { "ACLON", "ACLONE" },
        { "ACLO",  "ACLONE" },
        { "ACL",   "ACL" },
```

To make use of your new attributes in your code, you need to know about the functions for manipulating attributes. They're all in attrib.c, but here's a sampler:

ATTR *atr_match(char *name)
: Given an uppercase attribute name, looks up a *standard* (i.e. defined in atr_tab.h or atr_tab.c) attribute and returns a pointer to the attribute as defined in atr_tab.h, with the flags, value, and owner as defined in atr_tab.h.

ATTR *atr_get(dbref obj, char *name)
: Given an object's db# (an integer) and an attribute name, return a pointer to that attribute on that object or its parent(s). If that attribute isn't set on the object or parent(s), the function returns NULL.

ATTR *atr_get_noparent(dbref obj, char *name)
: Just like atr_get, but doesn't check parents.

int atr_add(dbref obj, char *name, char *value, dbref enactor, int flags)
: This is the basic function for setting attributes. Given an object, an attribute name, a value for the attribute (or NULL to clear the attribute), the db# of the thing trying to do the setting (for control checks), and the attribute flags to set on the attribute, it tries to set the attribute and returns -1 if unable to, 1 if an attribute is set, and 0 if the attribute is cleared. If you want to be sure that the attribute is set, pass GOD as the enactor. If you don't want to set any attribute flags, pass NOTHING as the flags.

Sometimes you'd like to have every newly created player start with a certain attrib, the way all players start out with LASTSITE and LAST. You can do this by editing the function create_player() in player.c, and adding lines like these near where LASTSITE is set up:

```
#ifdef EXTENDED_MAIL
    (void) atr_add(player, "MAILCURF", "0", GOD,
                   AF_LOCKED|AF_NOPROG|AF_WIZARD|AF_ODARK);
#endif
```

This example is from the post-pl10 extended mailer, and adds the MAILCURF attribute (mail current folder) to the newly created player, set to "0", owned by GOD, and with the appropriate attribute flags.

Looking through the code for do_edit (and other code in set.c) can be very educational in learning to work with attributes.

Add locks

Editor's note: PennMUSH pl13 introduced a new system for managing locks, written by Ralph Melton (Rhyanna), which allows serverhacks to create any number of new locks. Here, Ralph describes how you create and use a new lock.

Adding lock types is more similar to adding a new flag or attribute than a new function or command. As an example, we'll add a Look lock that controls who can look at you.

First, edit lock.h to add a declaration of your new lock type. Go down to where it says '/* Declare new lock types here */'. Add a declaration for your new lock type like this:

```
extern const lock_type Look_Lock.
```

Then edit lock.c. At the point where it says, '/* define new lock types here */' Add a definition of the lock type, with the string you want to use to identify the lock to the user. This string is used in the output from examining the lock, and in '@lock/<string>'. For example, we'll add the definition

```
const lock_type Look_Lock = "Look";
```

Then, add the same string to the lock_types array just below, where the code says '/* Add new lock types */'

That's all you need to do to define a new type of lock. @lock, examine, and so forth will all work correctly with the new lock. In most cases, though, you'll want to define the effects of the lock, too.

The function eval_lock() returns the value of a given lock for a given person. For example, to add our Look_Lock, we'll add the following code near the place where it tests for the 'OPAQUE' flag:

```
        if (!eval_lock(player, thing, Look_Lock)) {
          notify(player, "You can't look at that.");
        }
```

That's it. Recompile and you're done.

Editor's Note: Don't forget to surround code which relies on RALPH_LOCKS with appropriate #ifdef...#endif pairs.

Add functions

Adding new functions to your game can be done via the @function command, as discussed in the section on Using #1 in the MUSH management chapter of this guide. But for many functions, it's more efficient to hack them directly into the code. The first time you do this, it can be confusing, but once you get the hang of it, it's easy.

The function parser changed between PennMUSH 1.50 and PennMUSH 1.6.x. In PennMUSH 1.50, most functions were defined in eval.c, with the exception of some wizard-only functions which were defined as external functions (XFUNCTIONs) in wiz.c. In 1.6.x, there are no more XFUNCTIONS, and most functions are defined in the fun*.c files (funlist.c, funstr.c, etc.) and sometimes in other files. You should add your own functions to funlocal.c; this file won't be overwritten by future patches or releases, so it's safe to add whatever you like to it.

Defining the function

First, you must write the actual function itself into eval.c (1.50) or funlocal.c (1.6.x) Functions are typically defined as follows:

```
FUNCTION(fun_<functionname>)
{
  text of the function
}
```

Find a function similar to the type of the one you're defining (i.e., find fun_add if you're defining a math function, etc.) and insert your function code after it. If you're using 1.50, *don't* add it to the end of eval.c, since the function must be defined before it is declared in the function table.

The FUNCTION macro sets up some standard conventions for coding functions. Your function is expected to return its result in the string buff (which is passed as a pointer, along with (in 1.6.x) bp, a pointer to the pointer). Its arguments are passed in the array of strings args[], and the number of arguments is passed in nargs. Finally, some dbrefs are passed. In 1.50 these are: privs, the dbref of the object executing the command (aka %! in MUSHcode), and doer, the dbref of the object which enacted the command, the object which triggered privs (aka %# in MUSHcode). In 1.6.x, there are 3 dbrefs: executor is the dbref of the object executing the command (%!, like privs); enactor is the dbref of the enacting object (%#, like doer); and caller is the dbref of the last object to do a ufun call (aka %@ in MUSH code).

Function arguments in 1.50

If your function will take a fixed number of arguments (which you specify when you declare it, later), checking for the correct number of arguments is handled automatically. If your function uses a variable number of arguments, you can check the number of arguments with nargs. If your function can take an optional delimiter, there are some special macros and functions you should use to check the number of arguments you got, and work with the delimiter:

varargs_preamble("FUNCTIONNAME", <max # of args>);
: This macro will check to be sure that the function hasn't been called with too many arguments, and checks that there should be either 1 less than the maximum number of arguments (and no delimiter) or the maximum number, in which case the last argument is a single character delimiter. The delimiter is returned in the char variable sep.

mvarargs_preamble("FUNCTIONNAME", <min # of args>, <max # of args>);
: This macro is like varargs_preamble, but allows fewer arguments (down to min # of args). If the function is called with the maximum number of arguments, the last argument is a single character delimiter.

char *trim_space_sep(char *str, char sep)
: This function takes a string (usually one of your arguments), and removes leading and trailing spaces from it if the delimiter sep is a space. If it's not a space, trim_space_sep just returns str again.

char *split_token(char *str, char sep)
: This function returns the next part of the string up to the delimiter sep. It destructively modifies the string, removing the token and the separator from the string. Check out fun_first for examples of trim_space_sep() and split_token().

char *next_token(char *str, char sep)
: This function returns a pointer to the string after the current token and delimiter sep. The only function which uses this directly is fun_extract.

int list2arr(char *r[], int max, char *list, char sep)
: This function takes a string (list) and splits it up using delimiter sep, and puts the results into the array r[]. It only uses the first <max> elements in the list. It returns the number of elements in the array.

void arr2list(char *r[], int max, char *list, char sep)
: The reverse of list2arr, this function takes the first <max> elements of array r[] and packs them into the string <list>, separated by the character sep. Check out fun_shuffle for examples of list2arr() and arr2list().

Function arguments in 1.6.x

PennMUSH 1.6.x is more flexible about function arguments, and allows variable arguments to be declared in the function's declaration (discussed below). You handle functions with optional delimiters by using:

int delim_check(char *buff, char **bp, int nargs, char **args, int sep_arg, char *sep);
: This function sets sep to the separator character if one was given. You must tell it (in sep_arg) which argument is the optional separator; usually, the last allowable argument to the function. This function return 0 and notifies the player if there's a problem, so it's usually used like this:

```
if (!delim_check(buff, bp, nargs, args, 4, &sep))
  return;
```

The main body of the function

Some other useful support functions when you're writing your function include:

dbref match_thing(dbref privs, char *name)
: Use this function if one of the arguments is the name of an object, and you want to find the corresponding dbref of the object. The function will return the dbref, NOTHING (-1), or AMBIGUOUS (-2), depending on whether the player can find the object or not, so be sure you check the result before you use it! A typical application might be:

```
dbref obj = match_thing(privs, args[0]);
if (!GoodObject(obj)) {
  strcpy(buff, "#-1");
  return;
}
```

int controls(dbref privs, dbref obj)
: Returns 1 if privs is allowed to control obj. There are also macros for some special kinds of control in dbdefs.h (Can_Examine, Can_Write_Attr, etc.)

void parse_attrib(dbref player, char *str, dbref *thing, ATTR **attrib)

[1.6.x] Takes a string in the format <obj>lt;attr> or just <attr> and returns the dbref of the object specified and a pointer to the attribute. It destructively modifies str. It's usually used something like this:

```
dbref thing;
ATTR *attrib;
parse_attrib(executor,string,&thing,&attrib);
if (GoodObject(thing) && attrib &&
    Can_Read_Attr(executor,thing,attrib)) {
  /* Legit to use */
}
```

Return values in 1.50

When you're building up your return value in buff, use the safe_str and safe_chr functions, which ensure that the buffer isn't overflowed. This means you must define a pointer to use with safe_str or safe_chr, typically done like this:

```
char *bp;       /* Defined with your other variables */
bp = buff;      /* Point it at buff */
safe_str(<string-to-append>, buff, &bp);
safe_chr(<char-to-append>, buff, &bp);
*bp = '\0';     /* Don't forget this! */
```

See fun_ansi for a nice clean example of how this works. You don't have to do this if you're returning a short message, like an error:

```
strcpy(buff,"#-1 NO WAY, DUDE");
```

Return values in 1.6.x

Functions return their results in the buff string in PennMUSH 1.6.x. This string should always be created/added to using the safe_str and safe_chr functions, which operate differently than in in PennMUSH 1.50. Their arguments are the string or character to append, buff, and bp, which is now defined as char **bp, and is passed to your function, so don't redeclare it. You also no longer need to add the null to the end of the string yourself, if you're using bp. This suffices:

```
safe_str(<string-to-append>, buff, &bp);
```

Declaring the function's prototype (1.6.x)

For each FUNCTION declaration, there should be a FUNCTION_PROTO declaration in hdrs/functions.h. These are easy. The function prototype for FUNCTION(fun_myfun) is simply: this:

```
FUNCTION_PROTO(fun_myfun);   /* funlocal.c */
```

Declaring the function

Finally, you must add the function's name to the function table. In 1.50 this is further down in eval.c; in 1.6.x, it's in function.c. Here's the 1.50 format:

```
{ "ABS", fun_abs, 1, FN_REG },
{ "ADD", fun_add, -1, FN_REG },
{ "ITER", fun_iter, -1, FN_NOPARSE },
```

The entry lists the function's name in the game, the name under which it's defined in the source code, the number of arguments the function should expect, or -1 if it expects a variable number of arguments, and whether the arguments should be treated normally and parsed (FN_REG) or passed to the function with no evaluation (FN_NOPARSE).

The 1.6.x format is similar:

```
{ "ABS", fun_abs, 1, 1, FN_REG },
{ "ADD", fun_add, 2, INT_MAX, FN_REG },
{ "CAPSTR", fun_capstr, 1, -1, FN_REG},
{ "ITER", fun_iter, 2, 4, FN_NOPARSE },
```

The entry lists the function's name in the game, the name under which it's defined in the source code, the minimum number of arguments the function should expect, the maximum number of arguments the function should expect, and whether the arguments should be treated normally and parsed (FN_REG) or passed to the function with no evaluation (FN_NOPARSE). If there is no maximum number of arguments, use INT_MAX as the maximum. If the last argument of the function may contain commas, and you don't want them to cause an error for passing too many arguments to the function, make the functions maximum number of arguments a negative number, such that abs(maxargs) = maximum number of arguments. For examples, see fun_capstr or fun_pemit.

The best way to learn to write new functions is to read the standard ones and follow along. Copy the code from a function that's similar to the one you want to write and modify it (don't forget to change its name, and to declare it in the function table!)

Add commands

Adding new commands, like adding functions, is a 3-step process. The command must be added to the command parser in game.c, so that the command will be recognized by the game. You must decide in which file to locate your command code. Finally, you must write the code to implement the command.

As an example, we'll create the simplest possible new command, @mynum. The @mynum command will take no arguments and will simply tell the thing executing it what their db# is, as if they'd typed "think [num(me)]".

How commands are processed

Before going into the actual hackery, it's useful to take a look at what happens when the MUSH receives a command from a player or object. Here's what goes on in the function process_command() in game.c:

1. Various consistency checks
    1. There's really a non-null command
    2. The object executing the command has a valid db#
    3. The object isn't HALT or GOING
    4. The object is in a location with a valid db# that isn't GOING
2. If the object is SUSPECT, the command is logged
3. If the object is VERBOSE, its owner is shown the command
4. If the command is 'home', try to do it.
5. If the command is a force of the form "#100 <action>", do it.
6. If the command is a single-character command (", :, ;, +), do it.
7. If the command is an exit name, go there
8. If the command is to set an attribute (@attr obj= or &attr obj=), try to set it
9. Do a huge switch statement which scans the command letter by letter until it matches a unique MUSH command, or fails.
10. If the command's an enter alias, enter something.
11. If the command's a leave alias, leave.
12. See if it matches a user-defined $command here.
13. See if it matches exits in a parent or zone master room
14. See if it matches $commands in a zone master room or object
15. See if it matches $commands on a player's personal zone
16. See if it matches global (master room) exits
17. See if it matches global $commands
18. If not, return a Huh? message.

That huge switch statement is where new commands are typically inserted. Here's a section of it around the area where we'll put @mynum:

```
case 'o':
case 'O':
  if (gagged) break;
  Matched("@motd");
  if (!slashp)
    do_motd(player, 1, argu);
  else IfSwitch("connect")
    do_motd(player, 1, argu);
  else IfSwitch("list")
    do_motd(player, 3, "");
  else IfSwitch("wizard")
    do_motd(player, 2, argu);
  else IfSwitch("down")
    do_motd(player, 4, argu);
  else IfSwitch("full")
    do_motd(player, 5, argu);
  else
    goto bad;
  break;
default:
    goto bad;
}
break;
```

```
    case 'n':
    case 'N':
      /* @name, @newpassword */
      switch (command[2]) {
        case 'a':
        case 'A':
          if (gagged)
            break;
          Matched("@name");
          do_name(player, arg1, arg2);
          break;
...etc...
```

Declaring the command in game.c

The idea, then, is to add our new command between the case 'o' which is for @motd and the default case, which is for commands that start with @m but don't match any known command. We add something like this:

```
case 'y':
case 'Y':
  /* @mynum */
  Matched("@mynum");    /* Tells the game you've hit a match */
  do_mynum(player);     /* Calls your actual code */
  break;                /* Don't forget this! */
```

Command switches and arguments

The declaration of @motd above illustrates how to handle a command which allows /switches and which takes arguments. The slashp variable returns 1 if there is a / in the command, and IfSwitch("str") tests the part after the / to see if it matches a string.

A set of macros are defined in game.c to make dealing with arguments easier. The follow variables and macros are useful things to pass to command code:

player
: The db# of the object executing the command (%!).

cause
: The db# of the enactor (%#), the object which triggered the command.

argu
: The argument to the command, basically everything after the command itself, with pronoun substitution and function evaluation performed on it. Use this when you expect the command to have a single argument (like @wall).

arg1
: The argument to the command up to an '=', with pronoun substitution and function evaluation performed already. For example, in "@set me=terse", arg1 will be "me".

arg2
: The right-hand side of the '=', as above. Pronoun and function evaluation is done on this, too. In the example above, arg2 would be "terse".

arg
: Like arg2, the right-hand side of the '=', but not parsed (no substitution or evaluation.) Useful for commands like @edit.

buff3
: Like argu, the whole argument to the command, but not parsed. Used for commands like @doing.

argv
: An array of arguments to the command from the right-hand side of the '='. Arguments are separated by commas. Used for commands like @open and @verb.

vargs
: Like argv, an array of arguments to the command, but not parsed. Used for commands like @switch.

Where to locate your command code

It remains to define the do_mynum() function which game.c will call to run your command. Many files contain the code for commands, organized roughly by the function of the command. Here's a list of files with a description of the sort of commands in them, and an example:

attrib.c
: Commands affecting attributes (do_atrlock)

bsd.c

Commands requiring access to the list of connected player descriptors (do_doing, which does WHO/DOING)

chat.c
: Chat system commands (do_chat)

cque.c
: Commands affecting the queue (do_wait)

create.c
: Commands related to object creation (do_create)

destroy.c
: Commands related to object destruction (do_destroy)

eval.c
: Commands related to functions (do_function, which does @function)

game.c
: Commands related to the game loop (do_shutdown)

log.c
: Commands for logging (do_log)

look.c
: Commands for looking and examining (do_look_at)

mail.c
: Commands for @mail (do_mail)

move.c
: Commands for motion (do_enter)

player.c
: Commands related to player objects (do_password)

predicates.c
: Commands related to testing things (do_switch)

rob.c
: Commands related to transferring MUSH coins (do_give)

set.c
: Commands which affect db objects (do_lock)

speech.c
: Commands which send messages from player to player (do_page)

wiz.c
: Commands which implement special powers (do_teleport)

What about @mynum? Well, it's an informational command about a db object (yourself), so the best match is probably in look.c, along with do_examine, and do_whereis, and the like.

Since it's called from game.c, however, you must put a function declaration in game.h, using the K&R C style, typically. That is, a line like:

```
extern void do_mynum();
```

Writing the command

Finally, you can write the command itself. In our example, we might edit look.c and add a function like this:

```
void do_mynum(player)
    dbref player;
{
 notify(player,tprintf("#%d",player));
  return;
}
```

The notify(dbref,string) function sends a message to an object. The tprintf() function is like printf() but just returns what printf would have printed as a string, and so is used widely with notify().

Examine more complex commands for examples of control checking, locating objects in the database, etc. Often you'll find that other commands in the same file will be similar and can be looked at for ideas.

Upgrading to a new patchlevel

When you've done substantial hacking on your current patchlevel of MUSH and a new patchlevel is released, you must decide if you want to upgrade to the new patchlevel. While you can certainly continue running your current code if you are satisfied with it, I

generally suggest that you take the time to upgrade. New patchlevels often bring valuable bugfixes and efficiency improvements, as well as new user features.

Having decided to upgrade, you must then figure out how to migrate your custom hacking over to the new patchlevel. While it is certainly possible to scan your files for your custom code (which is all contained in #ifdef's, right?) and manually update the new patchlevel, the process can often be automated through use of the diff and patch programs, or through the rcsmerge program if you use RCS.

The basic idea is to try to merge your changes with the changes in the new patchlevel wherever possible, and to note where the two are incompatible so you can manually decide what to do. There are three basic approaches, each of which requires that you have on-hand a copy of the stock distribution at your patchlevel (let's say it's pl9, for example), the stock distribution at the patchlevel you want to upgrade to (pl10, for example), and your hacked source code.

1. You can use diff -c to make a patch file which would upgrade the stock pl9 code to stock pl10, and use patch to apply it to your hacked pl9 code. If you do this, be sure you copy your working code, and do the patching on the copy!
2. You can use diff -c to make a patch file which would upgrade the stock pl9 code to your hacked pl9 code, and use patch to apply it to the stock pl10 code.
3. You can make an RCS directory under the stock pl9 code, and check everything in. Then make symlinks to this directory under your hacked code and the pl10 code and check all of those sources in under different revision names (ci -npl10 *.c *.h, for example). Then you check out the stock pl9 code again, and use rcsmerge to merge the changes from hacked pl9 and from pl10 into the stock pl9 code (rcsmerge -rpl9hack -rpl10 *.c *.h). You will be warned when there is overlapping code in the two new versions, and you'll have to edit the resulting file and select which version you want (both will be written to the file).

I have had success using all of these methods, though my preference is usually for the second, since I like to start with the new patchlevel and try to have my hacks added to it. I make a set of diffs using

```
foreach file (*.c *.h)
diff -c pl9/$file pl10/$file > diffs/$file.d
end
```

and typically glance through each diff before patching it into the new code. If parts of the patch fail, I examine the resulting .rej file and manually incorporate (or not) my hack before going on to the next diff. I can usually upgrade in about 90 minutes.

None of these methods will be perfect, and you'll still have to hand-edit a number of files. The merge method lets you make your choices while editing the files. The diff methods will create .rej files when parts of patches are rejected, which you'll have to examine one by one. Be especially careful if you've made hacks which relied on features of the old patchlevel which are no longer present in the new patchlevel - you won't want to copy those over.

The PennMUSH code style

From the time of PennMUSH pl13, I've been trying to take pains to make the PennMUSH code as portable and standardized as possible to make porting, diffing, and other such things easier.

To support this, I (and other contributing Pennhacks) have adopted some coding conventions, which are listed below. If you use these conventions, your hacks are more likely to work on multiple systems, and will be easier for me to integrate into new patchlevels.

- All source files should #include "config.h" *before* any other include file (except perhaps copyright.h) and should #include "confmagic.h" *after* all other include files. This lets the file take full advantage of the autoconfiguration script.
- Use *memcpy* in preference to bcopy
- Use *strchr* in preference to index
- Signal handlers return *Signal_t*, defined through autoconfiguration. If SIGNALS_KEPT is defined, you don't have to reset signals in the handler.
- malloc returns *Malloc_t*, and calls to free should have their parameters cast as (Malloc_t). I.e.: free((Malloc_t)buff);
- Use *mush_malloc(size to malloc,"name of mem_check")* and *mush_free((Malloc_t) ptr to free, "name of mem_check")* when possible. These are macros which are preprocessed to plain old malloc/free when MEM_CHECK is not defined, and which call a function in utils.c to add/delete a mem_check before doing the malloc/free when MEM_CHECK is defined. You still need to add MEM_CHECK's manually if you get your memory by strdup or safe_uncompress or something.
- Use the *UPCASE()* and *DOWNCASE()* macro to get the uppercase or lowercase versions of a character. The autoconfig checks whether toupper() can accept only lower-case letters, and defines UPCASE to protect toupper. If toupper() is safe, UPCASE() is defined as toupper() to be more efficient.
- All functions should be explicitly prototyped (in the source file and in externs.h if appropriate) by using the *_()* macro. This will enable the prototype to be used by systems which can use it, and ignored by those which can't. For example: void myfunc _((void));

- Function definitions should be written in the older K&R style, in general. There are two exceptions:
  1. Functions which require a variable number of arguments must be written both ways, using the I_STDARG symbol (*NOT* __STDC__) to control which definition is used. See the code for tprintf() as an example.
  2. Functions which take char or float variables must be defined using both new-style and K&R-style definitions, using the CAN_NEWSTYLE symbol to control which is used. See atr_comm_match() as an example. This is necessary because an ANSI compiler will see the prototype and expect a char value, but based on the K&R definition will "widen" the value and pass an int value. Floats get widened to doubles. This could be bad.
- If a module needs string[s].h, use:

```
#ifdef I_STRING
#include <string.h>
#else
#include <strings.h>
#endif
```

- If a module needs varargs, use:

```
#ifdef I_STDARG
#include <stdarg.h>
#else
#include <varargs.h>
#endif
```

- The code now follows a standard indentation scheme, which is documented in src/Makefile under the 'indent' target. It requires GNU indent 1.9 or later, and once you set the path to GNU indent in src/Makefile, you can use 'make indent' to reindent your code. Please re-indent before making context diffs for patches.

Miscellany

Uncompressing attribute text

When you're writing code that gets attributes from objects and then works with the attribute text, you'll find that the text (stored in attribpointer->value) is in a compressed format designed to save memory, and you'll need to uncompress it if you want to read it or modify it.

There are two different uncompression functions, and you should be careful which you choose and how you use it. The *uncompress* function stores the resulting text in a static buffer which it returns to the calling function. Since this buffer changes with each call, you want to copy the results of calls to uncompress, using code like:

```
char tbuf1[BUFFER_LEN];
strcpy(tbuf1,uncompress(a->value));
```

The *safe_uncompress* function, on the other hand, mallocs memory to store the uncompressed text, and returns a pointer to that memory. If you then use strcpy, you will effective have mallocs memory to store the uncompressed text, and returns a pointer to that memory. If you then use strcpy, you will effective have two copies of the text, one of which be lost (you won't have anything pointing to it), a situation called a "memory leak", since every time your code executes, memory will be getting allocated and never freed. This is bad. The correct way to use safe_uncompress is:

```
char *buff;
buff = safe_uncompress(a->value);
/* Do stuff with buff */
free(buff);     /* crucial to free the allocated memory */
```

Using a tags file

If you use vi as your editor, you can make your life a lot easier by using the 'ctags' program to create a "tags" file, which is a file your editor can use to immediately take you to the file and location where a given function is defined.

You can create a tags file by going to your MUSH source directory and typing 'ctags *.c'. Ignore any warning messages.

To use the tags file with vi, use the command ':tag functionname' (for example, ':tag do_whereis') and vi will load up the file containing the function functionname(), and put you at its definition.

Ralph Melton (Rhyanna@Castle D'Image) provided instructions for using a tags file with emacs. First, create the file by going to your source directory and typing 'etags *.c *.h'.

To use the tags file with emacs, use the 'M-.' command. For example, 'M-. do_whereis' will take you to the definition of the do_whereis() function.

Running a Successful Game: Tips from Gods

This section of the manual is devoted to tips and ideas about how to make a MUSH successful. Because I'm far from the authority on the subject, the section is made up of contributions from other MUSH Gods (including mine, of course :) who've been good enough to submit them.

If you have tips you'd like to see added to this page, you can mail them to Javelin. Please be sure to include your email address and your character name on the MUSH you run (as well as the name of the MUSH itself!). Including your RL name is encouraged.

Contents

Contributors

The tips are now organized by subject rather than author, but here's a list of current contributors and a few notes about each:

Amberyl (Lydia Leong, lwl@digex.net)
: These thoughts come from Amberyl (Lydia Leong, lwl@digex.net), who's been a Wizard at too many MUSHes for me to relate, including PernMUSH and AmberMUSH. I first met her as Polgara, the God of BelgariadMUSH. She was the original maintainer

of the PennMUSH code, and is the author of the MUSH Manual, required reading for Gods.

Fenring@DuneII
: These thoughts come from Fenring@DuneII, who, while not yet God of a MUSH, is nevertheless a fine Wizard who I first got to know at DuneMUSH. He is also a consummate role-player.

Gohs@GohsMUSH (Geoff Tuffli, tuff@midway.uchicago.edu)
: These thoughts come from Gohs@GohsMUSH (Geoff Tuffli, tuff@midway.uchicago.edu), who has also been the God of the MUSHes TaiesMUSH, Mua'kaar, and Pandemonium. Geoff's MUSHes are known for their rich original-theme worlds, emphasis on dynamic role-playing, and complex coded systems. I've had the pleasure of knowing him since we were both admin at Belgariad MUSH.

Javelin, Paul@DuneMUSH (Alan Schwartz, dunemush@mellers1.psych.berkeley.edu)
: Usually this would be an introduction and a little note about the God offering the tips, but that seems a bit self-serving. :)

Rhyanna@Castle D'Image (Ralph Melton, ralph@cs.cmu.edu)
: These tips come from Rhyanna@Castle D'Image (Ralph Melton, ralph@cs.cmu.edu). Ralph is a top-flight hacker who often contributes significant additions and fixes to the PennMUSH code.

Talek (T. Alexander Popiel, popiel@colorado.edu)
: These thought-provoking insights were written by Talek (T. Alexander Popiel, popiel@colorado.edu), who, while also never actually the God of a public MUSH, has been intimately involved with Belgariad MUSH, DuneMUSH, and many others. He's well-known as both a psychocoder and a server hack for both PennMUSH and TinyMUSH.

Westley@PrincessBrideMUSH, Korba@DuneMUSH (Christofer Hardy, chardy@jackalope.lcc.whecn.edu)
: These thoughts come from Westley@PrincessBrideMUSH (Christofer Hardy, chardy@jackalope.lcc.whecn.edu). He's been deeply involved not only in PrincessBrideMUSH, but also in Heretics of Dune MUSH and DorsaiMUSH. I came to know him as Korba@DuneMUSH, a loyally fanatic Fremen. :)

Mephisto@MUSHtv (Jason Newquist, newquist@netcom.com)
: In addition to his work on MUSHtv, Jason Newquist headed up two versions of CamelotMUSH (as Mordred and Dagonet), and is always working on new MUSH projects.

Design tips

These tips cover issues related to starting or designing a MUSH or game world.

When NOT to Start a MUSH (Talek)

When you start a MUSH, you should have four things:

- An idea of what type of MUSH you want, be it social or RP, themed or anything-goes, whith lots of coded systems or just the bare minimum. If you don't know what you want the MUSH to be, then nobody else will either, and it will end up as something nobody likes.
- Far too much free time; running a MUSH can easily eat 20 hours (or more) a week.
- A group of people to help build and run the MUSH. These will form your initial admin team; they'll have plenty of influence over your MUSH, so make sure you know and trust them.
- A site to run the MUSH. A great site isn't necessary immediately, but a bad site can kill your MUSH before it really gets started.

If you don't have these four things, it might be a good idea to wait before starting your MUSH.

Tips for new Gods (Westley)

Get good admin, see how others act on other MUSH's before asking them to join (I had two rather Bad Wizards because I didn't do that.) Get people who want to make the mush better, and who love the genre, make sure of that before they become admin. I recruit new admin, by watching, and listening to the current admin, and if a player shows they are doing a lot of work, I start to talk to them, then pass it through the other admin. Then make them an admin, but the bit is temporary for 2 weeks, then it's permanent. I also have a head

Wizard, on both MUSH's (one of whom will become its God). They help take some of the pressures off.

Think out what you are going to do, it's a lot of responsiblity. Make decisions, and stand by them, but if a Player/Admin has a comment, or objection, listen to them with an open mind, after all, they are your biggest asset. Try not to get caught up in bureaucracy, and fights between players. It looks bad.

Have a good mood going into this, after all, if you don't, it will only get worse.

Listen to people who have done this before, and don't try to run 2 mush's at once :)

And remember, if you don't work to make this fun for the players and admin, and be fair, they will go and play in their own sandboxes.

Oh, and it's nice to have a listserver too :)

Theme (Gohs)

Most MUSHs have some sort of theme; in general, this is enforced fairly stringently. Players who come on must learn the theme and adhere to it. If the players are truly trying to learn and are genuinely willing to correct mistakes they may (and probably will) make, there is generally little problem here, but there will inevitably be those who either refuse to follow the theme or who will argue with the admin. It is a good idea to place somewhere in the news or policy that the word of the admin on matters of theme is final - this can dodge the problem of a player holding a book up in an admin's face and snidely insisting that the MUSH is all wrong. While this may be true, a MUSH almost has to operate on the basis of a specifically determined interpretation of the theme, even though this may mean that early errors are propogated and even maintained.

Of those MUSHs that are themed, virtually all are based off of a series of books or a movie or set of movies. In all cases you should secure permission for doing this - while it hasn't happened yet that I know of, it is entirely possible to be sued for copywrite infringement if you put up a MUSH based on an author's work without their publisher's permission; it's simply not worth the risk, and, moreover, is rude and inconsiderate to the author as well.

MUSHs whose theme are based off of a series of books or movie have the single great advantage over originally themed MUSHs that players can come onto the MUSH and have a good idea of the theme. Even if they haven't read the books or seen the movie or what have you, if they pick up interest on the MUSH, they can, and often will, borrow or beg or buy the works in question. In addition, a MUSH based off of an established work will often be able to attract an initial group of players who are interested in the series or movie and so are thus willing to give your MUSH a chance. Finally, MUSHs based off of an established work do not have to worry about working out the gritty details or worry about creating an integrated whole, something that can be difficult and is always time consuming to do.

Nonetheless, originally themed MUSHs do have their place. They require substantially more work to set up and require a combination of skill, luck and effort to make them work - and it is almost guaranteed that no matter how good a job you do, you will almost never be able to achieve the sheer numerical popularity of MUSHs with themes based on established works. Creating an originally themed MUSH has its payoffs, however. First and most importantly, creating an originally themed MUSH enables you to tailor-fit a theme to the constraints and capabilities of MUSH code. For example, if you have a fantasy themed MUSH, you can look at the various @powers and the various options that MUSH code allows and can create a magic system based on this - and a magic system like this will be a hundred times easier to implement than virtually any established work's. Secondly, while you will get people complaining about realism, so long as you maintain internal consistency you never have to worry about players attempting to prove your MUSH dreadfully, hopelessly wrong by waving a book in your face. Thirdly, it allows you to have a theme that you can all but guarantee is unique - and if you do it well, you will have something that no other MUSH will have; there will be no need to worry about a half-dozen other MUSHs based on, say, the Pern series or the White Wolf games. It can be enormously rewarding and challenging, but it requires an even greater investment of time and thought, and even more so than is the case for putting up a normal MUSH, it is not something that can be undertaken if you wish to have a serious chance at creating a successful and popular MUSH.

Scope and Geography (Gohs)

If you are creating a MUSH which has as a primary or at least secondary purpose that is roleplaying, it is a good idea to be watchful for design factors which will either assist your MUSH or harm it. One of the key factors to keep in mind is the player:room ratio. The more rooms you have in relation to the number of active players on at any particular time, the harder it will generally be for a given player to accidenly meet up with another player. Thus, it is almost always a good idea to keep the number of rooms down as much as is reasonably possible, as this enhances the opportunities for spontaneous roleplaying, something that can be a serious concern in attracting new players who do not know who to contact or do not themselves have an established group of people with whom they can regularly roleplay.

Another aspect of this is geography. If you wish to maintain a sense of realism, it can be problematic to have the MUSH spread out over a very large geographical area. A person logging in may not know everybody or even anybody who is on-line at that particular time. If they are in one part of the MUSH world, they either have the option of waiting for someone in their area to come on, or else bending realism and crossing uncounted miles or lightyears to go to where "the action" is. To some degree this can be modulated by deliberately twisting time. Dune did something like this with its shuttles. In reality, the shuttles would have taken far longer to venture between planets, but for the sake of the game, this time was shortened to an amount that the players would find acceptable.

Because of this, it is often a good idea to restrict the geographical confines of the MUSH to an area that could realistically be crossed by an in character person in a day. This would obviously vary depending on the technology, and there are ways around this, but establishing this can also have the additional side benefit of reining in unlimited growth, which can lead to database bloat and the problems associated with that. Similarly, if you have established factions it is a good idea to keep the absolute number of factions down to a reasonable number; presumably you wish for there to be several active players in each faction, and if there are a hundred factions, this is unlikely in the extreme.

Roleplaying and Regulations (Gohs)

There are two issues that I want to address here. First, there is the issue of in character and out of character play. A MUSH may be intended for virtually solely in character play, or it may be intended to supply an out of character medium for players to chat and amuse themselves in. If your intention is for in character play, to prevent problems arising from characters doing things or being places that they normally could not do in character and are claiming immunity because of being out of character, it may be a good idea to set aside an OOC or Out of Character Room where anything happening outside of it is automatically considered "in character".

The second issue arises from the fact that while MUSHs originally were mostly social places, they have slowly over the past few years changed into mostly roleplaying places. Two common ways of dealing with in character plots, or "tinyplots" present two very different ways of running a MUSH. The first is where there is a ruling to the effect that you must obtain someone's permission to involve them in a tinyplot. The second, on the other hand, holds that if you are on the MUSH and if you have created a thematic character, you are there to roleplay, and should log off or go to the OOC Room if you don't wish to roleplay. The first method works adequately if the MUSH is still somewhat social in nature, and also if the MUSH has relatively few coded systems (though this is not a hard and fast rule by any means), but it can run into problems where players who are there to roleplay get frustrated in their attempts to start plots or who are seeking spontaneity. Whichever method you choose to use, be aware that the players will adapt to it. Players used to a social environment will be far more comfortable on a MUSH requiring active permission, but many players will find this tedious, and requiring active permission to involve people in tinyplots will dampen the number and inter-relatedness of roleplayed plots on the MUSH.

Rule Enforcement (Talek)

There are two main styles to rule enforcement: you can make it impossible (well, difficult) to break the rules, or you can ask everybody to play fair and then punish the people who don't.

Making it impossible to break the rules also comes in two flavors: not having any rules to break, and having lots of code built to try to keep people in line. The former leads to a rather unpredictable environment (pure anarchy comes to mind), and the latter tends to attract code breakers who will find all of the security glitches in the system. I find neither of these options attractive.

Asking people to play fair is far from a perfect solution; inevitably, some people don't, and dealing with such people can be discouraging and annoying. However, most players will be cooperative, and I think the resulting community atmosphere is well worth the occasional setbacks.

Thoughts on Faction-based MU*'s (Fenring)

Introduction

An ongoing subproject of MU* development is experimentation with the faction-based environment. From my understanding of things, DuneMUSH was the first to really dive into this complicated area, and made extreme strides towards fleshing it out and developing means of regulating faction conflict. DuneMUSH-II has followed in its footsteps, and has attempted to build on the strides made by its predecessor. Most of this discussion will refer to these two mushes. I was a longtime player, and later a Judge and Roy at the original Dune, and was and am part of the team that conceived and built DuneMUSH2 from the ground up. Most of the serious thought I have given to MU*'s has been directed at solving the inherent problems with faction-based mushes, and so this submission will focus there as well.

Special thanks goes first and foremost to Javelin/Paul, who authored this Guide for Gods, who was creator and god of the first mush I played on, DuneMUSH (and who gave me that first ROYbit, which got me hooked but good :) and who enabled DuneMUSH II to come into being. Thanks also goes to ALL the admins at DuneMUSH II whom I have worked with to create a MUSH that we are all very proud of.

I. What is a faction-based mush?

The creators of any mush should think long and hard about the nature of their mush at the most basic level. I tend to lump mushes into two general categories: pure RP mushes, and faction-based mushes. Of course, labels never sit neatly, and there is a lot of in-between. But I will try to elaborate somewhat on this distinction before moving on. A 'pure RP' mush is one in which characters enter and RP with very little structure surrounding them. Personal allegiances may form, but one is not confronted with an immediate _choice_ of allegiance. The various vampire mushes fall in this category. Though it is my understanding that people do form secret groups, etc, generally, a player is a person in a modern-day setting, without clear lines of allegiance. The emphasis is on interpersonal RP, generally with players serving their own agendas to a large extent. In the middle of pure RP and faction-based mushes are places like Gohs, Pandemonium and Amber, where people often do have titles and/or factional associations. But at the same time, these factions are only part of what's going on; the emphasis still seems to remain to a large extent on interpersonal RP. Finally, there are the purely faction-based mushes. Dune and Dune2MUSH fall in this category, as do the Star Wars mushes. In these, the focus is on the struggle between clearly defined units, groups, or factions. Individual RP is important, but is almost always focussed on the advancement of the goals of the faction, rather than the individual.

II. Why is this distinction important?

There are two major reasons. The first is, you should consciously choose which of the two types best fit the theme you want to bring to life. This is very, very important. To go to a ludicrous extreme, if you wanted to bring to life the NFL, in FootballMUSH, then the choice would clearly be faction- based. On the other hand, if you wanted to have a mush set in a small town in North Carolina where everyone played colorful people like a sheriff, a barber, a little boy, or a schoolteacher, then you would be tending towards a pure RP structure. This initial decision should basically effect all future decisions that you will make in developing your mush.

Secondly, once you have chosen the 'right' type of mush, you should now be clued into what sort of rules systems you need to develop. The importance of this may perhaps best be shown by negative example. Assume that FootballMUSH is in the development stage. The admins want to have a mush that recreates the thrill of a football season, with weekly games, playoffs, even a super bowl. They go to the various ftp sites and port in all sorts of coded systems, included combat, places code, and various other traditional systems, the kind most mushes have. They opt for a judged-RP system, and have several judges ready to go. Game day rolls around, and the players take the field. Then it occurs to the admins: 'how do we determine who wins'? There are judges, but how can they fairly determine this? Will it be based on how many players showed up? The quality of RP? A coin toss? The point is, if your mush is based on faction conflict, you _must_ have means to resolve conflicts between those factions. Otherwise, you will frustrate the whole purpose of your mush. The first step to solving this problem is thus being aware from the start whether or not you have a faction-based mush, and then preparing for the inevitable clashes between those factions.

III. Why bother with a faction-based mush?

This is a very good question. Building and maintaining a faction mush is a _lot_ of work. It is arguably the most difficult type of mush to run successfully. The argument has been made that it isn't worth it, and some experienced admins have backed off the faction concept for that very reason.

The primary reason that one should opt for a faction-based mush is that, if done properly, it is extremely rewarding and fun. At its best, a faction-based mush gives a level of RP that is supercharged and _real_, in that skilled players can tangibly alter the universe around them. In a sense, the perfect faction mush (which hasn't yet been seen, of course), is the pinnacle of RP. The Gamemaster is not a single

individual choreographing events, but a rather a dynamic collective... your opponents are other players, scheming and carrying out plots in an attempt to triumph over you, even as you do the same to them. Pure RP mushes, on the other hand, tend to be fairly limited in their scope, and often scenes are choreographed, so that they might be considered 'acting' or 'co-authoring' rather than roleplaying. Now, RPing a good scene can be _extremely_ rewarding, but at a visceral level, the adreneline boost of teamwork and faction conflict has the potential, at least, to be more rip-roaring fun!

In sum, the net certainly has a place for all sorts of MU*'s. If Pure RP is your thing, so be it. But there will be those who always have an eye out for that new faction-based mush, to see if it has made any steps towards the ultimate, the perfect faction mush.

IV. The Difficulties of Arranging For Faction Conflict

As in developing any system to resolve disputes that occur in an imaginary context, the primary difficulty is developing a system that is both playable (easy to understand and use) and that is also realistic. Generally, the simpler in real life that the process you are attempting to approximate is, the better the rules to approximate it online will be, too. To continue with using ludicrous examples, consider the following. In FootballMUSH, an integral part of the game is the coin toss by the referee. So, the admins set out to make sure that their coin toss system is the best it can be! They code a global which uses a 50/50 probability, and whenever someone types '+toss', 'heads' or 'tails' is emitted to the room, along with a few poses showing that someone tossed the coin in the air, and it landed. This system is basically unassailable, because it is easy to use and approximates almost identically the actual tossing of a coin. The reason it is so easy is that the thing itself being imitated is so simple and basic.

Now, a very nettlesome and more difficult area is that of individual combat. This is an area most admins dread, because it is so highly charged, and virtually every player has their own opinion on it (and a good many of them tell you that opinion, whether you want to hear it or not). Finally, it is important because often the life or death of a beloved character hinges on the use of the system. The reason combat is so devilish is because of the fact that in RL, there are many, many variables, far more than can be realistically approximated online. The debate goes back and forth on this one: should combat be judged only; should it be coded; should it go round by round; should it be resolved by one roll; should it be purely cooperative; etc. Yet, individual combat is far less complicated than is arranging for faction conflict.

The reason for this is that on a large scale, there are many, many more variables than there are in a single combat. Considering an example from DuneMUSH II, the conflict between two Great Houses of the Landsraad, in a war of assassins. This conflict generally takes the form of a War of Assassins. The war can take place on many, many levels. There is straight military conflict; assassination attempts and covert operations; economic warfare; terror campaigns; political warfare in the Landsraad; political warfare in the Imperial Court; and a public relations war as well. A system that arranges for conflict such has this is going to be a relatively complicated one.

V. How do I deal with these difficulties?

I give the following advice: define clear areas of conflict, develop a system, and don't be afraid to say that things that don't fall into your system are out of the scope of the system, and simply have no RP effect. Then, devise a comprehensive system _before_ you open, and prepare to be flexible about fixing holes. Finally, look at the systems people have already made. Don't reinvent the wheel if you don't have to. Ask the creators if you can borrow their ideas.

More specific advice: clearly define the legs of power which support a faction. Once defined, develop a system of measuring that power. Then, develop systems that enable that power to be increased and decreased. Sound confusing? It isn't, well, at least not in theory. :)

The legs of power, as I call them, are the things which give a faction its ability to do things. They are, in essence, the "statistics" of a faction. To use the Landsraad House example, a list of the legs would include hoarded wealth; ability to generate new wealth; size and skill of military; votes in the landsraad; and the skills of its leaders. Note that the last one refers to people who are most likely going to be player characters, which leads to a later point I'll make, about weaving players into the faction structure.

Once you see the areas which define a faction's ability to interact, you need to then figure out systems that allow for these to interact with each other and with those 'legs' of other factions. For example, an economy is very important to develop, in order for factions to gain an economic advantage over another. A warfare system is also important, for that day when one faction takes the field against another. A political and legal structure is also important... it's crucial that RP occur on the same level playing field, and the laws and politics are a great way to balance that out. More importantly, weave these systems together: success in war should be tied in part to economic strength and political strength. Allow a party to expend political capital to gain wealth. And so on.

VII. Where do I draw the line when designing systems?

This is a _very_ difficult decision. My basic advice is theoretical, and it's clear to me, though I can't always convey it properly. But the gist of it is: look at your theme and decide which areas of it are static and generally even. Then, DON'T make a system for them... just let them rest in the background. But areas where distinctions are dynamic and changing, DO make systems to account for these areas.

And one thing I cannot emphasize enough... whenever you make a system, make it cyclical. Make it so that every faction has some sort of role, something to contribute to it. If you don't do that, power will inevitably reside in the faction which has something to offer but doesn't have an need. This will occur no matter how well or badly they RP.

All this theory could use some examples, so I'll try to illustrate it, using DuneMUSH II at first. When developing the economy and warfare system, I looked at the theme, and determined that some things are just not factors. For example: there was no evidence of any major food shortages in the economy. Therefore, there was no real need to factor in food consumption in the economy. It is just assumed that people can eat. Similary, in warfare, the basic presumption is that most Houses have a small armed force for planetary defense, and that most Houses have no problem affording it. Therefore, a basic, default military status is available to each House at no charge. This is a background cost, since it is in effect even among the Houses. However, an _enhanced_military, either in training level or size, _is_ an advantage and unusual, so it costs IC money to maintain the level above the default. So much for the "background costs" theory.

Now for the "cyclical system" theory. Basically, the key here is to make sure that at some stage of the cycle, each faction has something to offer and to take away. Even if the theme has stronger and weaker factions, they should be able to take part in systems proportionate to their strength and weakness. A negative example of this is a now-defunct mush which shall remain nameless. Its economy consisted of a single commodity: weapons. A very few players were "smiths" and created weapons which were combat- capable. What happened was, players who performed other functions that would inevitably be needed, such as tavern owners who provided drinks and food, and clothesmakers, etc., had no IC ability to garner payment for their wares, since the system didn't force people to go to them IC to get drinks, food, clothes, etc. So, in essence, "smiths" were the king of this economy, and could exact huge prices for something as simple as a sword. Since there was no other IC commodity, this price was inevitable a "favor" or an IC deed. Therefore, major, major power was clustered in a tiny group of players, for no discernable IC reason, but merely because the economy was undeveloped, and non-cyclical. DISCLAIMER: if you know what MUSH this is, and/or were responsible for this system, don't take offense. I know that the focus of this place wasn't on the economy. But it's the best example of this phenomenon I could think of, and it's true to boot.

VIII. Integrating player skills into the systems

Traditionally, the statistics attached to players have been to resolve direct conflict between them and other players. Most of the time, this relates to individual combat. Other times, it applies to magic systems and the like. However, in a faction mush, you have something else to take into account: the ability of highly trained players to influence the fates of the faction itself.

Note: some areas are very hard to code a system, and are best left to RP. An example of this is diplomacy. To code a system by which a skilled diplomat can influence other players to do things a certain way is simply unrealistic and illogical. Let diplomats RP their job, and succeed or fail that way.

However, an area where a system _does_ make sense is that of mass conflict. It is basically impossible to RP a war where 50000 or more people meet on the battlefield and fight. Also, very few MUSHers have the RL abilities that would enable them to RP commanding such a group effectively. Therefore, a system is clearly in order. Whatever mechanics you decide upon, consider including as a factor the skill of your head soldier (in DuneMUSH II, the House Warmaster). The more skilled in strategy a Warmaster is, the better his House's chance for triumph is. Of course, other factors such as the number and skill of soldiers is taken into account. But a good warmaster can genuinely make the difference in a close battle. And because of that, that Warmaster is also an important figure in the House, valuable and prized for the ability to help win wars. But if you didn't account for that, the Warmaster would essentially be one more individual combat badass, wandering around looking to pick fights with individuals. This cheapens RP for everyone when a situation like that occurs.

IX. Fine Tuning

Once you have decided what systems are needed to monitor conflict at various levels between factions, be prepared to tweak and refine your systems once play starts. Nothing exists in a vacuum, except of course an RP system before people start playing in it. You will find that players will find the holes in your rules faster than you could have imagined. They will probably reopen discussions you had with your admin team about how to handle a specific problem.

The key to dealing with this is finding a way to gather information without going crazy. Don't let every little complaint bother you. But at the same time, recognize that sometimes, a player can and will not only find a genuine problem in your system, but will also propose a perfect solution. Have an open mind, and remember that the perfect MUSH hasn't been created, and never will be. The best you can do is to try to make constant improvements.

X. Selecting the perfect theme for a faction mush (or any mush, really)

The perfect theme, in my opinion, would be one in which there were clearly-defined factions which were smallish and centrally located geographically. The actions of individuals could further the status of the faction directly, and frequent fluctuations in power would be

acceptable and thematic.

Sound familiar? This is because most mushes, intended or not, follow this pattern. People tend to cluster in a few places no matter how spread out geographically the mush is. Factions tend to have 5-15 members, since anything bigger gets unwieldy. People try to advance their status and that of their faction, and often succeed. Quite often, the whole fact of a faction is tied to the actions of one or two folks. Wars, assassinations, coups and the like occur at an alarming frequency. Why? Because these are exciting, and who MUSHes to be bored? Also, faction heads lose access or quit playing, and so they stop showing up, and their characters meet a violent IC fate as a result.

Therefore, why not go ahead and make a theme that fits these criteria, and others I may have missed? Then, the systems on the mush would perfectly align the theme you are trying to portray. The problem is, I haven't thought of such a theme yet. Have you? :)

XI. RPing conflict

Some notes on RPing conflict on a large or small scale: (Note: this essay is directed at Dune2MUSH, but can apply to other mushes at well, particularly faction-oriented ones).

First, a word on what I will call 'MUSH distortion.' What I am talking about is the inherent problems in the medium in which our game takes place. There are many differences between a simulated environment such as ours, and the hypothetical 'real world' that we are simulating. These differences create a hazy area which can be exploited by unscrupulous mushers. What are some of these differences?

1. people not being online at the same time, even though their characters _are_ in the hypothetical world, doing _something_ :)
2. the impersonality and relative slowness of interaction in the medium
3. differing visions of theme, of the incident, even of the room that RP occurs in.
4. picturing OOC friends and enemies in a purely IC light

A good example of how this distortion can be exploited is this: a Househead is on vacation for two weeks. That House's mortal enemy engineers a major smear campaign against the House in that two weeks. The victim House cannot fight back. Now, this is simply unrealistic and unfair. Just because the House head's player is on vacation, doesn't mean that his House would not be well-led in the interim, and that a counteroffensive wouldn't be mounted. Clearly, OOC cooperation could avoid this sort of thing.

The major key to 'conquering' the ill effects of MUSH distortion is to realize that it is out there. Once it is defined, it can then be worked around via cooperation. There are reasons other than pure gamesmanship to do this. First and foremost, cooperation to avoid MUSH distortion focuses the conflict between factions on a higher plane. If you know that you aren't going to get blindsided by any dirty tricks, then you can throw yourself into the meat of the conflict, which is RP. Almost all conflict involves coalition building and politics. _That's_ where you should concentrate your energies, and, coincidentally, that's also where all the fun is!

To this end, the following is suggested as a means to keep conflict between factions where it should be: IC. Whenever a major faction conflict is brewing, the heads of that faction should get together for a 'parley.' Ask for an admin or some other neutral party to sit down with you if you want. Talk out your vision of things as they stand, and perhaps some possible outcomes. Try to agree on the crux of your conflict: is this a battle of who can win the support of the Landsraad? Of who can get the Emperor's support? An all-out treachery campaign, where anything goes? The focus should be on ironing out what your characters would know. Don't give away legitimate secrets, but give as much information as you can in order to help the other side understand how things stand.

Here is an example of a Parley between the heads of House A and B, who have had several clashes in the landsraad, and who are bitter trade rivals. Recently, House A tried to lure away several close trading partners of House B, who found out about the maneuver, and is planning to retaliate with an offensive to scare House A off. B @mails A telling her that he (B) is planning some faction-level conflict, and that he'd like to talk.

```
A: What's up?
B: Well, I think that the course of our RP has progessed to the point
where I am going to try to commence some hostilities against you.
For starters, I want you to know that it isn't personal, and I want
to get some great RP out of this.
A: Hmmmm. What kind of hostilities?
B: Well, I'm not going to tell you exactly what I'm planning, but I'd
like to go over a few things and maybe answer some of your questions
before I actually do it. First off, do you have any problems with me
taking action against you?
A: No, in fact, I'm surprised it's taken this long. I'm still nervous
though, I've never been in a nasty conflict before.
B: Yeah, me too. Well, I'm glad you understand the why. Now, I'd like
to propose some ground rules. First off, I'm going to do my best to tell
you things that I think you'd know, and I'd like you to do the same for
me. For example, your character would know that I'm considered fairly
trustworthy. Also, it's no secret that my diplomats have been rallying
```

```
support for me. They haven't done so in back rooms.
A: Oh, I think I see what you mean. Well, as you probably know by
now, my character is very sneaky. Right now, what you see is what
you get. You already know about the attempt to steal your DU suppliers.
B: Okay, now, another thing. I'm not saying that I'm planning this, but
if it comes to the point where the conflict moves to the military aspect,
I suggest we meet together with the RP Admin in charge, and when
he does the combat numbers, we work together to 'co-author' what
happened, and how.
A: I don't know about that...
```

And so on. This is just an example, there are many many ways that faction conflict can be coordinated. Be creative. Set your own chances of success or failure if you want. Ultimately, the reward will be higher quality RP, and conflict that you can be proud of.

XII. Final notes

Comments on this should be directed to Fenring@DuneMUSH II, currently located at mindport.net 4201. We have frequent RP seminars where we try to tackle topics that relate to faction conflict, and means of creating and improving RP in mushes in general. If you want to discuss this, feel free to come and get a hold of me there.

MUSH design (Mephisto)

There's a few factors that should be kept well in mind. I'm not going to elaborate too much on them. I feel that anyone reading this will want to walk away with the essence of ideas instead of expositions.

- Set goals for your MUSH that are achievable and practical; don't forget them. Setting goals (what you want the MUSH to be) that are clear, published (to all the admin, and even the players), and reasonable is one of the best ways to define in your mind and for everyone else the feel of your MUSH. Whenever I visit a MUSH, I always ask the wizards what the goals for the MUSH are; by setting your goals, you define what it means for your MUSH to be "successful." Some believe that raw login count determines the success/failure of a MUSH. Not so. I've seen MUSHes that have a low player count, but those players are among the most excellent, and the ensuing activity among the most satisfying and fun. There are many valid paths to success; define the one that's most meaningful to you and your admincorps.
- Realize that there are different kinds of players, and pick your target audience with care. Some people think there are two kinds of MUSHers: potential players and non-potential players. Nope! There are several ways to distinguish between people who MU*. The first distinction I tend to make hinges on maturity. :) The next depends on taste: are they social people (you know a few - they hang out on RP MUSHes only to visit with their friends)?, are they people who enjoy RP and TPing, are they coding/programming types? There's generally room for all, but you might want to tailor your MUSH to be as freeform or "exclusive" as you like. Realize that for every decision you make you're limiting your target audience, which is not necessarily a bad thing. Depending on the nature of your theme, you may well want to target mature, sophisticated players!
- Pick a theme that's not just a passing interest. An enduring, powerful series of books (or creation), and don't be afraid to be bold in your implementation. There's something to be said in faithfully recreating a world, but a world that lacks vision won't inspire the players. When considering a theme, think about what PRECISE and PARTICULAR elements attract you to the story, the setting, the characters. There are certain deep and powerful themes that run through literature, art, drama - any creative effort; and, as a creative effort, MUSH is no exception. Strive to understand why you think your theme is so appealing.
- Implement your ideas powerfully. Having grasped the essence of a world, be creative, bold, and clever in how you implement your ideas. A MUSH that's supposed to be tongue-in-cheek and comic should be -consistently- so. These elements should pervade everything, from the news, to the descs of rooms, to the presentation of the code onscreen, to the behavior of the admincorps. There are a LOT of MUSHes out there competing for players. Don't be afraid to distinguish your MUSH in every way you can.
- Build a MUSH culture. Ritual and familiarity are the foundations of a culture; strive to create a culture among your admincorps and your MUSH in general that reflects the goals, purpose, theme, and mood of the MUSH. Create things that happen only on your MUSH, and that people will miss if they don't play. This is a very elusive thing, but every great MUSH I've seen (there've only been a few) has had its own culture in abundance. Cultivate yours.

Administration tips

How should a God organize his/her administrators? How should administrators work together to run the MUSH? These and other questions are considered in the tips below.

- Admin Roles (Talek)
- Admin, players, meetings, management, and involvement (Javelin)
- Conflict resolution for admin (Javelin)

- Administrative tips (Mephisto)

Admin Roles (Talek)

A MUSH is not run by just one person (at least, I've never seen one that is). A MUSH isn't even run by one type of person; a wide variety of talents are necessary. The most common roles that I have seen are:

Manager
: The manager (most often the God of the MUSH) keeps track of who's doing what and what needs to be done. In some situations, the manager also assigns tasks to people (though a task will often be better done by volunteers than by forced labor). Unfortunately, the manager will often have to deal with admin politics more than anything else that ought to be done.

Rule Enforcer
: The rule enforcers make sure that nuisance players are identified and dealt with. This can include everything from lecturing the player to @nuking them.

Judge
: The judges mediate conflicts between players and admin (and, occasionally, among the admin themselves). While the jobs of rule enforcer and judge often fall on the same people, it is generally a bad idea for one person to be both rule enforcer and judge for any given situation.

Player Helper
: These are (probably) the people that the players see most; their purpose is to answer player questions and help players get started on the MUSH. Never underestimate the amount of work this can be.

MUSHcoder
: The MUSHcoders deal with making sure all MUSHcode systems get written and maintained. They also deal with security problems arising from said systems (or MUSHcode in general). It's also good to have an experienced MUSHcoder who is also a player helper; many of the questions asked by non-newbies deal with MUSHcode and how to use it. It is often helpful for MUSHcoders who work on large systems on the MUSH to have access to a test MUSH where they can test their changes without the possibility of crashing the real MUSH.

Server Hack
: The server hack (usually one (or none) to a MUSH) adds new features to the MUSH server itself. Make sure you trust your server hack; there are no security protections against what can be done in the server. It is almost mandatory for server hacks to have a test MUSH to try out their changes; a single typo can keep a MUSH down for days.

These roles are not exclusive; admin normally fill three or more roles. The trick is to make sure you've got at least one of everything (with the possible exception of server hack); big problems can result from not having someone to deal with each of those jobs.

I personally have been all of these things (except manager) at one time of another, but I tend to be a MUSHcoder and server hack more than anything else, and that colors my views.

Admin, players, meetings, management, and involvement (Javelin)

In my opinion, it's the people that make the MUSH, and running a MUSH is more about management of people than about hacking, MUSHcode, or building - though all of those are important!

Choose your admin with care, and look for balance. You don't want all psychocoders - you need strong player and newbie-help admin, people who have experience in building and coding, and a few people who are just very trustworthy and friendly and serve to help everyone on the admin team get along.

When I am God, I reserve to myself the final approval on admin and the power to de-admin folks who fail to live up to the standard of admin'ing I expect. Other than that, I prefer to let the admin as a group make all the important decisions, serving as a facilitator and final arbiter as needed.

That's one aspect of my general belief that the more you can involve your players and admin, and give them responsibility and a stake in the game, the more enjoyment they will get and the more constructive and innovative your game will run.

One of the ideas I'm proudest of from DuneMUSH was the "player positions" or "awards" which gave recognition to players who contributed to the MUSH as a whole, through coding, building, role-playing, helping newbies, or judging. The system recognized that there were a lot of different types of valuable player contributions, and as players contributed, they were given inereasing recognition and powers to help them contribute further. It's also a great way to identify potential admin!

In short, I believe that God should treat every player fairly, seek positive and constructive "win-win" solutions to problems, and continually seek to improve the MUSH.

Conflict Resolution for Admin (Javelin)

As God, you may well be called upon to mediate conflicts between admin or between admin and players. You may even have a conflict yourself. Here are my tips for how to handle such situations:

Willingness to resolve

The key to all conflict resolution is for all the parties involved to agree in principle that they are willing to resolve the conflict. Maybe it'll be for their own peace of mind, and maybe for the good of the MUSH, but either way, willingness to resolve is crucial.

If you're in conflict

If you're the admin in conflict, and you've decided that your goal is to resolve the conflict for the benefit of yourself and the mud, here are some steps you can take:

1. Initiate communication. Take the initiative with the other admin, and agree to discuss the issue with an eye to working out differences for the good of the MUSH.
2. Provide constructive feedback. Yelling at another admin is unlikely to result in positive change. Instead, indicate specifically what your concern is and how their actions affect you. This type of communication places trust in the other wizard to see that they address the situation - trust which they will appreciate.
3. Listen to constructive feedback. When another wizard is giving you feedback about your actions, try not to get defensive. Listen actively and be sure you understand and clarify their concern. Then you can take action to alleviate it.
4. Maintain a professional relationship. If you can't resolve your differences, at least agree that they don't have to make it impossible for you to work with each other or to respect each other as people and wizards. The ability to respectfully disagree is the hallmark of the mature wizard.
5. Follow up. Set a time when you and the other admin will evaluate the results of any changes you decide to make, and see if they've rectified the situation.

If you're mediating conflict

Here's a 7-step procedure for mud troubleshooting. It's useful when two players or admin are in conflict, when someone reports a game problem, and in many other situations.

1. Get all the facts. Action without knowledge is bound to lead to disaster in the long run. Before making a decision or taking action, be sure that you've got all the information you need to make the best decision for the long term.
2. Don't take sides. While you may be rendering a judgment which will be to the advantage of one player and the disadvantage of another, it's important to be fair to both sides and consider each one as objectively as you can. Your job is not to take a side, but to take an action.
3. Discuss with other wizards. Constant communication between wizards has many benefits. It gives you the advantage of opinions and options you might not have considered. It also keeps the other wizards informed about your decisions, which prevents players from playing one wizard against another. The only exception to this rule is if the situation is personal or warrants privacy.
4. Encourage responsibility in others. Solving a conflict between players is great. Even better is when the players learn to solve their own conflict. Actively seek to involve people in the solution of their problems. Teach admin the steps discussed above. Making players more responsible for the mud also increases their commitment.
5. Look for win-win solutions. Usually we seek compromise solutions in which one party gives and the other takes. But often there are win-win collaborative solutions which might satisfy both parties and enable them to engage in better future relations. The key to finding these solutions is a willingness of the parties to try to work together, which requires some maturity on their part.
6. Leave yourself an out, if appropriate. Although you're God and the buck stops with you, consider making decisions which leave room for change should new information come to light. However, to maintain consistency, decisions should rarely be reversed, or players will come to see the admin as wishy-washy.
7. Follow up. Explicitly set a time at which you will review the results of the action or decision and make sure that it's really working the way you wanted it to. This is crucial; making a decision and then forgetting about it will give the impression that you say a lot but don't mean it.

Administrative tips (Mephisto)

Being a MUSH admin is, depending on the MUSH, the theme, the codebase, and the phase of the moon, a pleasure and a pain in the neck. The key is patience and consistency. Here are a few things to keep in mind, for both the experienced and the green administrator:

- It's a game. This is the hinge upon which the door to a MUSH's succeeds succeeds or fails. To be honest, I've seen people who

"use" MUSHes out of an isolated interest for the particular theme, out of a desire to master the programming language, or to impress a love interest. These people are USERS, not players. Most people, though, are PLAYERS. And as such, your MUSH is a game. Unless you're doing a highly non-traditional MUSH, you should ALWAYS remember that having fun and being interesting and/or amusing should be among the highest priorities of any game that aims for success (depending on your goals, of course).

- Know when to step down. One of the main reasons for MUSH death is an admincorps that has gradually lost interest, ability, or time to do what needs to be done to run a good MUSH. Times like these are the most painful for an administrator. You're torn between giving up your position of power and keeping your activity and creative role. You know that, to be fair, you can't have it both ways. And sometimes, the MUSH is in dire straights and stepping down might well mean its death. That's the part of being a wizard that they don't tell you about in the manuals, but it happens a lot. Sometimes, it's better to step down or close a MUSH than it is to string yourself or your MUSH along. There's a time to nuture and a time to let go, and let the MUSH succeed or fail on its merits. In a way alike to none other, for MU*ers, the journey is the reward. And that is that.
- Communication between admin is the most important thing to be concerned about. Complications that arise from mis- or non-communication are among the most insidious and can have a disastrous affect on morale. Have an email dissemination list setup that everyone can email to. I prefer one list to which all the admin are subscribed, and one list to which all the admin and any players that have the desire are subscribed. Use the @wall/wizard and @wall/royal channels. Set up an online bulletin board or "captain's log" to which any admin can add, or review. These are essential tools: use them!
- Never stop playing. Perspective is key for an administrator: knowing what it's like for people new to MUSH, to the Internet even, knowing what being a PLAYER is like. Once an admin, it's likely your outlook was changed, but always try to remember what it was like to be a player, with the tools and services and toys that your MUSH provides. It's easy to say "I want to make a MUSH I'd play on", but sometimes, it's hard to remember what being a player is all about.
- Remember the magic. When you were just a player, life was different. Remember? The exploration, the wonder, being concerned not with arranging the next meeting, but with getting online to RP. To figure out how to get that text to line up straight, to figure out how to lour the king away from his bodyguard... Being an admin can take you away from all that to the point where you might forget, might lose track of what it was like to PLAY the game. Never forget, though, that the best administrators are they who best understand what makes a game great. When a new Royal or Wizard joins the ranks, you can smell their interest an enthusiasm. Try very hard to not let that sense of play (even in administration) be overcome by the somewhat "traditional MUSH admin somberness" that plagues administrators everywhere. MUSHing - for both administrator and player - should be fun. Period. You won't always succeed, but you should try to remember what it was like the day you first discovered MUSH, and let the measure of those first magical moments be your guide in all you do.

MUSHcode tips

As anyone who's read Amberyl's MUSH manual knows, there's much more to writing MUSHcode than @create. Large MUSHcoded systems like combat, economy, and dynamic space systems pose particular problems for MUSH Gods. These tips offer suggestions for tackling MUSHcode.

Queue and CPU efficiency (Javelin)

If you haven't read Amberyl's MUSH manual thoroughly, read it. The psychocoder section does a nice job of explaining the difference between queue and CPU efficiency, and the tradeoff you make when you code.

DB Growth (Gohs)

The single biggest reason for a MUSH to die, other than having its site yanked, is excessive or out of control database growth. If you are running a social MUSH with minimal or very freeform roleplaying, you can simply slap belated @quotas on everybody and stop the growth that way - Rhostshyl had survived quite some time when I was on it, so this can work, although players will tend to find it a frustrating situation, and if they become too frustrated they may well pack up and move to another MUSH. Restricted building and @quotas from the start may seem harsh and may discourage some builders unless the admin are very conscientious about making sure

that those who want to build have the opportunity to, but the reality is that most players come on to play, not to build, and restricted building and @quotas from the start can be tolerated, and in the longrun will be ignored and treated as normal. Though it may seem draconian, this can save you a great deal of grief later on, as well as extending the lifespan of your MUSH.

MUSHcode Systems (Talek)

Many MUSHes these days have large, complex systems of MUSHcode to do cool and whizzy things, ranging from automated combat to economic simulation to autocreation of terrain. I helped introduce this sort of code into the world of MUSHes, and now I'm having second thoughts.

The benefits of MUSHcode systems are numerous: they can make make information more easily available, they can act as an impartial (but stupid) judge for things like combat, they can make seemingly complex automatic responses simple to code, and many more things. The drawbacks are few but substantial: MUSHcode systems often use large amounts of CPU time, they require learning a new set of commands to use the system, and in many cases they just add complexity to the game instead of enjoyment.

A well coded, documented, and integrated MUSHcode system can be a great benefit to a MUSH. When I see one, I'll be sure to let everyone know.

Combat systems (Javelin)

I've now written 3 different complete MUSH combat systems. If you're considering a combat system for your MUSH, think carefully about how you want it to work. Should players have to type attack commands repeatedly, or should a single command run the whole fight? Should there be both attack and defense commands? Should the combat system do the emits and assess damage, or simply tell the players how to role-play? How should player healing be handled? Should there be elements of fatigue, agility, skill? How will you prevent players from abusing combat? How will you ensure that combat works fairly even if a player is lagged, the queue is lagged, or the game gets shut down during a combat? I'm not going to give suggestions here, but just mention these things in the hope that you'll have a head start if you decide to write one - it's a great way to improve your MUSHcoding!

Coded Systems (Gohs)

Recently, (as in the last couple of years) the use of large MUSH coded systems has arisen over a large number of MUSHs. Even in those cases where the "systems" are relatively small, the sheer number can often be daunting, and the phenomenon is somewhat like the coathangers, nuclear warheads, and rabbit slippers - they tend to multiply, often out of control. '+help' files often rival the news files (something which seems an obscenity to me, but...), and the proliferation of global systems has attained truly staggering proportions on many MUSHs.

If there is one thing to remember about global code and systems, it is this:

Only code a global if it will substantially add to the enjoyment of the players and this enjoyment is in excess of the lag it will inevitably result in. '+commands' are nifty and fun to make, but the vast majority fall under the category of 'gadgets' - make 'em, play with 'em, then throw them out. Every neat little code toy put in the Master Room (#2) will add to the lag. On large systems this may be bearable, but the chances are good you won't have the machine of your dreams, and in any event it's generally a good idea to save on unnecessary lag in preparation for unavoidable lag.

Large coded setups, or systems, can include everything from weather, ecology, magic, tides, economy, time, skills, and, most commonly, combat. They are generally only used on MUSHs that are dedicated to roleplaying, but there are other types of MUSHs that may see them. What goes for +commands goes tenfold for coded systems: use them, yes, but don't use them unless they are a major benefit to the gameplay. It is very, very easy to go out of control here, and the temptation to add 'one more system' is hard to resist. It's often a good idea to determine what systems you want and then sticking with that.

That being said, there are a number of tricks that can help improve efficiency. Every attribute in the Master Room (#2) is evaluated every time a player enters a command. Because of this, when coding a system (or any global, for that matter) store as much information as possible on objects outside of the Master Room that can be called to by the object in the Master Room. Another thing to keep an eye on is organization of the Master Room. Planning from the beginning what objects will be in there and what will be on each of the objects can make tracking down bugs and buggy code an enormously simplified task, and is almost always well worth the effort. Yet another thing to be wary of are systems or parts of systems that will require the MUSH to constantly check thins. A good example of this can be found on the way time is handled on GohsMUSH. Instead of running an eternal @wait that periodically checks to see if the time has changed, it only checks to see what the time is when a player performs an action that requires that the time be displayed. A description of a room that changes as time goes by would have in its code to check for the time whenever someone looks at it - this is far more efficient than having the MUSH actively check to see if the time has changed and then change the descriptions appropriately. A setup like that could result in rooms changing their description even when nobody is present in the room, thus resulting in a waste of

computing resources.

If you do decide you want to use systems, don't reinvent the wheel and don't have code that will on average detract from the enjoyment of the game. Look around to see if what you want has already been done. Generally it won't, but things like bulletin boards and 'who' machines are very common, and it's much more efficient of your time to ask permission to use code that's already been done than to code it from scratch. Don't follow this *too* slavishly, though. If the only code you can find is inefficient or doesn't do quite what you want, you may well be better off coding it from scratch. In addition to not reinventing the wheel, it's important when determining what systems you are going to have (if any) to keep in mind that, in general, it's the enjoyment of the players that will determine the success or failure of the MUSH. To give a good example of a system that went too far, I'll take one of my own blunders: a coded system of eating and sleeping. On Mua'kaar, I had Helbergina code a system whereby players had to eat on a regular basis or else slowly begin to lose health and finally die of starvation. While this was definitely "neat" and quite "realistic", it contributed little to the MUSH save for frustrating players, who, damn it, wanted to *roleplay*, not have to sit around worrying about how they were going to pay for their next meal.

Combat

Probably the most commonly attempted system is combat. While on LPs and Dikus it is generally non-player automations that get the sharp end of the sword or the wrong end of the maula, on MUSHs it generally winds up being players and only to a much lesser extent non-player characters. This being the case, I've seen three ways of handling the lethality of combat systems. Each has their problems, and which one you use will depend on taste and on what exactly you intend for the MUSH. The first method is reminiscent of the "kill" command (which is usually taken out). The "kill" command would send a character to the room they are linked to, or their home. This may entail the loss of their possessions and often a time spent as a ghost. One of the Tolkein MUSHs, Elendor, used a system like this for a while. The second method is to make combat very non-lethal - to make it very hard to kill characters. Usually this works best if the characters go unconscious before they die, making it very hard to kill someone outright. This, like the first option, has the advantage of not having to deal with too many players who are upset because their favorite character was fried by a laser. The third option is to make the combat system anywhere from somewhat lethal to very lethal. If combat is supposed to play a peripheral or secondary role on the MUSH, this can often serve to maintain a sense of realism that many people are attracted to while at the same time discouraging too much combat - a system that is extremely lethal generally finds players loathe to risk their characters unless absolutely necessary. One rather interesting benefit of this form of combat system that I've noticed is that when a system is very lethal, players will often resort to tactics rather than just charging into combat. Ambushes, traps, and ganging up are all forms of this.

Economy

Another system that is very popular and has been attempted many times and only very rarely successfully is a MUSH economy. One of the first things to decide when setting up a coded economy is to decide whether you are going to use MUSH coins (pennies, credits, coins) or coded currency (where you see things like 'check purse' and 'pay <character>=<number> <coin type>'). There are problems with both, and each has its advantages. One major advantage of using MUSH coins is that they are hard coded, they are faster than a system of coded currency can be, and they require little or no additional code. The disadvantage of using MUSH coins is that they are also used for building and for queue-intensive commands, meaning that a player may find their in character money supply dwindling because of out of character actions. On the other hand, a system of coded currency is additional code and represents additional commands for the players to learn. Dune and Pandemonium are good examples of some early attempts at setting up an economy, and both are good examples of why it is so difficult to set up a workable economy. On both Dune and Pandemonium there were only a very small handful of items or services that were exchanged with much frequency. Weapons, armor, and in the case of Dune, training fees and shuttle tickets were the primary items of value in the economy. Both systems worked, and both can claim to have had working economies, but they were nonetheless very simplistic, in that they had a very narrow range of commodities - and the fewer the number of commodities, the more difficult it is to stabalize and spread out the economy. In a system such as these, very often it is only the soldier or mercenary who has any major role in the economy, which can serve to focus people onto these aspects of the MUSH - specifically, combat. Thus, if you are going to run an economy, it is worth determining what items and services characters will be able to purchase - and, extending from this, it is important to find commodities that players will *want* to buy and have a need for.

From this point, there are two types of economic activity that are possible to be engaged in. The most common is the micro, or the part of the economy in a MUSH sense where the characters are buying personal items - weapons, armor, non-player character bodyguards, etc. Any item in this has to have a need for that item or else a desire for it. People don't like to die or have fun engaging in combat, so there is a demand for weapons and armor. On Mua'kaar, the ill-fated food system required that characters eat regularly or begin to starve, so there was a need for food. As a general rule, on a MUSH, focus on cmmodities that are desires rather than needs. Players come to a MUSH to have fun, and though they'll fulfill a need, they won't obtain any particular enjoyment out of it, which they will if they are fulfilling a desire. The other part of the economy is the macro. Dune II is a superlative example of how this can be done. In this instance, you see a system of Houses and factions that are trading, producing, and consuming goods - often abstract - for the purpose of attaining in character power and/or prestige. Needless to say, these two parts of the economy can overlap greatly, and there is no absolute requirement for either or both. It is entirely possible to code a macro economy without a micro economy, or to code a micro

economy without a macro, or both, or none. It depends almost entirely on what you, as the administrator, want to do with the MUSH.

A micro economy has a further problem that the macro economy does not have to worry about. Weapons and armor are generally registered or otherwise marked as "official", and only these official weapons and armor will work in the context of a coded combat system. If any player can simply @create a weapon or piece of armor, then the value becomes nil, and it is virtually possible to sustain an economic relationship with that item. Registering those items that are integrated into the economy is a very good idea; one idea that seems to work particularly well here is to have, say, all weapons owned by a Weapon Master character, and all armor owned by an Armor Master character, and so on. Not only does this make it easy to find all items of a particular type if you need to change all of them for some reason, but it also provides for a built in check - Weapon #1 is legal if and only if it is owned by the Weapon Master character, for example. There is another, slightly more drastic tactic that can be used in tandem with this, this being to require that *all* in character objects be in some way registered or owned by a common character. Thus, if a character wants a lamp, he or she has to buy an "official" one. Needless to say, this will work considerably better if the item in question has a coded use that cannot be accomplished by an object that someone has just @created. A lamp that will change a room's flag from DARK to !DARK has value, for example, that a player-created lamp would not.

DSpace (Talek)

One of the things that I'm best known for is Dynamic Space (DSpace for short). I originally wrote it to deal with Belgariad I's oceans; at the time it seemed like a good solution to a small problem. Since then, DSpace has used for everything from open countryside to city streets to mazes. Unfortunately, while DSpace is good for large tracts of fairly uniform terrain, many of the uses it has seen are woefully inappropriate.

For those unfamiliar with DSpace, I'll give a brief description: DSpace is a MUSHcode system which automatically digs and destroys rooms as players (or other things) wander through them. This keeps the number of rooms that actually exist to a minimum without limiting the amount of terrain represented. It does this through the use of zone exits which compute where you should go based on where you currently are and which direction you are going; if the room that you should be going to doesn't exist, DSpace creates it, and if the room you leave is empty, DSpace queues it for destruction. The end result is that DSpace trades CPU time for DB size.

All of this is great stuff if you are dealing with miles and miles of uninhabited ocean, desert, or wilderness; anything which is huge and not very populated. However, when used for something like city streets, which are used nearly continuously and are not all that uniform, all DSpace does is drastically increase the amount of work that the MUSH server has to do, with very little savings in the DB.

Now, a few years after writing Dynamic Space, I almost wish I never had written it in the first place. Almost everything that it does can be better done by rethinking the layout of an area; the primary questions being "Does the area need to be this big?" and "Is the area uniform enough so that using DSpace will be less work than individually crafting the rooms?" and "Is the area going to be so rarely used that the CPU time is worth the reduction in DB size?". All too often, one (or more) of the answers to the above questions is "No"; unfortunately, people seem to use DSpace anyway.

DSpace is one of the big ugly warts on the face of MUSHdom. Despite efforts to make it a new, clean, shiny wart, it is still a wart. Avoid it if you can.

MUSHcode for player purges (Rhyanna)

[Rhyanna notes that the tinyfugue purge macros which I discuss are incomplete. /nuke may leave rooms with more than 3 exits, and the lsearch() may fail to get all the players due to buffer length problems. - Javelin]

This is the MUSH code that I use for purging. All this code exists on a wizard player that is @uselocked against all other players. It could be easily modified to exist on a wizard object that had an appropriate @uselock.

'check' is the equivalent of Javelin's /ppurge:

```
&DO_CHECK me=$check:&purgees me; @tr me/for-all-players=summarize
&FOR-ALL-PLAYERS me=&num-objs me=first(lstats(all)); @tr me/for-all-players-aux=%0, 0
&FOR-ALL-PLAYERS-AUX me=@switch/first 1=gte(%1, v(num-objs)), {think Lists done.}, match(type(#%1), PLAYER), {@tr
```

The 'summarize' attribute is the one that contains the policy about who should be purged and who should not be purged. Castle D'Image's policy is that players can be purged after they have not logged on for 90 days, or 30 days if they log on only once. My 'summarize' attribute, formatted for readability, also shows the @comment and @away attributes. The @comment and @away attributes are used to record reasons why players should not be purged; these reasons are usually either 1) that they intend to come back by a definite time (if they are gone for summer vacation, or the like), or 2) That they own rooms or objects that other people would miss.

```
&SUMMARIZE me=think %1;
```

```
  @stats %1;
  think Last on: [get(%0/last)];
  think get(%0/comment);
  think get(%0/away);
  @switch/first 1=
    gte( sub(secs(), convtime(get(%0/last))), get(me/90-days) ),
    { think Recommendation: Purge--not logged in for 90 days.;
      &purgees me=setunion(v(purgees), %0)
    },
    and( gte( sub(secs(), convtime(get(%0/last))), get(me/30-days) ),
         lte( sub( convtime(get(%0/last)), convtime(get(%0/creation_date)) ),
              100 )
       ),
    { Think Recommendation: Purge--Only logged in once.;
      &purgees me=setunion(v(purgees), %0)
    },
    { Think Recommendation: Leave Existing.}
&90-DAYS me=7776000
&30-DAYS me=2592000
```

My equivalent of Javelin's '/nuke' macro is the following set of commands. I chose to make my 'purge' command require confirmation of the purge; I don't want to purge any player accidentally.

```
&PURGE me=$purge *:@switch/first 1=match(%0,all),
        { think Surely you don't want to purge the whole MUSH?},
        not(isdbref(num(*%0))),
        { think %0 is not a valid player.},
        not(match(type(*%0),PLAYER)),
        {think You can only purge players.},
        hasflag(*%0, connected),
        {think That player is currently connected!},
        { &purgee me=name(*%0);
          think {Do you really want to purge [v(purgee)], with [first(lstats(v(0)))] items? ('pyes' or 'pno'.)};
          @tr me/summarize=num(*%0),name(*%0);
          @wait me/60=
            { think [switch(type(*%0),player,{'purge [name(*%0)]' canceled.}, {[v(purgee)] purged.})];
              &purgee me
            }
        }
```

The 'pyes' command does the actual purging. It uses the 'iter(v(types),...)' so that it destroys all the exits the player owns before any of the rooms, so that the 'The room must have less than three exits' message doesn't happen. 'pno' cancels the purge; it also cancels if 'pyes' is not typed within 60 seconds.

```
&PYES me=$pyes:@dolist/notify iter(v(types),lsearch(v(purgee),type,##) )=
        @nuke ##
&TYPES me=EXIT OBJECT ROOM PLAYER
&PNO me=$pno:think {'purge [v(purgee)]' cancelled.}; &purgee me
```

Server tips

Most Gods wrestle with server issues now and then. Server hacks and contributors to the PennMUSH distribution are steeped in them. Here are some tips for keeping your code straight, handling some common problems, and improving your peace of mind.

Code Management (Talek)

Keeping track of code and all the modifications that have been done to it is an important part of maintaining any large system. I cannot overstress the importance of using a revision management system like RCS or SCCS for both the server and all important MUSHcode systems you have. The ability to easily undo a change can save you days of painful bughunting.

Recovering a database from a core file (Rhyanna)

I have recently managed to recover a PennMUSH 1.50 database from a core file. I will explain what I did, in the hopes that this technique may be useful to others.

I will assume that at the beginning, you are in the game/ directory, and that your core file is in the same file, named core. You should also have a debugger available. I only know gdb, so I can't give directions for using debuggers other than gdb.

1. First, copy the core file to a safe place so that you won't accidentally overwrite it. Therefore, if something goes wrong, you'll still be able to try again.
2. If you are currently trying to run the MUSH with an old database while you restore another database from the core file, make a backup copy of outdb.Z. Also connect to it and use @uptime to make sure that it will not save for a while. Personally, I would be paranoid and wait until I had at least 40 minutes before it saved again.
3. Next, get and install undump. It might possibly be installed on your system; more likely, it won't be. You can get it by anonymous FTP as part of the TeX distribution; one site, for example, is uceng.uc.edu, in the directory /pub/wuarchive/packages/TeX/support /undump/. Undump, it should be noted, is about as system-dependent as a program can possibly be, and there's an excellent chance that there won't be an undump available for your system. In that case, you won't be able to use this method.
4. From the game/ directory, do this: undump core.netmush netmush core
5. Load it up in your favorite debugger. For concreteness, I am using gdb: gdb core.netmush
6. If your core dump was caused by data corruption, you should fix that data corruption now.
7. Then, from the (gdb) prompt, type: print dump_database()

If there is no data corruption, this will write out a copy of the database as it was at the time the core dump was generated.

This database will be generated as outdb.Z (or whatever your standard output database is). If you are currently running the MUSH, you should immediately rename this outdb.Z so that it does not get overwritten the next time the running MUSH saves.

Working with a corrupt core file (Rhyanna)

Suppose you are in my situation: you are an avid server-hacker, but often you get core dumps that are inconveniently truncated, so that a core file that should be 21M long ends up being from 300K to 900K long.

You can still get very useful debugging information out of even a partial core file with your favorite debugger:

Open up the core file in your favorite debugger and look at the variables ccom, cplr, wenv, and renv. These are all global variables; they have the following meanings:

- *ccom* is the last command that was run.
- *cplr* is the dbref of the object doing the command.
- *wenv* is the values for %0-%9.
- *renv* is the values for %q0-%q9.

These variables get set just before every call to process_command(), which is the procedure that handles all the command parsing. In my experience, they're low enough in memory that they can usually be recovered from some of the most truncated core dumps around. And they provide an .enormous. aid to diagnosing where a problem may lie.

Recovering from a lost God password (Amberyl)

If you lose the password of a single wizard, just have another wizard @newpassword that wizard.

If you lose the password of the God character, you're in a bit more trouble. However, hand-hacking the database is _not_ the solution here (or at least, not the optimal one), and trying to change the password under the debugger is just heinous. [Editor's note: Rhyanna has another opinion about this.]

Take another character who is a wizard that you _have_ the password for (or even just any other character with a known password). Note the dbref. Shut the game down. Go into the options.h file, and change the definition of GOD to that dbref. Recompile. Restart. Log in that character. Change the password of the old God (since now after the recompile he's just an ordinary wizard, and the character you have logged in as now God, this can be done). Shutdown. Re-edit the options.h file to its original state, recompile, restart. Voila.

Editor's Note: The technique described above has one significant caveat for the unwary. When you redefine God, all the "Garbage" objects will be owned by a non-God player, and therefore won't be garbage. This will go away when you fix the old God's password and redefine God again and restart.

Recovering from a lost God password (Rhyanna)

I disagree with Amberyl's statement that "trying to change [God's] password under the debugger is just heinous." It's actually pretty easy, if you do it right--easier than recompiling with God redefined.

1. Set a breakpoint at do_newpassword(), and let the MUSH run.
2. Have some character do '@newpassword God=<desired password>'. You'll hit the breakpoint.
3. Change the value of the 'player' parameter to God's dbref. For example, in gdb, you would probably do 'p player=1'.
4. Let the MUSH keep running. Now, God is trying to @newpassword himself-- and lo and behold, he can.

You won't get any notification that the password has been changed, but you can test that it has been fairly easily.

One obvious corollary of this: if you don't trust your site-admin, you are out of luck, because it is impossible to defend against a site-admin who wants to screw you. But we knew this anyway.

Using MEM_CHECK to find memory leaks (Rhyanna)

One of the general sources of hairiness and complexity in the MUSH server is memory management.

Actually, a MUSH server is an applications that would really benefit with an incremental copying garbage collector. The usual pattern of CPU usage for a MUSH is one of brief bursts of activity, separated by relatively long periods of idleness. If you could use a nice, modern garbage collector to relegate garbage collection to the slack periods when no command was being processed, you could significantly reduce player lag and overwhelmingly reduce the stress on server-hackers.

But we don't have a C-compatible garbage collector...yet.

In the meantime, we're left with malloc() and free() for memory management. Now, one of the bad things you can do with malloc() and free() is to leak memory by mallocing memory and never freeing it.

PennMUSH provides a simple optional memory checker that can help in tracking down memory leaks. It works like this:

When you write a function that must allocate memory, pick a name that characterizes the function that that piece of memory serves. For example, the buffer allocated by safe_uncompress() for its results is called "safe_uncompress.buff".

After each allocation of memory for this purpose, put the following:

```
#ifdef MEM_CHECK
        add_check("name");
#endif
```

For example, the only place where a safe_uncompress.buff is allocated is in safe_uncompress().

After each free(), add a call to del_check for the appropriate name. For example, do_restart() calls safe_uncompress() to get the value of the STARTUP attribute, so it should contain the following code:

```
        free(buf);
#ifdef MEM_CHECK
        del_check("safe_uncompress.buff");
#endif
```

Now, if you compile with MEM_CHECK defined, the calls to add_check and del_check will be counted per type of memory allocated.

You can then check the memory usages by logging in as God and doing an @stats. However, this is only a static picture. To understand whether you are leaking memory, you need to check several times over a period. Which brings me to a discussion of strategies of testing.

Alpha testing happens when a server-hacker, perhaps with a few others, exercises a change. This is very useful, and can find a range of bugs, and this can be an invaluable way of precisely diagnosing a bug, particularly if you run from within a debugger. (I run my alpha test MUSH from within gdb by default.)

Alpha testing is extremely important, but a few testers can not uncover the same range of subtle bugs and quirks that a host of players can. This is why beta-testing has to happen. Beta-testing happens when some brave and courageous MUSH makes a backup (You do make backups, right?) and then begins to test a new version of the code.

When I'm beta-testing with MEM_CHECK, I usually write down all the counts once per day and look for trends. Some things like

"name" and the "attrib" entries should be persistent and reflect the composition of the database; these numbers will probably grow, but not too rapidly. Others, like "exec.buff" and "safe_uncompress.buff" should only be used temporarily, and if they grow out of control, you should be alert to the possibility of a leak.

Also, in my opinion, it is a very good thing for functions that allocate memory to mention the MEM_CHECK types of the memory that they allocate in a header comment, so that it's easier to use the right types for del_check later.

Moving to a New Site (Rhyanna)

Sometimes it becomes necessary for a MUSH to move to a new site. These are some of the common reasons:

- The site gets taken offline because of host problems.
- The site provider graduates / flunks out / otherwise loses their account.
- The MUSH starts taking up too much memory, disk space, or CPU time.
- The sysadmin of the site realizes that a MUSH is being run without permission. (This is far more common than it ought to be. **Always** get permission before running a MUSH.)

In my time, I have participated in two atrociously handled moves, and one move that seems to have been relatively successful. Based on that experience, here are my tips for moving.

1. Move as seamlessly as possible. Ideally, you would @wall about the move at the old site, shut it down, ftp the database to the new site (where you've already compiled and tested the code), and restart, with a total elapsed time of ten minutes or less. This is feasible, if you have planned ahead, compiled and tested the code at the new site, and so forth.

One atrocious move of my experience involved having two copies of a MUSH up at different sites for a period of more than a week, each with a MOTD that said the other MUSH was the 'official' MUSH. Another atrocious move involved the MUSH being down at one site for two weeks before coming up at the new site.

In the successful move, on the other hand, we had been testing the new site for anomalies for a few days, and we had declared a time when the database would move from the old site to the new site. Even though the move didn't happen at exactly the time we had declared, because the new site was down for maintenance, the move was a success, in large part because we came as close as possible to always having one and only one 'official' version of the MUSH around.

2) Leave a forwarder on the old site and port. This is extremely important.

Unfortunately, a large number of players don't pay close attention to the MOTD or to any other announcement ahead of time. No matter how much publicity you try to create for a move, there will be a large number of players who don't know anything about it until they try to connect to the old site and fail. This is why you need a port forwarder on the old site.

Almost all of the reasons why a legitimately running MUSH would need to move can be worked around for a port forwarder:

- A port forwarder is very small (portmsg.c compiles to 32K on a SunOS 4.1.1 system) and takes an insignificant amount of CPU time.
- If the host computer itself is going down, it is often possible to coax your friendly neighborhood sysadmin to remap the old name to a new computer. This is fairly standard practice anyway, to keep e-mail from bouncing and so forth. If the name gets remapped, it's often possible to run the port forwarder on the new host.
- If you are losing your account on the site, you can often coax a friendly sysadmin or another person with an account on that host to run a port forwarder for you, because it is such a small program with low cost in resources.
- If the MUSH is being evicted because of its drain on system resources, a port forwarder should be acceptable as a non-draining alternative. If even a port forwarder is a significant drain on the system, it begs the question of how you ever got a MUSH to run on a Vic-20 in the first place.

All of these require some planning and require that you be on relatively good terms with your sysadmins. (It's worthwhile to be on good terms with your sysadmins even if you're not running a MUSH.) From this, a corollary follows:

3) Move well before it is absolutely necessary to move.

Many times, it is possible to have ample forewarning before a move is going to be required. A machine may fail with increasing frequency before it goes down for the last time. Graduation is usually not a surprise, either. A kindly sysadmin can tell you ahead of time when a MUSH is starting to consume too many resources. (Note: If you ask to be notified if the MUSH starts to become a drain on system resources when you get permission to run your MUSH, you can save yourself a world of grief.)

You need to run a port forwarder after the MUSH moves. You may need to do some negotiation to get the port forwarder to run. Therefore, you should move as soon as you have a new site upon which you have compiled and tested the code. You should not wait until the last minute; the more critical your need to move, the more trouble you will have moving without losing many of your players.

The God who didn't make backups: A parable (Rhyanna)

Once upon a time, there was a God of a MUSH. It was a pleasant MUSH, of a medium size. It had just recently passed 5000 objects, and he was very proud. This God, though, was not taking backups of his database.

One day, then, calamity struck in the form of a socket programming project when required many people to use the MUSH's host and use many processes apiece. The MUSH crashed, and because there were no available processes, it truncated both the maildb and the indb to length 0.

That's the way it was told to me, but it was never very clear. But for ther purposes of the parable, it doesn't matter. Perhaps the disk head swung too close and plowed up slivers of aluminum from the spinning platter. Perhaps he ran afoul of a subtle and malignant bug with compression. The point is that he had not been making backups, and a stray accident had destroyed every copy of the database.

This would have spelled the death of many MUSHes. Many MUSHes that experience such a catastrophe never return at all. This one decided to try to rebuild, but even so, two of the wizards and far more of the players never returned, and the structure of the physical building and the roleplaying environment of the MUSH were never the same as they had been before.

The moral is left as an exercise for the reader.

A useful perl script: dated db-backups (Westley)

```
#!/usr/bin/perl
#
# dbchk.pl: external db file management for MU*
#
# Designed to be called by cron and make
# a backup of the db tagged with the date.
# Any given backup will be held for 7 days
# before being deleted by this program.
#
# UNTESTED! PLEASE REPORT BUGS TO alansz@mellers1.psych.berkeley.edu
#
# usage: dbchk [-d(data directory)] [-s(save directory)] [-f(filename)]
#
# By Michael Baker (Inigo, mbaker@cp.tybrin.com,
# Modified by Chris Hardy (Westley), modified slightly by Alan Schwartz


# Set usage for use in error messages.
$usage = "dbchk [-d(data directory)] [-s(save directory)] [-f(filename)]\n";

# require/include the getopts.pl program
# to handle switch type arguments.
require 'getopts.pl';

# Set default variables for db location
$directory = "~/game/data";
$savedir = "~/game/save";
$filename = "minimal.db.Z";

# Call Getopts (included earlier) to handle -d and -f switches
# the ':' following indicates that data follows the switch.
# Then process the data via if's.
&Getopts('d:f:s:') || die "$usage";

if ($opt_d) { $directory = $opt_d; }
if ($opt_f) { $filename = $opt_f; }
if ($opt_s) { $savedir = $opt_s; }

# Get the current time to stamp the backup with.
($sec, $min, $hour, $mday, $mon, $year, $wday, $yday, $isdst) = localtime(time());
$filestamp = join("",$mon,$mday,$year);

# Make the backup of the db
system("cp $directory/$filename $savedir/$filename.$filestamp");

# This section removes the saves over 7 days old.
opendir(DATED,$savedir) || die "Unable to remove old backups.\n";
```

```
@files = readdir(DATED);
closedir(DATED);
@files = grep(/$filename\./,@files);

while (shift(@files)) {
        if (-M > 7) {
                unlink($_) || warn "Unable to remove $_!\n";
        }
}
exit 0;
```

Resources

Sometimes you'll find that you need to ask a question and don't know who to turn to. Or perhaps you've found a bug in PennMUSH and want to report it (maybe you've even got a patch for it!). Or maybe you're just trying to track down some useful software.

Here's a list of where to find people and things to help.

Contents:

Where to Report Bugs/Problems

Bugs or problems with PennMUSH can always be reported to either Javelin <dunemush@pennmush.tinymush.org> or to the PennMUSH developers (Javelin, T. Alexander Popiel, and Ralph Melton). The developers have two different addresses. Bugs should be reported to <pennmush-bugs@pennmush.tinymush.org>, and development suggestions to <pennmush-developers@pennmush.tinymush.org>.

The PennMUSH mailing list

Javelin currently manages the list. To subscribe, send mail to listproc@mellers1.psych.berkeley.edu, and put the following in the message body:

```
subscribe pennmush Your name here
```

To mail to the list, mail to pennmush@mellers1.psych.berkeley.edu.

In order to keep the list relevant to its members, please *don't* send in problems or questions which relate to compiling the server, first startup, or anything which precedes having a running MUSH. Those sorts of problems are often site-specific, and should be directed to the Newsgroups The rec.games.mud.admin (r.g.m.a) newsgroup is devoted to issues involved in running a MU*. If you read the newsgroup, you'll find questions, answers, announcements, and discussions relating not only to MUSH, but to MOO, MUCK, LPmud, Diku, etc. The rec.games.mud.tiny newsgroup focuses exclusively on "tiny" genre MU*'s (MUSH, MUCK, MUSE, and TinyMUD), and its discussions are broader than simply administrative issues.

When posting a question to a newsgroup, be sure to include information about exactly what kind of code you're running, and on what kind of system (e.g. PennMUSH 1.50pl11 on Solaris 2.3). If you're posting to multiple newsgroups, don't post to each individually - cross-post by listing all the newsgroups on the Newsgroups: line of the posting.

If you don't know how to read and/or post news, ask your local system administrator. It's yet another marvelous resource (read: a new way to spend more hours on-line!)

FTP Sites

Here are some ftp sites with MUSH-useful stuff:

ftp.gnu.ai.mit.edu contains all the GNU utilities, including gcc, gdb, emacs, RCS, diff, patch, and others. Some are big to compile, but all are useful. This site can be overloaded, so you might also try gatekeeper.dec.com.

pennmush.tinymush.org is where PennMUSH releases after pl10 live, in /pub/PennMUSH/Source, as well as this guide and the scripts and programs discussed in the Appendix, in /pub/PennMUSH/Guide. ftp.cis.upenn.edu is where PennMUSH pl10 source is.

ftp.tinymush.org is where a lot of MUSH stuff is (/pub/mud/tinymush), including a MUSHcode archive, TinyMUSH code, etc.

ftp.tcp.com has other MUSH/MUD stuff, including robots (/pub/mud).

Other MUSHes

Often if you're facing a problem in the design or running of your MUSH, there's another MUSH which has solved it and may be willing to provide assistance. Some MUSHes will allow you to use systems originally coded for that MUSH with the approval and proper crediting of the original coder; others won't, but may still have helpful ideas. Amberyl maintains a hypertext list of MUSHes on the World-Wide Web at http://www.cis.upenn.edu/~lwl/muds.html

---

Appendix: Useful scripts and code

This appendix is a collected description of each of the scripts, programs, or MUSHcode mentioned in the rest of the guide. These scripts are all available by ftp from pennmush.tinymush.org in the /pub/DuneMUSH/Scripts directory. If you're reading this as hypertext, you also get links to let you download each script directly.

If you have a useful shell script or program that you'd like to add to the ftp site, you may deposit it in /pub/DuneMUSH/Scripts /incoming. Submitted scripts should be fully commented and not rely on local information (for example, people shouldn't have to change pathnames everywhere in the script - once at the top is fine.) Scripts should work with a clean distribution of PennMUSH, unpacked into the standard hierarchies. Perl, sh, csh, C programs, and patchfiles (context diffs) are all great. MUSHcode should be uploaded to the MUSHcode ftp site at ftp.tinymush.org.

Without further ado, then, here's the list:

DuneRestart
: A replacement for the stock restart script, written in csh. Supports remote restarts (via email), the mugshot program, logging of restarts, and keeping an extra db copy for safety.

backup
: A csh script which can be run from cron or in an infinite loop, which makes a backup copy of your db, and optionally will back up your source and db over the network to a remote site.

check
: A sh script which is meant to be run by cron to periodically check if your MUSH is up or down. If down, it mails you a message and restarts it. Submitted by Corey@FlunkyMush.

compose-simple
: A csh script to ease the updating of news.txt and events.txt files. The script allows you to create a subdirectory and split your news entries into many smaller files, which it will concatenate, mkindx, and install.

compose-tricky
: Like compose-simple, but supports automatic indexing if index-events is available, automatic updating of motd.txt, and automatic rumor maintainance if mkrumor is available.

dbchk.pl
: A perl script, meant to be run from cron, which makes daily dated backups of your db, and removes backups over 7 days old. Submitted by Westley@PrincessBrideMUSH.

index-events
: A perl script which builds an "& index" topic for news/events files, which lists a table of all the other topic names.

mkrumor
: A perl script which automatically selects files whose names indicate that they are within 4 months old, and builds a "& rumors" entry for news or events including the text of those files. Enables you to keep all your rumors, but only include the recent ones.

mugshot
: A perl script which sorts SUSPECTs out of a command.log, and appends each suspect player's log to their own personal file (by db#), for easy reading.

mushclock.msh
: MUSHcode for an object which performs actions at the top of each hour, like making general announcements.

portmsg.c
: A port announcer, like announce.c, but more reliable. You can compile this and run it when your MUSH is down, and player who connect will see a message and then be disconnected. Also handy if you change sites - you can run this on the old site to help players bamf to the new one.

procmailrc
: An example of a .procmailrc file for use with procmail, to filter incoming mail for special passwords which indicate an attempt to restart the MUSH via email.

setmotd.msh
: MUSHcode for an object to preserve @motd's across shutdowns by providing a place for Wizards to set motd entries and the 'setmotd' command to update the motd.

txtindex
: A perl script which reads in a list of keywords and some .txt files (e.g. help.txt, news.txt, and events.txt) and produces a new .txt file with keywords as topics, and a list of which help, news, and events topics have text containing that keyword as entries. Used with the post-pl10 INDEX_COMMAND define, which puts the 'index' command into the server for reading this new .txt file.

Webster
: A MUSH robot, written in perl, which implements the +spell spell-checking command (so players can check the spelling of @descs and things), the +rumor command (which enables players to submit rumors which are logged to a file on the MUSH account for easy perusal and editing), and, if your site has a websterd server, the +define command to return the definitions of words.

Last modified: Sep 21, 1996

Alan Schwartz (Javelin/Paul), alansz@mellers1.psych.berkeley.edu

Please leave your comments!