
Lmod Documentation

Release 6.0

Robert McLay

October 03, 2017

CONTENTS

1	PURPOSE	1
2	OVERVIEW	3
3	Introduction to Lmod	5
4	Installing Lmod	19
5	Advanced Topics	35
6	Topics yet to be written	47
7	Indices and tables	49

PURPOSE

Lmod is a Lua based module system that easily handles the MODULEPATH Hierarchical problem. Environment Modules provide a convenient way to dynamically change the users' environment through modulefiles. This includes easily adding or removing directories to the PATH environment variable. Modulefiles for Library packages provide environment variables that specify where the library and header files can be found.

OVERVIEW

This guide is written to explain what Environment Modules are and why they are very useful for both users and system administrators. Lmod is an implementation of Environment Modules, much of what is said here is true for any environment modules system but there are many features which are unique to Lmod.

Environment Modules provide a convenient way to dynamically change the users' environment through modulefiles. This includes easily adding or removing directories to the `PATH` environment variable.

A modulefile contains the necessary information to allow a user to run a particular application or provide access to a particular library. All of this can be done dynamically without logging out and back in. Modulefiles for applications modify the user's path to make access easy. Modulefiles for Library packages provide environment variables that specify where the library and header files can be found.

Packages can be loaded and unloaded cleanly through the module system. All the popular shells are supported: `bash`, `ksh`, `csh`, `tcsh`, `zsh`. Also available for `perl` and `python`.

It is also very easy to switch between different versions of a package or remove it.

INTRODUCTION TO LMOD

If you are new to Lmod then please read the User Guide and possibly the Frequently Asked Questions Guide. Users who wish to read about how to have their own personal modules should read the Advanced User Guide.

3.1 User Guide for Lmod

The guide here explains how to use modules. The User's tour of the module command covers the basic uses of modules. The other sections can be read at a later date as issues come up. The Advance user's guide is for users needing to create their own modulefiles.

3.1.1 User's Tour of the Module Command

The module command sets the appropriate environment variable independent of the user's shell. Typically the system will load a default set of modules. A user can list the modules loaded by:

```
$ module list
```

To find out what modules are available to be loaded a user can do:

```
$ module avail
```

To load packages a user simply does:

```
$ module load package1 package2 ...
```

To unload packages a user does:

```
$ module unload package1 package2 ...
```

A user might wish to change from one compiler to another:

```
$ module swap gcc intel
```

The above command is short for:

```
$ module unload gcc  
$ module load intel
```

If a module is not available then an error message is produced:

```
$ module load packageXYZ  
Warning: Failed to load: packageXYZ
```

It is possible to try to load a module with no error message if it does not exist:

```
$ module try-load packageXYZ
```

Modulefiles can contain help messages. To access a modulefile's help do:

```
$ module help packageName
```

To get a list of all the commands that module knows about do:

```
$ module help
```

The module avail command has search capabilities:

```
$ module avail cc
```

will list for any modulefile where the name contains the string “cc”.

Modulefiles can have a description section know as “whatis”. It is accessed by:

```
$ module whatis pmetis
pmetis/3.1    : Name: ParMETIS
pmetis/3.1    : Version: 3.1
pmetis/3.1    : Category: library, mathematics
pmetis/3.1    : Description: Parallel graph partitioning..
```

Finally, there is a keyword search tool:

```
$ module keyword word1 word2 ...
```

This will search any help or whatis description for the word(s) given on the command line.

Another way to search for modules is with the “module spider” command. This command searches the entire list of possible modules. The difference between “module avail” and “module spider” is explained in the “Module Hierarchy” and “Searching for Modules” section.

```
$ module spider
```

ml: A convenient tool

For those of you who can't type the *mdoule*, *moduel*, err *module* command correctly, Lmod has a tool for you. With **ml** you won't have to type the module command again. The two most common commands are *module list** and **module load <something>* and **ml** does both:

```
$ ml
```

means *module list*. And:

```
$ ml foo
```

means *module load foo* while:

```
$ ml -bar
```

means *module unload bar*. It won't come as a surprise that you can combine them:

```
$ ml foo -bar
```

means *module unload bar; module load foo*. You can do all the module commands:

```
$ ml spider
$ ml avail
$ ml show foo
```

If you ever have to load a module name *spider* you can do:

```
$ ml load spider
```

If you are ever forced to type the module command instead of **ml** then that is a bug and should be reported.

SAFETY FEATURES

(1): Users can only have one version active.

If a user does:

```
$ module avail xyz

----- /opt/apps/modulefiles -----
xyz/8.1   xyz/11.1 (D)   xyz/12.1

$ module load xyz
$ module load xyz/12.0
```

The first load command will load the 11.1 version of xyz. In the second load, the module command knows that the user already has xyz/11.1 loaded so it unloads that and then loads xyz/12.0. This protection is only available with Lmod.

(2) : Users can only load one compiler or MPI stack at a time.

Lmod provides an additional level of protection. If each of the compiler modulefiles add a line:

```
family("compiler")
```

Then Lmod will not load another compiler modulefile. Another benefit of the modulefile family directive is that an environment variable “LMOD_COMPILER_FAMILY” is assigned the name (and not the version). This can be useful specifying different options for different compilers. In the High Performance Computing (HPC) world, the message passing interface (MPI) libraries are important. The mpi modulefiles can contain a family(“MPI”) directive which will prevent users from loading more than one MPI implementation at a time. Also the environment variable “LMOD_FAMILY_MPI” is defined to the name of the mpi library.

Module Hierarchy

Libraries built with one compiler need to be linked with applications with the same compiler version. If sites are going to provide libraries, then there will be more than one version of the library, one for each compiler version. Therefore, whether it is the Boost library or an mpi library, there are multiple versions.

There are two main choices for system administrators. For the XYZ library compiled with either the UCC compiler or the GCC compiler, there could be the xyz-ucc modulefile and the xyz-gcc module file. This gets much more complicated when there are multiple versions of the XYZ library and different compilers. How does one label the various versions of the library and the compiler? Even if one makes sense of the version labeling, when a user changes compilers, the user will have to remember to unload the ucc and the xyz-ucc modulefiles when changing to gcc and xyz-gcc. If users have mismatched modules, their programs are going to fail in very mysterious ways.

A much saner strategy is to use a module hierarchy. Each compiler module adds to the MODULEPATH a compiler version modulefile directory. Only modulefiles that exist in that directory are packages that have been built with that compiler. When a user loads a particular compiler, that user only sees modulefile(s) that are valid for that compiler.

Similarly, applications that use libraries depending on MPI implementations must be built with the same compiler - MPI pairing. This leads to modulefile hierarchy. Therefore, as users start with the minimum set of loaded modules, all they will see are compilers, not any of the packages that depend on a compiler. Once they load a compiler they will see the modules that depend on that compiler. After choosing an MPI implementation, the modules that depend on that compiler-MPI pairing will be available. One of the nice features of Lmod is that it handles the hierarchy easily. If a user swaps compilers, then Lmod automatically unloads any modules that depends on the old compiler and reloads those modules that are dependent on the new compiler.

```
$ module list

1) gcc/4.4.5 2) boost/1.45.0

$ module swap gcc ucc

Due to MODULEPATH changes the follow modules have been reloaded: 1) boost
```

If a modulefile is not available with the new compiler, then the module is marked as inactive. Every time MODULEPATH changes, Lmod attempts to reload any inactive modules.

Searching For Modules

When a user enters:

```
$ module avail
```

Lmod reports only the modules that are in the current MODULEPATH. Those are the only modules that the user can load. If there is a modulefile hierarchy, then a package the user wants may be available but not with the current compiler version. Lmod offers a new command:

```
$ module spider
```

which lists all possible modules and not just the modules that can be seen in the current MODULEPATH. This command has three modes. The first mode is:

```
$ module spider

lmod: lmod/lmod
Lmod: An Environment Module System

ucc: ucc/11.1, uuc/12.0, ...
Ucc: the ultimate compiler collection

xyz: xyz/0.19, xyz/0.20, xyz/0.31
xyz: Solves any x or y or z problem.
```

This is a compact listing of all the possible modules on the system. The second mode describes a particular module:

```
$ module spider ucc
-----
ucc:
-----

Description:
Ucc: the ultimate compiler collection
```

```

Versions:
ucc/11.1
ucc/12.0

```

The third mode reports on a particular module version and where it can be found:

```

$ module spider parmetis/3.1.1
-----
parmetis: parmetis/3.1.1
-----
Description:
Parallel graph partitioning and fill-reduction matrix ordering routines

This module can be loaded through the following modules:
ucc/12.0, openmpi/1.4.3
ucc/11.1, openmpi/1.4.3
gcc/4.4.5, openmpi/1.4.3

Help:
The parmetis module defines the following environment variables: ...
The module parmetis/3.1.1 has been compiled by three different versions of the ucc compiler and one M

```

Controlling Modules During Login

Normally when a user logs in, there are a standard set of modules that are automatically loaded. Users can override and add to this standard set in two ways. The first is adding module commands to their personal startup files. The second way is through the “module save” command.

To add module commands to users’ startup scripts requires a few steps. Bash users can put the module commands in either their `~/.profile` file or their `~/.bashrc` file. It is simplest to place the following in their `~/.profile` file:

```

if [ -f ~/.bashrc ]; then
    . ~/.bashrc
fi

```

and place the following in their `~/.bashrc` file:

```

if [ -z "$BASHRC_READ" ]; then
    export BASHRC_READ=1
    # Place any module commands here
    # module load git
fi

```

By wrapping the module command in an if test, the module commands need only be read in once. Any sub-shell will inherit the `PATH` and other environment variables automatically. On login shells the `~/.profile` file is read which, in the above setup, causes the `~/.bashrc` file to be read. On interactive non-login shells, the `~/.bashrc` file is read instead. Obviously, having this setup means that module commands need only be added in one file and not two.

Csh users need only specify the module commands in their `~/.cshrc` file as that file is always sourced:

```

if ( ! $?CSHRC_READ ) then
    setenv CSHRC_READ 1
    # Place any module command here
    # module load git
endif

```

User defined initial list of login modules:

Assuming that the system administrators have installed Lmod correctly, there is a second way which is much easier to setup. A user logs in with the standard modules loaded. Then the user modifies the default setup through the standard module commands:

```
$ module unload XYZ
$ module swap gcc ucc
$ module load git
```

Once users have the desired modules load then they issue:

```
$ module save
```

This creates a file called `~/.lmod.d/default` which has the list of desired modules. Once this is setup a user can issue:

```
$ module restore
```

and only the desired modules will be loaded during login.

Users can have as many collections as they like. They can save to a named collection with:

```
$ module save <collection_name>
```

and restore that named collection with:

```
$ module restore <collection_name>
```

Finally a user can print the contents of a collection with:

```
$ module describe <collection_name>
```

3.2 An Introduction to Writing Modulefiles

This is a different kind of introduction to Lmod. Here we will remind you what Lmod is doing to change the environment via modulefiles. Then we will start with the four functions that are typically needed for any modulefile. From there we will talk about intermediate level module functions when things get more complicated. Finally we will discuss the advanced module functions to flexibly control your site via modules. All the Lua module functions available are described at [Lua Modulefile Functions](#). This discussion shows how they can be used.

3.2.1 A Reminder of what Lmod is doing

All Lmod is doing is changing the environment. Suppose you want to use the “`ddt`” debugger installed on your system which is made available to you via the module. If you try to execute `ddt` without the module loaded you get:

```
$ ddt
bash: command not found: ddt

$ module load ddt
$ ddt
```

After the `ddt` module is loaded, executing **`ddt`** now works. Let’s remind ourselves why this works. If you try checking the environment before loading the `ddt` modulefile:

```
$ env | grep -i ddt
$ module load ddt
$ env | grep -i ddt
```

```
DDTPATH=/opt/apps/ddt/5.0.1/bin
LD_LIBRARY_PATH=/opt/apps/ddt/5.0.1/lib:...
PATH=/opt/apps/ddt/5.0.1/bin:...
```

```
$ module unload ddt
$ env | grep -i ddt
$
```

The first time we check the environment we find that there is no **ddt** stored there. But the second time there we see that the **PATH** and **LD_LIBRARY_PATH** have been modified. Note that we have shortened the path-like variables to show the important changes. There are also several environment variables which have been set. After unloading the module all the references for **ddt** have been removed. We can see what the modulefile looks like by doing:

```
$ module show ddt

help([[
For detailed instructions, go to:
  https://...
]])
whatis("Version: 5.0.1")
whatis("Keywords: System, Utility")
whatis("URL: http://content.allinea.com/downloads/userguide.pdf")
whatis("Description: Parallel, graphical, symbolic debugger")

setenv(      "DDTPATH",      "/opt/apps/ddt/5.0.1/bin")
prepend_path( "PATH",      "/opt/apps/ddt/5.0.1/bin")
prepend_path( "LD_LIBRARY_PATH", "/opt/apps/ddt/5.0.1/lib")
```

Modulefiles are state the actions that need to happen when loading. For example the above modulefile uses **setenv** and **prepend_path** to set environment variables and prepend to the **PATH**. If the above modulefile is unloaded then the **setenv** actually unsets the environment variable. The **prepend_path** removes the element from the **PATH** variable. That is unload causes the functions to be reversed.

3.2.2 Basic Modulefiles

There are two main module functions required, namely **setenv** and **prepend_path**; and two functions to provide documentation **help** and **whatis**. The modulefile for **ddt** shown above contains all the basics required to create one. Suppose you are writing this module file for **ddt** version 5.0.1 and you are placing it in the standard location for your site, namely */apps/modulefiles* and this directory is already in **MODULEPATH**. Then in the directory */apps/modulefiles/ddt* you create a file called *5.0.1.lua* which contains the modulefile shown above.

This is the typical way of setting a modulefile up. Namely the package name is the name of the directory, *ddt*, and version name, *5.0.1* is the name of the file with the *.lua* extension added. We add the lua extension to all modulefile written in Lua. All modulefiles without the lua extension are assumed to be written in TCL.

If another version of **ddt** becomes available, say *5.1.2*, we create another file called *5.1.2.lua* to become the new modulefile for the new version of **ddt**.

When a user does *module help ddt*, the arguments to the **help** function are written out to the user. The **whatis** function provides a way to describe the function of the application or library. This data can be used by search tools such as **module keyword search_words**. Here at TACC we also use that data to provide search capability via the web interface to modules we provide.

3.2.3 Intermediate Level Modulefiles

The four basic functions describe above is all that is necessary for the majority of modulefiles for application and libraries. The intermediate level is designed to describe some situations that come up as you need to provide more than just packages modulefile but need to set up a system.

Meta Modules

Some sites create a single module to load a default set of modules for all users to start from. This is typically called a meta module because it loads other modules. As an example of that, we here at TACC have created the TACC module to provide a default compiler, mpi stack and other modules:

```
help([[
The TACC modulefile defines ...
]])

-- 1 --
if (os.getenv("USER") ~= "root") then
    append_path("PATH", ".")
end

-- 2 --
load("intel", "mvapich2")

-- 3 --
try_load("xalt")

-- 4 --
-- Environment change - assume single threaded.
if (mode() == "load" and os.getenv("OMP_NUM_THREADS") == nil) then
    setenv("OMP_NUM_THREADS", "1")
end
```

This modulefile shows the use of four new functions. The first one is **append_path**. This function is similar to **prepend_path** except that the value is placed at the end of the path-like variable instead of the beginning. We add `.` to our user's path at the end, except for root. This way our new users don't get surprised with some programs in their current directory that do not run. We used the **os.getenv** function built-in to Lua to get the value of environment variable "USER".

The second function is **load**, this function loads the modulefiles specified. This function takes one or more names. Here we are specifying a default compiler and mpi stack. The third function is **try_load**, which is similar to **load** except that there is no error reported if the module can't be found.

The fourth block of code shows how we set **OMP_NUM_THREADS**. We wish to set **OMP_NUM_THREADS** to have a default value of 1, but only if the value hasn't already been set and we only want to do this when the module is being loaded and not at any other time. So when this module is loaded for the first time **mode()** will return "load" and **OMP_NUM_THREADS** won't have a value. The **setenv** will set it to 1. If the TACC module is unloaded, the **mode()** will be "unload" so the if test will be false and therefore the **setenv** will not be reversed. If the user changes **OMP_NUM_THREADS** and reloads the TACC modulefile, their value won't change because **os.getenv("OMP_NUM_THREADS")** will return a non-nil value, therefore the **setenv** command won't run. Now this may not be the best way to handle this. It might be better to set **OMP_NUM_THREADS** in a file that is sourced in `/etc/profile.d/` and have all the important properties. Namely that there will be a default value that the user can change. However this example shows how to do something tricky in a modulefile.

Typically meta modules are a single file and not versioned. So the TACC modulefile can be found at `/apps/modulefiles/TACC.lua`. There is no requirement that this will be this way but it has worked well in practice.

Modules with dependencies

Suppose that you have a package which needs libraries or an application. For example the octave application needs gnuplot. Let's assume that you have a separate applications for both. Inside the octave module you can do:

```
prereq("gnuplot")
...
```

So if you execute:

```
$ module unload gnuplot
$ module load octave
$ module load gnuplot octave
$ module unload octave
```

The second module command will fail, but the third one will succeed because we have met the prerequisites. The advantage of using `prereq` is after fourth module command is executed, the gnuplot module will be loaded.

This can be contrasted with including the load of gnuplot in the octave modulefile:

```
load("gnuplot")
...
```

This simplifies the loading of the octave module. The trouble is that when a user does the following:

```
$ module load    gnuplot
$ module load    octave
$ module unload octave
```

is that after unloading *octave*, the *gnuplot* module is also unloaded. It seems better to either use the **prereq** function shown above or use the **always_load** function in the octave module:

```
always_load("gnuplot")
...
```

Then when a user does:

```
$ module load    gnuplot
$ module load    octave
$ module unload octave
```

The *gnuplot* module will still be loaded after unloading *octave*. This will lead to the least confusion to users.

Fancy dependencies

Sometimes an application may depend on another application but it has to be a certain version or newer. Lmod can support this with the **atleast** modifier to both **load**, **always_load** or **prereq**. For example:

```
-- Use either the always_load or prereq but not both:

prereq(    atleast("gnuplot", "5.0"))
always_load(atleast("gnuplot", "5.0"))
```

The **atleast** modifier to **prereq** or **always_load** will succeed if the version of gnuplot is 5.0 or greater.

Assigning Properties

Modules can have properties that will be displayed in a *module list* or *module avail*. Properties can be anything but they must be specified in the *lmodrc.lua* file. You are free to add to the list. For example, to specify a module to be

experimental all you need to do is:

```
add_property("state", "experimental")
```

Any properties you set must be defined in the **lmodrc.lua** file. In the source tree the properties are in `init/lmodrc.lua`. A more detailed discussion of the `lmodrc.lua` file can be found at `lmodrc-label`

Pushenv

Lmod allows you to save the state in a stack hidden in the environment. So if you want to set the **CC** environment variable to contain the name of the compiler.:

```
-- gcc --
pushenv("CC", "gcc")

-- mpich --
pushenv("CC", "mpicc")
```

If the user executes the following:

#		SETENV	PUSHENV
\$ export CC=cc;	echo \$CC	# -> CC=cc	CC=cc
\$ module load gcc;	echo \$CC	# -> CC=gcc	CC=gcc
\$ module load mpich;	echo \$CC	# -> CC=mpicc	CC=mpicc
\$ module unload mpich;	echo \$CC	# -> CC is unset	CC=gcc
\$ module unload gcc;	echo \$CC	# -> CC is unset	CC=cc

We see that the value of **CC** is maintained as a stack variable when we use *pushenv* but not when we use *setenv*.

Setting aliases and shell functions

Sometimes you want to set an alias as part of a module. For example the visit program requires the version to be specified when running it. So for version 2.9 of visit, the alias is set:

```
set_alias("visit", "visit -v 2.9")
```

Whether this will expand correctly depends on the shell. While C-shell allows argument expansion in aliases, Bash and Zsh do not. Bash and Zsh use shell functions instead. For example the `ml` shell function can be set like this:

```
local bashStr = 'eval ${$LMOD_DIR/ml_cmd "$@"}'
local cshStr  = "eval `${$LMOD_DIR/ml_cmd $*}`"
set_shell_function("ml", bashStr, cshStr)
```

3.3 Frequently Asked Questions

How does the module command work?

We know that the child program inherits the parents' environment but not the other way around. So it is very surprising that a command can change the current shell's environment. The trick here is that the module command is a two part process. The module shell function in bash is:

```
$ type module
module() { eval ${$LMOD_CMD bash "$@"} }
```

Where `$LMOD_CMD` points to your lmod command (say `/apps/lmod/lmod/libexec/lmod`). So if you have a module file (`foo/1.0`) that contains:

```
setenv("FOO", "BAR")
```

then “\$LMOD_CMD bash load foo/1.0” produces:

```
export FOO=BAR
...
```

The eval command read that output from stdout and changes the current shell’s environment. Any text written to stderr bypasses the eval and written to the terminal.

What are the environment variables _ModuleTable001_, _ModuleTable002_, etc doing it in the environment?

The module command remembers its state in the environment through these variables. The way Lmod does it is through a Lua table called ModuleTable:

```
ModuleTable = {
  mT = {
    git = { ... }
  }
}
```

This table contains quotes and commas and must be store in environment. To prevent problems the various shells, the table is encoded into base64 and split into blocks of 256 characters. These variable are decoded at the start of Lmod. You can see what the module table contains with:

```
$ module --mt
```

How does one debug a modulefile?

There are two methods. Method 1: If you are writing a Lua modulefile then you can write messages to stderr with and run the module command normally:

```
local a = "text"
io.stderr:write("Message ",a,"\n")
```

Method 2: Take the output directly from Lmod. You can put print() statements in your modulefile and do:

```
$ $LMOD_CMD bash load *modulefile*
```

Why doesn’t %module avail |& grep ucc work under tcsh and works under bash?

It is a bug in the way tcsh handles evals. This works:

```
% (module avail) |& grep ucc
```

However, in all shells it is better to use:

```
% module avail ucc
```

instead as this will only output modules that have “ucc” in their name.

Why are messages printed to standard error and not standard out?

The module command is an alias under tcsh and a shell routine under all other shells. There is an lmod command which writes out commands such as export FOO=”bar and baz” and messages are written to standard error. The text written to standard out is evaluated so that the text strings make changes to the current environment.

Can I disable the pager output?

Yes, you can. Just set the environment variable LMOD_PAGER to **none**.

Can I force the output of list, avail and spider to go to stdout instead of stderr?

Bash and Zsh user can set the environment variable `LMOD_REDIRECT` to **yes**. Sites can configure Lmod to work this way by default. However, no matter how Lmod is set-up, this will not work with tcsh/csh due to limitations of this shell.

Can I ignore the spider cache files when doing `module avail`?

Yes you can:

```
$ module --ignore_cache avail
```

or you can set:

```
$ export LMOD_IGNORE_CACHE=1
```

to make Lmod ignore caches as long as the variable is set.

I have created a module and “module avail” can’t find it. What do I do?

Assuming that the modulefile is in `MODULEPATH` then you have an out-of-date cache. Try running:

```
$ module --ignore_cache avail
```

If this does find it then you might have an old personal spider cache. To clear it do:

```
$ rm -rf ~/.lmod.d/.cache
```

If “module avail” doesn’t find it now, then the system spider cache is out-of-date. Please ask your system administrator to update the cache. If you are the system administrator then please read [System Spider Cache](#) and `user-spider-cache-label`

Why doesn’t the module command work in shell scripts?

It will if the following steps are taken. First the script must be a bash script and not a shell script, so start the script with `#!/bin/bash`. The second is that the environment variable `BASH_ENV` must point to a file which defines the module command. The simplest way is having `BASH_ENV` point to `/opt/apps/lmod/lmod/init/bash` or wherever this file is located on your system. This is done by the standard install. Finally Lmod exports the module command for Bash shell users.

How do I use the initializing shell script that comes with this application with Lmod?

The short answer is you don’t. Among the many problems is that there is no way to unload that shell script. If the script is simple you can read it through and create a modulefile. To simplify this task, Lmod provides the `sh_to_modulefile` script to convert shell scripts to modulefiles.

Why is the output of `module avail` not filling the width of the terminal?

If the output of `module avail` is 80 characters wide, then Lmod can’t find the width of the terminal and instead uses the default size (80). If you do `module --config`, you’ll see a line:

```
Active lua-term true
```

If it says **false** instead then lua-term is not installed. One way this happens is to build Lmod on one computer system that has a system lua-term installed and the package on another where lua-term isn’t installed on the system.

Why does isn’t the module defined when using the **screen** program?

The screen program starts a non-login interactive shell. The Bash shell startup doesn’t start sourcing `/etc/profile` and therefore the `/etc/profile.d/*.sh` scripts for non-login interactive shells. You can patch bash and fix `/etc/bashrc` (see [Issues with Bash](#) for a solution) or you can fix your `~/.bashrc` to source `/etc/profile.d/*.sh`

You may be better off using **tmux** instead. It starts a login shell.

Why does `LD_LIBRARY_PATH` get cleared when using the **screen** program?

The screen program is a **guid** program. That means it runs as the group of the program and not the group associated with the user. For security reasons all of these programs clear `LD_LIBRARY_PATH`.

You may be better off using **tmux** instead. It is a regular program.

3.4 Advanced User Guide for Personal Modulefiles

This advanced guide is for users wishing to create modulefiles for their own software. The reasons are simple:

1. Install newer version of open source software than is currently available.
2. Easily change version of applications or libraries under their own development.
3. Better documentation for what software is available.

You can create new version of some software and place it in your personal `PATH` and forget about it. At least when it is in a module, it will be listed in the loaded modules it will also appear in the list of available software via `module avail`

3.4.1 User Created Modules

Users can create their own modules. The first step is to add to the module path:

```
$ module use /path/to/personal/modulefiles
```

This will prepend `/path/to/personal/modulefiles` to the `MODULEPATH` environment variable. This means that any modulefiles defined here will be used instead of the system modules.

Suppose that the user creates a directory called `$HOME/modulefiles` and he wants a personal copy of the “git” package and he does the usual “tar, configure, make, make install” steps:

```
$ wget https://www.kernel.org/pub/software/scm/git/git-2.6.2.tar.gz
$ tar xf git-2.6.2.tar.gz
$ cd git-2.6.2
$ ./configure --prefix=$HOME/pkg/git/2.6.2
$ make
$ make install
```

This document has assumed that 2.6.2 is the current version of git, it will need to be replaced with the current version. To create a modulefile for git one does:

```
$ cd ~/modulefiles
$ mkdir git
$ cd git
$ cat > 2.6.2.lua
local home      = os.getenv("HOME")
local version   = myModuleVersion()
local pkgName   = myModuleName()
local pkg       = pathJoin(home, "pkg", pkgName, version, "bin")
prepend_path("PATH", pkg)
^D
```

Starting first from the name: `git/2.6.2.lua`, modulefiles with the `.lua` extension are assumed to be written in lua and files without are assumed to be written in TCL. This modulefile for git adds `~/pkg/git/2.6.2/bin` to the user’s path so that the personal version of git can be found. Note that the use of the functions **myModuleName()** and **myModuleVersion()** allows the module to be generic and not hard-wired to a particular module file. We have used

the `cat` command to quickly create this lua modulefile. Obviously, this file can easily be created by your favorite editor (emacs, vi, nano, ...).

Starting first from the name: `git/2.6.2.lua`, Modulefiles with the `.lua` extension are assumed to be written in lua and files without are assumed to be written in TCL.

The first line reads the user's HOME directory from the environment. The second line uses the "pathJoin" function provided from Lmod. It joins strings together with the appropriate number of "/". The last line calls the "prepend_path" function to add the path to git to the user's path.

Finally the user can do:

```
$ module use $HOME/modulefiles
$ module load git
$ type git
~/pkg/git/2.6.2/bin/git
```

For git to be available on future logins, users need to add the following to their startup scripts or a saved collection.

```
$ module use $HOME/modulefiles
$ module load git
```

The modulefiles can be stored in different directories. There is an environment variable `MODULEPATH` which controls that. Modulefiles that are listed in an earlier directory are found before ones in later directories. This is similar to command searching in the `PATH` variable. There can be several versions of a command. The first one found in the `PATH` is used.

3.4.2 Finding Modules With Same Name

Suppose the user has created a "git" module to provide the latest available. At a later date, the system administrators add a newer version of "git"

```
$ module avail git
----- /home/user/modulefiles -----
git/2.6.2

----- /opt/apps/modulefiles -----
git/1.7.4  git/2.0.1  git/3.5.4 (D)

$ module load git
```

The load command will load `git/3.5.4` because it is the highest version.

If a user wishes to make their own version of git the default module, they will have to mark it as a default. Marking a module as a default is discussed in section [Marked a Version as Default](#)

INSTALLING LMOD

Anyone wishing to install Lmod on a personal computer or for a system should read the Installation Guide as well as the Transitioning to Lmod Guide. The rest of the guides can be read as needed.

4.1 Installing Lua and Lmod

Environment modules simplify customizing the users's shell environment and it can be done dynamically. Users load modules as they see fit. It is completely under their control. Environment Modules or simply modules provide a simple contract or interface between the system administrators and users. System administrators provide modules and users get to choose which to load.

There have been environment module systems for quite a while. See <http://modules.sourceforge.net/> for a TCL based module system and see <http://www.lysator.liu.se/cmod> for another module system. Here we describe Lmod, which is a completely new module system written in Lua. For those who have used modules before, Lmod automatically reads TCL modulefiles. Lmod has some important features over other module system, namely a built-in solution to hierarchical modulefiles and provides additional safety features to users as described in the User Guide.

The hierarchical modulefiles are used to solve the issue of system pre-built libraries. User applications using these libraries must be built with the same compiler as the libraries. If a site provides more than one compiler, then for each compiler version there will be separate versions of the libraries. Lmod provides built-in control making sure that compilers and pre-built libraries stay matched. The rest of the pages here describe how to install Lmod, how to provide the module command to users during the login process and some discussion on how to install optional software and the associated modules.

The goal of installing Lmod is when completed, any user will have the module command defined and a preset list of modules will be loaded. The module command should work without modifying the users startup files (`~/.bashrc`, `~/.profile`, `~/.cshrc`, or `~/.zshenv`). The module command should be available for login shells, interactive shells, and non-interactive shells. The command `ssh YOUR_HOST module list` should work. This will require some understanding of the system startup procedure for various shells which is covered here.

4.1.1 Installing Lua

In this document, it is assumed that all optional software is going to be installed in `/opt/apps`. The installation of Lmod requires installing lua as well. On some system, it is possible to install Lmod directly with your package manager. It is available with recent fedora, debian and ubuntu distributions.

Install lua-X.Y.Z.tar.gz

One choice is to install the lua-X.Y.Z.tar.gz file. This tar ball contains lua and the required libraries. This can be downloaded from <https://sourceforge.net/projects/lmod/files/>:

```
$ wget https://sourceforge.net/projects/lmod/files/luarocks-5.1.4.5.tar.gz
```

The current version is 5.1.4.5 but it may change in the future. This can be installed using the following commands:

```
$ tar xf lua-X.Y.Z.tar.gz
$ cd lua-X.Y.Z
$ ./configure --prefix=/opt/apps/lua/X.Y.Z
$ make; make install
$ cd /opt/apps/lua; ln -s X.Y.Z lua
$ mkdir /usr/local/bin; ln -s /opt/apps/lua/lua/bin/lua /usr/local/bin
```

The last command is optional, but you will have to somehow put the `lua` command in your path. Also obviously, please replace `X.Y.Z` with the actual version (say 5.1.4.5)

Using Your Package Manager

You can use your package manager for your OS to install Lua. You will also need the matching packages: `lua` Filesystem (lfs) and `luaposix`. On Ubuntu Linux, the following packages will work:

```
liblua5.1-0
liblua5.1-0-dev
liblua5.1-filesystem-dev
liblua5.1-filesystem0
liblua5.1-posix-dev
liblua5.1-posix0
lua5.1
```

Note; Centos may require looking the EPEL repo. At TACC we install the following rpms:

```
$ rpm -qa | grep lua

lua-posix-5.1.7-1.el6.x86_64
lua-5.1.4-4.1.el6.x86_64
lua-filesystem-1.4.2-1.el6.x86_64
lua-devel-5.1.4-4.1.el6.x86_64
```

You will also need the `libtcl` and `tcl` packages as well.

Using Luarocks

If you have installed `lua` but still need `luafilesystem` and `luaposix`, you can install the `luarocks` program from your package manager or directly from <https://luarocks.org/>. The `luarocks` programs can install many lua packages including the ones required for Lmod.

```
$ luarocks install luaposix; luarocks install luafilesystem
```

Now you have to make the lua packages installed by luarocks to be known by lua. On our Centos system, Lua knows about the following for `*.lua` files:

```
$ lua -e 'print(package.path)'
./?.lua;/usr/share/lua/5.1/?.lua;/usr/share/lua/5.1/?/init.lua;/usr/lib64/lua/5.1/?.lua;/usr/lib64/l
```

and the following for shared libraries:

```
$ lua -e 'print(package.cpath)'
./?.so;/usr/lib64/lua/5.1/?.so;/usr/lib64/lua/5.1/loadall.so;
```

Assuming that luarocks has installed things in its default location (/usr/local/...) then you'll need to do:

```
LUAROCKS_PREFIX=/usr/local
export LUA_PATH="$LUAROCKS_PREFIX/share/lua/5.1/?.lua;$LUAROCKS_PREFIX/share/lua/5.1/?.init.lua;;"
export LUA_CPATH="$LUAROCKS_PREFIX/lib/lua/5.1/?.so;;"
```

Please change LUAROCKS_PREFIX to match your site. The exporting of LUA_PATH and LUA_CPATH must be done before any module commands. It is very important that the trailing semicolon are there. They are replaced by the built-in system path.

4.1.2 Why does Lmod install differently?

Lmod automatically creates a version directory for itself. So, for example, if the installation prefix is set to /apps, and the current version is X.Y.Z, installation will create /apps/lmod and /apps/lmod/X.Y.Z. This way of configuring is different from most packages. There are two reasons for this:

1. Lmod is designed to have just one version of it running at one time. Users will not be switching version during the course of their interaction in a shell.
2. By making the symbolic link the startup scripts in /etc/profile.d do not have to change. They just refer to /apps/lmod/lmod/... and not /apps/lmod/X.Y.Z/...

4.1.3 Installing Lmod

Lmod has a large number of configuration options. They are discussed in the Configuring Lmod Guide. This section here will describe how to get Lmod installed quickly by using the defaults:

Note: If you have a large number of modulefiles or a slow parallel filesystem please read the Configure Lmod Guide on how to set-up the spider caching system. This will greatly speed up module avail and module spider

To install Lmod, you'll want to carefully read the following. If you want Lmod version X.Y installed in /opt/apps/lmod/X.Y, just do:

```
$ ./configure --prefix=/opt/apps
$ make install
```

The installation will also create a link to /apps/lmod/lmod. The symbolic link is created to ease upgrades to Lmod itself, as numbered versions can be installed side-by-side, testing can be done on the new version, and when all is ready, only the symbolic link needs changing.

To create such a testing installation, you can use:

```
$ make pre-install
```

which does everything but create the symbolic link.

In the init directory of the source code, there are profile.in and cshrc.in templates. During the installation phase, the path to lua is added and profile and cshrc are written to the /apps/lmod/lmod/init directory. These files are created assuming that your modulefiles are going to be located in /apps/modulefiles/\$LMOD_sys and /apps/modulefiles/Core, where \$LMOD_sys is what the command "uname" reports, (e.g., Linux, Darwin). The layout of modulefiles is discussed later.

Note: Obviously you will want to modify the profile.in and cshrc.in files to suit your system.

The profile file is Lmod initialization script for the bash, and zsh shells and cshrc file is for tcsh and csh shells. Please copy or link the profile and cshrc files to `/etc/profile.d`

```
$ ln -s /opt/apps/lmod/lmod/init/profile /etc/profile.d/z00_lmod.sh
$ ln -s /opt/apps/lmod/lmod/init/cshrc /etc/profile.d/z00_lmod.csh
```

To test the setup, you just need to login as a user. The module command should be set and `MODULEPATH` should be defined. Bash or Zsh users should see something like:

```
$ type module
module ()
{
    eval `${LMOD_CMD} bash $*`
}

$ echo $LMOD_CMD
/opt/apps/lmod/lmod/libexec/lmod

$ echo $MODULEPATH
/opt/apps/modulefiles/Linux:/opt/apps/modulefiles/Core
```

Similar for csh users:

```
% which module
module: alias to eval `/opt/apps/lmod/lmod/libexec/lmod tcsh !*`

% echo $MODULEPATH
/opt/apps/modulefiles/Linux:/opt/apps/modulefiles/Core
```

If you do not see the module alias then please read the next section.

4.1.4 Integrating module Into Users' Shells

Bash:

On login, the bash shell first reads `/etc/profile`, and if `profiles.d` is activated, that in turn should source all the `*.sh` files in `/etc/profile.d` with something like:

```
if [ -d /etc/profile.d ]; then
  for i in /etc/profile.d/*.sh; do
    if [ -r $i ]; then
      . $i
    fi
  done
fi
```

Similarly, the system `BASHRC` file should source all the `*.sh` files in `/etc/profile.d` as well. Here is where things can get complicated. See the next section for details.

Bash Shell Scripts:

Bash shell scripts do not source any system or user files before starting execution. Instead it looks for the environment variable `BASH_ENV`. It treats the contents as a filename and sources it before starting a bash script.

Bash Script Note:

It is important to remember that all bash scripts should start with:

```
#!/bin/bash
```

Starting with:

```
#!/bin/sh
```

and sh is linked to bash won't define the module command. Bash will run those scripts in shell emulation mode and it doesn't source the file that BASH_ENV points to.

Csh:

Csh users have an easier time with the module command setup. The system cshrc file is always sourced on every invocation of the shell. The system cshrc file is typically called: `/etc/csh.cshrc`. This file should source all the *.csh files in `/etc/profile.d`:

```
if ( -d /etc/profile.d ) then
  set nonomatch
  foreach i (/etc/profile.d/*.csh)
    source $i
  end
  unset nonomatch
endif
```

Zsh:

Zsh users have an easy time with the module command setup as well. The system zshenv file is sourced on all shell invocations. This system file can be in a number of places but is typically in `/etc/zshenv` or `/etc/zsh/zshenv` and should have:

```
if [ -d /etc/profile.d ]; then
  setopt no_nomatch
  for i in /etc/profile.d/*.sh; do
    if [ -r $i ]; then
      . $i
    fi
  done
fi
```

4.1.5 Issues with Bash

Interactive Non-login shells

The Bash startup procedure for interactive non-login shells is complicated and varies between Operating Systems. In particular, Redhat & Centos distributions of Linux as well as Mac OS X have no system bashrc read during startup where as Debian based distributions do source a system. One easy way to tell how bash is set up is to execute the following:

```
$ strings `type -p bash` | grep bashrc
```

If the entire results of the command is:

```
~/.bashrc
```

then you know that your bash shell doesn't source a system BASHRC file.

If you want to have the same behavior between both interactive shell (login or non) and your system doesn't source a system bashrc, then you have two choices:

1. Patch bash so that it does source a system bashrc. See `contrib/bash_patch` for details on how to do that.
2. Expect all of your bash users to have the following in their `~/ .bashrc`

```
if [ -f /etc/bashrc ]; then
    . /etc/bashrc
fi
```

As a side note, we at TACC patched bash for a different reason which may apply to your site. When an MPI job starts, it logs into each node with an interactive non-login shell. When we had no system bashrc file, many of our fortran 90 programs failed because they required `ulimit -s unlimited` which makes the stack size unlimited. By patching bash, we could guarantee that it was set by the system on each node.

You may have to also change the `/etc/bashrc` (or `/etc/bash.bashrc`) file so that it sources `/etc/profile.d/*.sh` for non-login shells.

Bash Shell Scripts

Bash shell scripts, unlike Csh or Zsh scripts, do not source any system or user files. Instead, if the environment variable, `BASH_ENV` is set and points to a file then this file is sourced before the start of bash script. So by default Lmod sets `BASH_ENV` to point to the bash script which defines the module command.

It may seem counterintuitive but Csh and Zsh users running bash shell scripts will want `BASH_ENV` set so that the module command will work in their bash scripts.

A bash script is one that starts as the very first line:

```
#!/bin/bash
```

A script that has nothing special or starts with:

```
#!/bin/sh
```

is a shell script. And even if `/bin/sh` points to `/bin/bash` bash runs in a compatibility mode and doesn't honor `BASH_ENV`.

To combat this Lmod exports the definition of the module command. This means that even `/bin/sh` scripts will have the module command defined when run by a Bash User. However, a Csh or Zsh user running a bash script will still need the `BASH_ENV` and run bash scripts. They won't have the module command defined if they run an sh script.

4.2 How to Transition to Lmod (or how to test Lmod without installing it for all)

In the *Installing Lua and Lmod* document, we described how to install Lua and Lmod for all. Sites which are currently running another environment module system will likely wish to test and then transition from their old module system to Lmod. This can be smoothly with changing all users on some Tuesday.

It is important to remember the following facts:

- Lmod reads modulefiles written in TCL. There is typically no need to translate modulefiles written in TCL into Lua. Lmod does this for you automatically.
- Some users can run Lmod while others use the old environment module system.

- However no user can run both at the same time in the same shell.

Obviously, since you are installing Lmod in your own account, this is a good way to test Lmod without committing your site to switch. Part of this document will describe TACC's transition experience.

4.2.1 Steps for Testing Lmod in your account

1. Install Lua
2. Install Lmod in your account
3. Build the list modules required
4. Purge modules using old module command
5. Reload modules using Lmod

Install Lua

The previous document described how to install Lua. If your system doesn't provide package for Lua, then it is probably easy to install the lua tarball found at [sourceforge.net](https://sourceforge.net/projects/lmod/files/lua-W.X.Y.Z.tar.gz) using the following command:

```
$ wget https://sourceforge.net/projects/lmod/files/lua-W.X.Y.Z.tar.gz
```

where you replace the *W.X.Y.Z* with the current version (i.e. 5.1.4.8).

Many linux distributions already have a lua package and it may even be install automatically. For example recent Centos and other Redhat based distributions automatically install Lua as part of the rpm tools.

Once you have lua installed and in your path. You'll need luafilesystem and luaposix libraries to complete the requirements. See the previous document on how to install these libraries via your package manager or luarocks.

Install Lmod

Please follow the previous document on how to install Lmod. Let's assume that you have installed Lmod in your own account like this:

```
$ ./configure --prefix=$HOME/pkg
$ make install
```

This will install Lmod in **\$HOME/pkg/lmod/x.y.z** and make a symbolic link to **\$HOME/pkg/lmod/lmod**.

Build the list of modules required

Many sites provide a default set of modules. When testing, you'll want to be able to load those list of modules using Lmod. Using your old module system, login and do:

```
$ module list

Currently Loaded Modules:

1) a1          3) A          5) b2          7) B
2) a2          4) b1         6) b3
```

It turns out that both the latest version of TCL/C modules and the pure TCL script list a module that loads other modules later in the list. In this made up case we unload module B and notice that unloading the B module also unloads modules b3, b2 and b1. Then unloading the A module also unloads modules a2 and a1. In this case then we would set:

```
export LMOD_SYSTEM_DEFAULT_MODULES=A:B
```

Purge modules using old module command

Execute:

```
$ module purge
```

to unload the currently loaded modules **using the old module command**.

Reload modules using Lmod

Once all modules have been purge and the environment variable LMOD_SYSTEM_DEFAULT_MODULES has been set. All that are required are to redefine the module command to use Lmod and to restore the default set of modules by:

```
$ export BASH_ENV=$HOME/pkg/lmod/lmod/init/bash
$ source $BASH_ENV
```

This will define the module command. Finally the default set of modules can be loaded.

```
$ module --initial_load restore
```

This command first looks to see if there is a default collection in **~/lmod.d/default**. If that file isn't found then it uses the value of variable LMOD_SYSTEM_DEFAULT_MODULES as a list of module to load.

If you have gotten this far then you have installed Lmod in your account. Congratulations!

Please test your system. Try to load your most complicated modulefiles. See if **module avail**, **module spider** works and so on.

If you have trouble loading certain TCL modulefiles then read the **How Lmod reads TCL modulefiles** to see why you might have problems.

4.2.2 An example of how this can be done in your bash startup scripts

All the comments above can be combined into a complete example:

```
if [ -z "$_INIT_LMOD" ]; then
  export _INIT_LMOD=1          # guard variable is crucial, to avoid breaking existing modules sett
  type module > /dev/null 2>&1
  if [ "$?" -eq 0 ]; then
    module purge >2 /dev/null  # purge old modules using old module command.
    clearMT                    # clear the stored module table (wipe _ModuleTable001_ etc.)
  fi

  export MODULEPATH=...        # define MODULEPATH
  export BASH_ENV=$HOME/pkg/lmod/lmod/init/bash # Point to the new definition of Lmod

  source $BASH_ENV             # Redefine the module command to point
                                # to the new Lmod
  export LMOD_SYSTEM_DEFAULT_MODULES=... # Colon separated list of modules
```

```

    module --initial_load restore
else
    source $BASH_ENV
    module refresh
fi
# to load at startup
# load either modules listed above or the
# user's ~/.lmod.d/default module collection
# redefine the module command for sub-shell
# reload all modules but only activate the "set_alias"
# functions.
```

Obviously, you will have to define **MODULEPATH** and **LMOD_SYSTEM_DEFAULT_MODULES** to match your site setup. The reason for the guard variable **_INIT_LMOD** is that the module command and the initialization of the modules is only done in the initial login shell. On any sub-shells, the module command gets defined (again). Finally the **module refresh** command is called to define any alias or shell functions in any of the currently loaded modules.

4.2.3 How to Transition to Lmod: Staff & Power User Testing

Once you have tested Lmod personally and wish to transition your site to use Lmod, I recommend the following strategy for staff and friendly/power users for testing:

1. Install Lua and Lmod in system locations
2. Install */etc/profile.d/z00_lmod.sh* to redefine the module command
3. Load system default modules (if any) after previous steps
4. Only user who have a file named *~/.lmod* use Lmod
5. At TACC, we did this for 6 months.

Using this strategy, you can have extended testing without exposing Lmod to any user which hasn't opted-in.

How to Deploy Lmod

Once Staff testing is complete and you are ready to deploy Lmod to your users it is quite easy to switch to an opt-out strategy:

1. Change */etc/profile.d/z00_lmod.sh* so that everyone is using Lmod
2. If a user has a *~/.no.lmod* then they continue to use your original module system
3. At TACC we did this for another 6 months
4. We broke Environment Module support with the family directive
5. We now only support Lmod
6. Both transitions generated very few tickets (2+2)

4.3 Lua Modulefile Functions

Lua is an efficient language built on simple syntax. Readers wanting to know more about lua can see <http://www.lua.org/>. This simple description given here should be sufficient to write all but the most complex modulefiles.

It is important to understand that modulefiles are written in the positive. That is one writes the actions necessary to activate the package. A modulefile contains commands to add to the **PATH** or set environment variables. When loading a modulefile the commands are followed. When unloading a modulefile the actions are reversed. That is the

element that was added to the PATH during loading, is removed during unloading. The environment variables set during loading are unset during unloading.

prepend_path (“PATH”, “/path/to/pkg/bin”): prepend to a path variable the value.

append_path (“PATH”, “/path/to/pkg/bin”): append to a path variable the value.

remove_path (“PATH”, “/path/to/pkg/bin”): remove value from path. This command is a no-op when the mode is unload.

setenv (“NAME”, “value”): assigns to the environment variable “NAME” the value.

pushenv (“NAME”, “value”): sets NAME to value just like **setenv**. In addition it saves the previous value in a hidden environment variable. This way the previous state can be returned when a module is unloaded.

unsetenv (“NAME”): unset the value associated with “NAME”. This command is a no-op when the mode is unload.

whatis (“STRING”): The **whatis** string, can be called repeatedly with different strings. See the Administrator Guide for more details.

help ([*help string*]): What is printed out when the help command is called. Note that the *help string* can be multi-lined.

pathJoin (“/a”, “b/c”, “d/”): builds a path: “/a/b/c/d”. It combines any number of strings with one slash and removes excess slashes. Note that trailing slash is removed. If you need a trailing slash then do **pathJoin**(“/a”, “b/c”) .. “/” to get “/a/b/c/”.

load (“pkgA”, “pkgB”, “pkgC”): load all modules. Report error if unable to load.

try_load (“pkgA”, “pkgB”, “pkgC”): load all modules. No errors reported if unable to load.

always_load (“pkgA”, “pkgB”, “pkgC”): load all modules. However when this command is reversed it does nothing.

set_alias (“name”, “value”): define an alias to name with value.

unload (“pkgA”, “pkgB”): When in load mode the modulefiles are unloaded. It is not an error to unload modules that were not loaded. When in unload mode, this command does nothing.

family (“name”): A user can only have one family “name” loaded at a time. For example family(“compiler”) would mean that a user could only have one compiler loaded at a time.

prereq (“name1”, “name2”): The current modulefile will only load if **all** the listed modules are already loaded.

prereq_any (“name1”, “name2”): The current modulefile will only load if **any** of the listed modules are already loaded.

conflict (“name1”, “name2”): The current modulefile will only load if all listed modules are NOT loaded.

4.3.1 Extra functions

The entries below describe several useful commands that come with Lmod that can be used in modulefiles.

os.getenv (“NAME”): Ask for environment for the value of “NAME”. Note that if the “NAME” might not be in the environment, then it is probably best to do:

```
local foo=os.getenv("FOO") or ""
```

otherwise `foo` will have the value of `nil`.

capture (“string”): Run the “string” as a command and capture the output.

isFile (“name”): Returns true if “name” is a file.

isDir (“name”): Returns true if “name” is a directory.

splitFileName (“name”): Returns both the directory and the file name. `local d, f=splitFileName("/a/b/c.ext")`. Then `d="/a/b"`, `f="c.ext"`

LmodMessage (“string”,...): Prints a message to the user.

LmodError (“string”,“...”): Print Error string and exit without loading the modulefile.

mode (): Returns the string “load” when a modulefile is being loaded and “unload” when unloading.

isLoading (“NAME”): Return true when module “NAME” is loaded.

LmodVersion (): The version of lmod.

execute {cmd=<any command>,modeA={“load”}} Run any command with a certain mode. For example **execute {cmd=“ulimit -s unlimited”,modeA={“load”}}** will run the command **ulimit -s unlimited** as the last thing that the loading the module will do.

4.3.2 Modifier functions to prereq and loads

atleast (“name”,“version”): This modifier function will only succeed if the module is “version” or newer.

between (“name”,“v1”,“v2”): This modifier function will only succeed if the module’s version is equal to or between “v1” and “v2”.

latest (“name”): This modifier function will only succeed if the module has the highest version on the system.

4.3.3 Introspection Functions

The following functions allow for more generic modulefiles by finding the name and version of a modulefile.

myModuleName (): Returns the name of the current modulefile without the version.

myModuleVersion (): Returns the version of the current modulefile.

myModuleFullName (): Returns the name and version of the current modulefile.

myModuleUsrName (): Returns the name the user specified to load a module. So it could be the name or the name and version.

myFileName (): Returns the absolute file name of the current modulefile.

myShellName ():

Returns the name of the shell the user specified on the command line.

hierarchyA (“fullName”, level): Returns the hierarchy of the current module. See the section on Generic Modules for more details.

4.4 How Lmod Picks which Modulefiles to Load

Lmod use the directories listed in `MODULEPATH` to find the modulefiles to load. Suppose that you have a single directory `/opt/apps/modulefiles` that has the following files and directories:

```
/opt/apps/modulefiles
StdEnv.lua  ucc/  xyz/

./ucc:
8.1.lua  8.2.lua
```

```
./xyz:
10.1.lua
```

Lmod will report the following directory tree like this:

```
----- /opt/apps/modulefiles -----
StdEnv    ucc/8.1    ucc/8.2 (D)    xyz/10.1
```

We note that the `.lua` extension has not been reported above. The `.lua` extension tells Lmod that the contents of the file are written in the Lua language. All other files are assumed to be written in TCL.

Here the name of the file or directory under `/opt/apps/modulefiles` is the name of the module. The normal way to specify a module is to create a directory to be the name of the module and the file(s) under that directory are the version(s). So we have created `ucc` and `xyz` directories to be the names of the module. There are two version files under `ucc` and one version file under `xyz`.

The `StdEnv.lua` file is the another way to specify a module. This file is a module with no version associated with it. These are typically used as a meta-module. That is a module that loads other modules.

4.4.1 Picking modules when there are multiple directories in MODULEPATH

When there are multiple directories specified in `MODULEPATH`, the rules get more complicated on what modulefile to load. Lmod uses the following rules to locate a modulefile:

1. It looks for an exact match in all `MODULEPATH` directories. Picking the first match it finds.
2. If the user requested name is a full name and version, and there is no exact match then it stops.
3. If the name doesn't contain a version then Lmod looks for a marked default in the first directory that has one.
4. Finally it looks for the "Highest" Version in all `MODULEPATH` directories. If there are two or more modulefiles with the "Highest" version then the first one in `MODULEPATH` order will be picked.
5. As a side node, if there are two version files, one with a `.lua` extension and one without, the lua file will be used over the other one. It will like the other file is not there.

As an example, suppose you have the following module tree:

```
----- /home/user/modulefiles -----
xyz/11.1

----- /opt/apps/modulefiles -----
StdEnv    ucc/8.1    ucc/8.2    xyz/10.1

----- /opt/apps/mfiles -----
ucc/8.3 (D)    xyz/12.1 (D)
```

If a user does the following command:

```
$ module load ucc/8.2 xyz
```

then `ucc/8.2` will be loaded because the user specified a particular version and `xyz/12.1` will be loaded because it is the highest version across all directories in `MODULEPATH`.

4.4.2 Marked a Version as Default

Suppose you have several versions of the mythical UCC compiler suite:

```
$ module avail ucc
----- /opt/apps/modulefiles/Core -----
ucc/8.1   ucc/9.2   ucc/11.1   ucc/12.2 (D)
```

and you like to make the 11.1 version the default. Lmod searches three different ways to mark a version as a default in the following order. The first way is to make a symbolic link between a file named “default” and the desired default version.:

```
$ cd /opt/apps/modulefiles/Core/ucc; ln -s 11.1.lua default
```

A second way to mark a default is with a .modulerc file in the same directory as the modulefiles.:

```
##Module
module-version ucc/11.1 default
```

There is a third method to pick the default module. If you create a .version file in the ucc directory that contains:

```
##Module
set ModulesVersion "11.1"
```

Please note that either a .modulerc or .version file must be in the same directory as the 11.1.lua file in order for Lmod to read it.

Using any of the above three ways will change the default to version 11.1.

```
$ module avail ucc
----- /opt/apps/modulefiles/Core -----
ucc/8.1   ucc/9.2   ucc/11.1 (D)   ucc/12.2
```

4.4.3 Highest Version

If there is no marked default then Lmod chooses the “Highest” version across all directories:

```
$ module avail ucc

----- /opt/apps/modulefiles/Core -----
ucc/8.1   ucc/9.2   ucc/11.1   ucc/12.2

----- /opt/apps/modulefiles/New -----
ucc/13.2 (D)
```

The “Highest” version is by version number sorting. So Lmod “knows” that the following versions are sorted from lowest to highest:

```
2.4dev1
2.4a1
2.4beta2
2.4rc1
2.4
2.4.0.0
2.4-1
2.4.0.0.1
2.4.1
```

4.4.4 Autoswaping Rules

When Lmod autoswaps hierarchical dependencies, it uses the following rules:

1. If a user loads a default module, then Lmod will reload the default even if the module version has changed.
2. If a user loads a module with the version specified then Lmod will only load the exact same version when swapping dependencies.

For example a user loads the intel and boost library:

```
$ module purge; module load intel boost; module list

Currently Loaded Modules:
1) intel/15.0.2  2) boost/1.57.0
```

Now swapping the Intel compiler suite for the Gnu compiler suite:

```
The following have been reloaded with a version change:
1) boost/1.57.0 => boost/1.56.0
```

Here boost has been reloaded with a different version because the default is different in the gcc hierarchy. However if the user does:

```
$ module purge; module load intel boost/1.57.0; module list

Currently Loaded Modules:
1) intel/15.0.2  2) boost/1.57.0
```

And:

```
$ module swap intel gcc;

Inactive Modules:
1) boost/1.57.0
```

Since the user initially specified loading boost/1.57.0 then Lmod assumes that the user really wants that version. Because version 1.57.0 of boost isn't available under the gcc hierarchy, Lmod marks this boost module as inactive. This is true even though version 1.57.0 is the default version of boost under the Intel hierarchy.

4.5 Providing A Standard Set Of Modules for all Users

Users can be provided with an initial set of modulefiles as part of the login procedure. Once a list of modulefiles have been installed, please create a file called StdEnv.lua and place it in the MODULEPATH list of directories, typically /opt/apps/modulefiles/Core/StdEnv.lua. The name is your choice, the purpose is provide a standard list of modules that get loaded during login. In StdEnv.lua is something like:

```
load("name1", "name2", "name3")
```

Using the /etc/profile.d directory system described earlier to create a file called z00_StdEnv.sh

```
if [ -z "$__Init_Default_Modules" -o -z "$LD_LIBRARY_PATH" ]; then
    export __Init_Default_Modules=1;
    export LMOD_SYSTEM_DEFAULT_MODULES="StdEnv"
    module --initial_load restore
else
    module refresh
fi
```

Similar for z00_StdEnv.csh:

```
if ( ! $?__Init_Default_Modules || ! $?LD_LIBRARY_PATH ) then
  setenv LMOD_SYSTEM_DEFAULT_MODULES "StdEnv"
  module --initial_load restore
  setenv __Init_Default_Modules 1
else
  module refresh
endif
```

The `z00_Stdenv.*` names are chosen because the files in `/etc/profile.d` are sourced in alphabetical order. These names guarantee they will run after the module command is defined.

The first time these files are source by a shell they will set `LMOD_SYSTEM_DEFAULT_MODULES` to `StdEnv` and then execute `module restore`. Any subshells will instead call `module refresh`. Both of these statements are important to get the correct behavior out of Lmod.

The `module restore` tries to restore the user's default collection. If that doesn't exist, it then uses contents of the variable `LMOD_SYSTEM_DEFAULT_MODULES` to find a colon separated list of Modules to load.

The `module refresh` solves an interesting problem. Sub shells inherit the environment variables of the parent but do not normally inherit the shell aliases and functions. This statement fixes this. Under a “refresh”, all the currently loaded modules are reloaded but in a special way. Only the functions which define alias and shell functions are active, all others functions are ignored.

ADVANCED TOPICS

5.1 How to report a bug in Lmod

Lmod has some built-in tools to make debugging possible on your site. The first feature of Lmod is the configuration report:

```
$ module --config
```

This reports how Lmod has been configured at build time as well as any `LMOD_*` environment variables set. The second tool is the debug output also built-in to Lmod:

```
$ module -D load foo 2> load.log
```

The `-D` option turns on the debug printing and will report all the steps that Lmod took to load a module called `foo`. Note that the configuration report is at the top of every debug output.

5.1.1 Steps to report a bug

1. Test your bug against the latest release from github. Please pull the HEAD branch.
2. Try to reduce the problem to the fewest number of modules. Shoot for 1 or 2 modulefiles if you can.
3. Run the command that fails. i.e. `module -D cmd module ... 2> lmod.log`
4. Combine the `lmod.log` file, the modulefiles from step 2, and possibly the spider cache file into a tar file.
5. Send the tar file to mclay@tacc.utexas.edu

5.2 How to use a Software Module hierarchy

Libraries built with one compiler need to be linked with applications with the same compiler version. For High Performance Computing there are libraries called Message Passing Interface (MPI) that allow for efficient communicating between tasks on a distributed memory computer with many processors. Parallel libraries and applications must be built with a matching MPI library and compiler. To make this discussion clearer, suppose we have the intel compiler version 15.0.1 and the gnu compiler collection version 4.9.2. Also we have two MPI libraries: mpich version 3.1.2 and openmpi version 1.8.2. Finally we have a parallel library HDF5 version 1.8.13 (phdf5).

Of the many possible ways of specifying a module layout, this flat layout of modules is a reasonable way to do this:

```
$ module avail  
  
----- /opt/apps/modulefiles -----  
gcc/4.9.2                               phdf5/gcc-4.9-mpich-3.1-1.8.13
```

intel/15.0.2	phdf5/gcc-4.9-openmpi-15.0-1.8.13
mpich/gcc-4.9-3.1.2	phdf5/intel-15.0-mpich-3.1-1.8.13
mpich/intel-15.0-3.1.2	phdf5/intel-15.0-openmpi-15.0-1.8.13
openmpi/gcc-4.9-1.8.2	
openmpi/intel-15.0-1.8.2	

In order for users to load the matching set of modules, they will have to load the matching set of modules. For example this would be correct:

```
$ module load gcc/4.9.2 openmpi/gcc-4.9-1.8.2 phdf5/gcc-4.9-openmpi-15.0-1.8.13
```

It is quite easy to load an incompatible set of modules. Now it is possible that the system administrators at your site might have setup `conflict s` to avoid loading mismatched modules. However using conflicts can be fragile. What happens if a site adds a new compiler such as clang or pgi or a new mpi stack. All those module file conflict statements will have to be updated.

A different strategy is to use a software hierarchy. In this approach a user loads a compiler which extends the **MODULEPATH** to make available the modules that are built with the currently loaded compiler (similarly for the mpi stack).

Our modulefile hierarchy is stored under `/opt/apps/modulefiles/{Core,Compiler,MPI}`. The Core directory is for modules that are not dependent on Compiler or MPI implementations. The Compiler directory is for packages which are only Compiler dependent. Lastly, the MPI directory is packages which dependent on MPI-Compiler pairing. The modulefiles for the compilers are placed in the Core directory. For example the gcc version 4.9.2 file is in `Core/gcc/4.9.2.lua` and contains:

```
-- Setup Modulepath for packages built by this compiler
local mroot = os.getenv("MODULEPATH_ROOT")
local mdir = pathJoin(mroot, "Compiler/gcc", "4.9")
prepend_path("MODULEPATH", mdir)
```

This code asks the environment for **MODULEPATH_ROOT** which is `/opt/apps/modulefiles` and the last two lines prepend `/opt/apps/modulefiles/Compiler/gcc/4.9` to the **MODULEPATH**.

The modulefiles for the MPI implementations are placed under the Compiler directory because they only depend on a compiler. The openmpi module file for the gcc-4.9.2 compiler is then stored at `/opt/apps/modulefiles/Compilers/gcc/4.9/openmpi/1.8.2.lua` and it contains:

```
-- Setup Modulepath for packages built by this MPI stack
local mroot = os.getenv("MODULEPATH_ROOT")
local mdir = pathJoin(mroot, "MPI/gcc", "4.9", "openmpi", "1.8")
prepend_path("MODULEPATH", mdir)
```

The above code will prepend `/opt/apps/modulefiles/MPI/gcc/4.9/openmpi/1.8` to the **MODULEPATH**.

We store packages as follows:

1. Core packages: **/opt/apps/pkgName/version**
2. Compiler dependent packages: **/opt/apps/compilerName-version/pkgName/version**
3. MPI-Compiler dependent packages: **/opt/apps/compilerName-version/mpiName-version/pkgName/version**

When **MODULEPATH** changes, Lmod unloads any modules which are not currently in the **MODULEPATH** and then tries to reload all the previously loaded modules. Any modules which are not available are marked as inactive. Those inactive modules become active if found with new **MODULEPATH** changes.

Note: In all of the example above, We have used just the first two version numbers. In other words, we have use 4.9 and not 4.9.2 and similarly 1.8 instead of 1.8.2. It is our view that for at least compilers and MPI stacks that the third

digit is typically a bug fix and doesn't require rebuilding all the dependent packages. Y.M.M.V.

5.3 Configuring Lmod for your site

How to configure Lmod to match site expectations?

5.4 How does Lmod convert TCL modulefile into Lua

Lmod uses a TCL program call **tcl2lua.tcl** to read TCL modulefiles and convert them to lua. The whole TCL modulefile is run through. However instead of executing the “module functions” they are converted to Lua. For example, suppose you have the following simple TCL modulefile for git:

```

#%Module
set appDir      $env(APP_DIR)
set version     2.0.3

prepend-path    PATH "$appDir/git/$version/bin"

```

Assuing that the environment variable APP_DIR is */apps* then the output of the **tcl2lua.tcl** program would be:

```
prepend_path("PATH", "/apps/git/2.0.3/bin")
```

Note that all the normal TCL code has been evaluated and the TCL **prepend-path** command has been converted to a lua **prepend_path** function call.

Normally this works fine. However, because Lmod does evaluate the actions of a TCL module file as a two-step process, it can cause problem. In particular, suppose you have two TCL modulefiles:

Centos:

```

#%Module
setenv SYSTEM_NAME Centos

```

And B:

```

#%Module
module load Centos

if { $env(SYSTEM_NAME) == "Centos" } {
    # do something
}

```

When Lmod tries to translate the B modulefile into lua it fails:

```

load("Centos")
LmodError([[/opt/mfiles/B: (???): can't read "env(SYSTEM_NAME)": no such variable]])

```

This is because unlike the TCL/C Module system, the **module load Centos** command is converted to a function call, but it won't load the module in time for the test to be evaluated properly.

The only solution is convert the B modulefile into a Lua modulefile (B.lua):

```

load("Centos")
if (os.getenv("SYSTEM_NAME") == "Centos") then
    -- Do something
end

```

The Centos modulefile does not have to be translated in order for this to work, just the B modulefile.

As a side note, you are free to put Lua modules in the same tree that the TCL/C Module system uses. It will ignore files that the first line is not `#%Module` and Lmod will pick B.lua over B.

5.5 Generic Modules

Lmod provides inspection functions that describe the name and version of a modulefile as well as the path to the modulefile. These functions provide a way to write “generic” modulefiles, i.e. modulefiles that can fill in its values based on the location of the file itself.

These ideas work best in the software hierarchy style of modulefiles. For example: suppose the following is a modulefile for Git. Its modulefile is located in the “/apps/mfiles/Core/git” directory and software is installed in “/apps/git/<version>”. The following modulefile would work for every version of git:

```
local pkg = pathJoin("/apps",myModuleName(),myModuleVersion())
local bin = pathJoin(pkg,"bin")
prepend_path("PATH",bin)

whatis("Name:          ", myModuleName())
whatis("Version:       ", myModuleVersion())
whatis("Description:   ", "Git is a fast distributive version control system")
```

The contents of this modulefile can be used for multiple versions of the git software, because the local variable `bin` changes the location of the bin directory to match the version of the used as the name of the file. So if the module file is in `/apps/mfiles/Core/git/2.3.4.lua` then the local variable `bin` will be `/apps/git/2.3.4`.

5.5.1 Relative Paths

Suppose you are interested in modules where the module and application location are relative. Suppose that you have an \$APPS directory, and below that you have modulefiles and packages, and you would like the modulefiles to find the absolute path of the package location. This can be done with the `myFileName()` function and some lua code:

```
local fn      = myFileName()          -- 1
local full    = myModuleFullName()    -- 2
local loc     = fn:find(full,1,true)-2 -- 3
local mdir    = fn:sub(1,loc)          -- 4
local appsDir = mdir:gsub("(.*)/","%1") -- 5
local pkg     = pathJoin(appsDir, full) -- 6
```

To make this example concrete, let’s assume that applications are in `/home/user/apps` and the modulefiles are in `/home/user/apps/mfiles`. So if the modulefile is located at `/home/user/apps/mfiles/git/1.2.lua`, then that is the value of `fn` at line 1. The `full` variable at line 2 will have `git/1.2`. What we want is to remove the name of the modulefile and find its parent directory. So we use Lua string member function on `fn` to find where `full` starts. In most cases `fn:find(full)` would work to find where the “git” starts in `fn`. The trouble is that the Lua find function is expecting a regular expression and in particular `.` and `-` are regular expression characters. So here we are using `fn:find(full,1,true)` to tell Lua to treat each character as is with no special meaning.

Line 3 also subtracts 2. The find command reports the location of the start of the string where the “g” in “git” is, We want the value of `mdir` to be `/home/user/apps/mfiles` so we need to subtract 2. This makes `mdir` have the right value. One note is that Lua is a one based language, so locations in strings start at one.

It was important for the value of `mdir` to remove the trailing `/` so that line 5 will do its magic. We want the parent directory of `mdir`, so the regular expressions says greedily grab every character until the trailing `/` and the `%1` says to capture the string found in and use that to set `appsdir` to `/home/user/apps`. Finally we wish to

set `pkg` to the location of the actual application so we combine the value of `appsdir` and `full` to set `pkg` to `/home/user/apps/git/1.2`.

The nice thing about this Lua code is that it figures out the location of the package no matter where it is as long as the relation between apps directories and modulefiles is consistent.

Creating modules like this can be complicated. See `debugging_modulefiles_label` for helpful tips.

5.5.2 Generic Modules with the Hierarchy

This works great for Core modules. It is a little more complicated for Compiler or MPI/Compiler dependent modules but quite useful. For a concrete example, let's cover how to handle the boost C++ library. This is obviously a compiler dependent module. Suppose you have the gnu compiler collection (gcc) and the intel compiler collection (intel), which means that you'll have a gcc version and an intel version for each version of boost.

In order to have generic modules for compiler dependent modules, there must be some conventions to make this work. A suggested way to do this is the following:

1. Core modules are placed in `/apps/mfiles/Core`. These are the compilers, programs like git and so on.
2. Core software goes in `/apps/<app-name>/<app-version>`. So git version 2.3.4 goes in `/apps/git/2.3.4`
3. Compiler-dependent modulefiles go in `/apps/mfiles/Compiler/<compiler>/<compiler-version>/<app-name>/<app-version>` using the **two-digit** rule (discussed below). So the Boost 1.55.0 modulefile built with gcc/4.8.3 would be found in `/apps/mfiles/Compiler/gcc/4.8/boost/1.55.0.lua`
4. Compiler-dependent packages go in `/apps/<compiler-version>/<app-name>/<app-version>`. So the same Boost 1.55.0 package built with gcc 4.8.3 would be placed in `/apps/gcc-4_8/boost/1.55.0`

The above convention depends on the **two-digit** rule. For compilers and mpi stack, we are making the assumption that compiler dependent libraries built with gcc 4.8.1 can be used with gcc 4.8.3. This is not always safe but it works well enough in practice. The above convention also assumes that the boost 1.55.0 package will be placed in `/apps/gcc-4_8/boost/1.55.0`. It couldn't go in `/apps/gcc/4.8/...` because that is where the gcc 4.8 package would be placed and it is not a good idea to co-mingle two different packages in the same tree. Another possible choice would be `/apps/gcc-4.8/boost/1.55.0`. It is my view that it looks too much like the gcc version 4.8 package location where as `gcc-4_8` doesn't.

With all of the above assumptions, we can now create a generic module file for compiler dependent modules such as Boost. In order to make this work, we will need to use the `hierarchyA` function. This function parses the path of the modulefile to return the pieces we need to create a generic boost modulefile:

```
hierA = hierarchyA(myModuleFullName(), 1)
```

The `myModuleFullName()` function returns the full name of the module. So if the module is named **boost/1.55.0**, then that is what it will return. If your site uses module names like `lib/boost/1.55.0` then it will return that correctly as well. The `1` tells Lmod to return just one component from the path. So if the modulefile is located at `/apps/mfiles/Compiler/gcc/4.8/boost/1.55.0.lua`, then `myModuleFullName()` returns **boost/1.55.0** and the `hierarchyA` function returns an array with 1 entry. In this case it returns:

```
{ "gcc/4.8" }
```

The rest of the module file then can make use to this result to form the paths:

```
local pkgName      = myModuleName()
local fullVersion  = myModuleVersion()
local hierA        = hierarchyA(myModuleFullName(), 1)
local compilerD    = hierA[1]:gsub("/", "-"):gsub("%.", "_")
local base         = pathJoin("/apps", compilerD, pkgName, fullVersion)
```

```
whatis("Name: "..pkgName)
whatis("Version: "..fullVersion)
whatis("Category: library")
whatis("Description: Boost provides free peer-reviewed "..
      " portable C++ source libraries.")
whatis("URL: http://www.boost.org")
whatis("Keyword: library, c++")

setenv("TACC_BOOST_LIB", pathJoin(base, "lib"))
setenv("TACC_BOOST_INC", pathJoin(base, "include"))
```

The important trick is the building of the *compilerD* variable. It converts the *gcc/4.8* into *gcc-4_8*. This makes the *base* variable be: */apps/gcc-4_8/boost/1.55.0*.

Creating modules like this can be complicated. See `debugging_modulefiles_label` for helpful tips.

A proposed directory structure of */apps/mfiles/Compiler* would be:

```
.base/      gcc/   intel/

.base/
boost/generic.lua

gcc/4.8/boost/

1.55.0.lua -> ../../../../base/boost/generic.lua

intel/15.0.2/boost/

1.55.0.lua -> ../../../../base/boost/generic.lua
```

In this way the *.base/boost/generic.lua* file will be the source file for all the boost version build with gcc and intel compilers.

The same technique can be applied for modulefiles for Compiler/MPI dependent packages. In this case, we will create the *phdf5* modulefile. This is a parallel I/O package that allows for Hierarchical output. The modulefile is:

```
local pkgName      = myModuleName()
local pkgVersion    = myModuleVersion()
local pkgNameVer    = myModuleFullName()

local hierA         = hierarchyA(pkgNameVer, 2)
local mpiD          = hierA[1]:gsub("/", "-"):gsub("%.", "_")
local compilerD     = hierA[2]:gsub("/", "-"):gsub("%.", "_")
local base          = pathJoin("/apps", compilerD, mpiD, pkgNameVer)

setenv("TACC_HDF5_DIR", base)
setenv("TACC_HDF5_DOC", pathJoin(base, "doc"))
setenv("TACC_HDF5_INC", pathJoin(base, "include"))
setenv("TACC_HDF5_LIB", pathJoin(base, "lib"))
setenv("TACC_HDF5_BIN", pathJoin(base, "bin"))
prepend_path("PATH", pathJoin(base, "bin"))
prepend_path("LD_LIBRARY_PATH", pathJoin(base, "lib"))

whatis("Name: Parallel HDF5")
whatis("Version: " .. pkgVersion)
whatis("Category: library, mathematics")
whatis("URL: http://www.hdfgroup.org/HDF5")
whatis("Description: General purpose library and file format for storing scientific data (parallel I/O)")
```

We use the same tricks as before. It is just that since the module for phdf5 built by gcc/4.8.3 and mpich/3.1.2 will be found at `/apps/mfiles/MPI/gcc/4.8/mpich/3.1/phdf5/1.8.14.lua`. The results of `hierarchyA(pkgNameVer,2)` would be:

```
{ "mpich/3.1", "gcc/4.8" }
```

This is because the `hierarchyA` works back up the path two elements at a time because the full name of this package is also two elements (phdf5/1.8.14). The `base` variable now becomes:

```
/apps/gcc-4_8/mpich-3_1/phdf5/1.8.14
```

The last type of modulefile that needs to be discussed is an mpi stack modulefile such as mpich/3.1.2. This modulefile is more complicated because it has to implement the two-digit rule, build the path to the package and build the new entry to the **MODULEPATH**. The modulefile is:

```
local pkgNameVer = myModuleFullName()
local pkgName    = myModuleName()
local fullVersion = myModuleVersion()
local pkgV       = fullVersion:match('( %d+%. %d+)%?.?')

local hierA      = hierarchyA(pkgNameVer,1)
local compilerV  = hierA[1]
local compilerD  = compilerV:gsub("/", "-"):gsub("%.", "_")
local base       = pathJoin("/apps", compilerD, pkgName, fullVersion)
local mpath      = pathJoin("/apps/mfiles/MPI", compilerV, pkgName, pkgV)

prepend_path("MODULEPATH", mpath)
setenv("TACC_MPICH_DIR", base)
setenv("TACC_MPICH_LIB", pathJoin(base, "lib"))
setenv("TACC_MPICH_BIN", pathJoin(base, "bin"))
setenv("TACC_MPICH_INC", pathJoin(base, "include"))

whatis("Name: ..pkgName")
whatis("Version ..fullVersion")
whatis("Category: mpi")
whatis("Description: High-Performance Portable MPI")
whatis("URL: http://www.mpich.org")
```

The **Two Digit** rule implemented by forming the `pkgV` variable. The `base` and `mpath` are:

```
base = "/apps/gcc-4_8/mpich-3_1/phdf5/1.8.14"
mpath = "/apps/mfiles/MPI/gcc/4.8/mpich/3.1"
```

The `rt` directory contains all the regression test used by Lmod. As such they contain many examples of modulefiles. To complement this description, the `rt/hierarchy/mf` directory from the source tree contains a complete hierarchy.

5.6 The Interaction between Modules, MPI and Parallel Filesystems

Items to discuss:

1. This is a complicated issue with many parts.
2. Mention Bash startup: What does a non-prompt non-login interactive shell mean.
3. MPI startup: non-login interactive shells.
4. Lustre and MDS and finding a module.
5. ibrun passing of the environment
6. Why unguarded module commands are a problem

7. Why TACC makes the module command a no-op when on a compute node.

5.7 Lmod on Shared Home File Systems

Sites that use a shared home file system across multiple clusters should take some extra steps to ensure the smooth running of Lmod. Typically each cluster will use different modules.

There are three steps that will make Lmod run smoothly on a shared home filesystem:

1. It is best to have a separate installation of Lmod on each cluster.
2. Define the environment variable “LMOD_SYSTEM_NAME” uniquely for each cluster.
3. If you build a system spider cache, then build a separate cache for each cluster.

A separate installation on each cluster is the safest way to install Lmod. It is possible to have a single installation but since there is some C code build with Lmod, this has to work on all clusters. Also the location of the Lua interpreter must be exactly the same on each cluster.

It is also recommended that you set “LMOD_SYSTEM_NAME” outside of a modulefile. It would be bad if a module purge would clear that value. When you set this variable, it makes the module collections and user spider caches unique for a given cluster.

A separate system spider cache is really the only way to go. Otherwise a “module spider” will report modules that don’t exist on the current cluster. If you have a separate install of Lmod on each cluster then you can specify the location of system cache at configure time. If you don’t, you can use the “LMOD_RC” environment to specify the location of the lmodrc.lua file uniquely on each cluster.

Lmod knows about the system spider cache from the lmodrc.lua file. If you install separate instances of Lmod on each cluster, Lmod builds the scDescriptT table for you. Otherwise you can modify lmodrc.lua to point to the system cache by adding scDescriptT to the end of the file:

```
scDescriptT = {
  {
    ["dir"] = "<location of your system cache directory>",
    ["timestamp"] = "<location of your timestamp file",
  },
}
```

where you have filled in the location of both the system cache directory and timestamp file.

5.8 System Spider Cache

Now with version 5.+ of Lmod, it is now very important that sites with large modulefile installations build system spider cache files. There is a file called “update_lmod_system_cache_files” that builds a system cache file. It also touches a file called “system.txt”. Whatever the name of this file is, Lmod uses this file to know that the spider cache is up-to-date.

Lmod uses the spider cache file as a replacement for walking the directory tree to find all modulefiles in your MODULEPATH. This means that Lmod only knows about system modules that are found in the spider cache. Lmod won’t know about any system modules that are not in this cache. (Personal module files are always found). It turns out that reading a single file is much faster than walking the directory tree.

While building the spider cache, each modulefile is evaluated for changes to MODULEPATH. Any directories added to MODULEPATH are also walked. This means if your site uses the software hierarchy then the new directories added by compiler or mpi stack modulefiles will also be searched.

Sites running Lmod have three choices:

1. Do not create a spider cache for system modules. This will work fine as long as the number of modules is not too large. You will know when it is time to start building a cache file when you start getting complains how long it takes to do any module commands.
2. If you have a formal proceeedure for installing packages on your system, then I recommend you to do the following. Have the install proceeedure run the `update_lmod_system_cache_files` script. This will create a file called “system.txt”, which marks the time that the system was last updated, so that Lmod knows that the cache is still good.
3. Or you can run the `update_lmod_system_cache_files` script say every 30 minutes. This way the cache file is up-to-date. No new module will be unknown for more than 30 minutes.

There are two ways to specify how cache directories and timestep files are specified. You can use “--with-spiderCacheDir=dirs” and “--with-updateSystemFn=file” to specify one or more directories with a single timestamp file:

```
./configure --with-spiderCacheDir=/opt/mData/cacheDir --with-updateSystemFn=/opt/mdata/system.txt
```

If you have multiple directories with multiple timestamp files you can use “--with-spiderCacheDescript=file” where the contents of the “file” is:

```
cacheDir1:timestamp1
cacheDir2:timestamp2
```

Lines starting with ‘#’ and blank lines are ignored. Please also note that a single timestamp file can be used with multiple cache directories.

5.8.1 How to decide how many system cache directories to have

The answer to this question depends on which machines “owns” which modulefiles. Many sites have a single location where their modulefiles are stored. In this case a single system cache file is all that is required.

At TACC, we need two system cache files because we have two different locations of files: one in the shared location and one on a local disk. So in our case Lmod sees two cache directories. Each node builds a spider cache of the modulefiles it “owns” and a single node (we call it master) builds a cache for the shared location.

5.8.2 What directories to specify?

If your site doesn’t use the software hierarchy, (see [How to use a Software Module hierarchy](#) for more details) then just use all the directory specified in **MODULEPATH**. If you do use the hierarchy, then just specify the “Core” directories, i.e. the directories that are used to initialize Lmod but not the compiler dependent or mpi-compiler dependent directories.

5.8.3 How to test the Spider Cache Generation and Usage

In a couple of steps you can generate a personal spider cache and get the installed copy of Lmod to use it. The first step would be to load the lmod module and then run the **update_lmod_system_cache_files** program and place the cache in the directory `~/moduleData/cacheDir` and the time stamp file in `~/moduleData/system.txt`:

```
$ module load lmod
$ update_lmod_system_cache_files -d ~/moduleData/cacheDir -t ~/moduleData/system.txt $LMOD_DEFAULT_M
```

Here we have use the trick that Lmod keeps track of the Core module directories in **LMOD_DEFAULT_MODULEPATH** so it should be safe to use, no matter whether your site is using the hierarchy or not.

Next you need to find your site's copy of `lmodrc.lua`. This can be found by running:

```
$ module --config
...
Active RC file(s):
-----
/opt/apps/lmod/6.0.14/init/lmodrc.lua
```

It is likely your site will have it in a different location. Please copy that file to `~/lmodrc.lua`. Then change the bottom of the file to be:

```
scDescriptT = {
  {
    ["dir"]      = "/path/to/moduleData/cacheDir",
    ["timestamp"] = "/path/to/moduleData/system.txt",
  },
}
```

where you have changed `/path/to` to match your home directory. Now set:

```
$ export LMOD_RC=$HOME/lmodrc.lua
```

Then you can check to see that it works by running:

```
$ module --config
...
Cache Directory      Time Stamp File
-----
$HOME/moduleData/cacheDir  $HOME/moduleData/system.txt
```

Where **\$HOME** is replaced by your real home directory. Now you can test that it works by doing:

```
$ module avail
```

The above command should be much faster than running without the cache:

```
$ module --ignore_cache avail
```

5.9 Deprecating Modules

There may come a time when your site might want to mark a module for deprecation. If you track module usage, you can find the modules that are rarely used, and you can find out which users are using the modules. Once you have decided which modules are marked for removal, you can make a message be printed when the module is loaded.

You can create a file called “admin.list” and place it in “`/path/to/lmod/etc/admin.list`”. Note that typically the `lmod` script will be in “`/path/to/lmod/lmod/libexec/lmod`”. The `etc` directory is independent to the version of Lmod. You can see the location that Lmod is looking for by executing:

```
$ module --config
```

Look for “Admin File”. You can also set the “`LMOD_ADMIN_FILE`” to point to the `admin.list` file.

The admin file consists of key-value pairs. For example:

```
moduleName/version:  message
<blank line>
```

Or:

```
Full_PATH_to_Modulefile: message
<blank line>
```

The message can be as many lines as you like. The message ends with a blank line. Below is an example:

```
gcc/2.95:      This module is deprecated and will be removed from the system on Jan 1. 1999.
               Please change you use of this compiler to a newer one.

boost/1.54.0:
We are having issues

/opt/apps/modulefiles/Compiler/gcc/4.7.2/boost/1.55.0:
We are having issues
```

Note that you don't include the .lua part when specifying the version number.

5.10 Kitchen Sink Modulefiles

Most of the time a modulefile is just a collection of setting environment variables and prepending to PATH or other path like variables. However, the modulefiles are actually programs so you can do a great deal if necessary.

5.10.1 Introspection

5.11 SitePackage.lua and hooks

Sites may wish to alter the behavior of Lmod to suit their needs. A good place to do this is the `SitePackage.lua`. Anything in this file will automatically be loaded everytime the Lmod command is run. This file can be used to provide common functions that can be used in a sites modulefiles. It is also a place where a site can implement their hook functions.

Hook functions are normal functions that if implemented and registered with Lmod will be called when certain operations happen inside Lmod. For example, there is a load hook. A site can register a function that is called everytime a module is loaded. There are several hook functions that are discussed in [Hook functions](#).

5.11.1 How to set up SitePackage.lua

Here are two suggestions on how to use your SitePackage.lua file:

1. Install Lmod normally and then overwrite your SitePackage.lua file over this one in the install directory.
2. Create a file named "SitePackage.lua" in a different directory separate from the Lmod installed directory and it will override the one in the Lmod install directory. Then you should modify your `z00_lmod.sh` and `z00_lmod.csh` (or however you initialize the "module" command) with:

```
(for bash, zsh, etc)
export LMOD_PACKAGE_PATH=/path/to/the/Site/Directory

(for csh)
setenv LMOD_PACKAGE_PATH /path/to/the/Site/Directory
```

5.11.2 Implementing functions in SitePackage.lua

For example your site might wish to provide the following function to set MODULEPATH inside your SitePackage:

```
function prependModulePath(subdir)
    local mroot = os.getenv("MODULEPATH_ROOT")
    local mdir = pathJoin(mroot, subdir)
    prepend_path("MODULEPATH", mdir)
end
```

This function must be registered with the sandbox so that Lmod modulefiles can call it:

```
sandbox_registration{ prependModulePath    = prependModulePath }
```

5.11.3 Hook functions

****load**(...):** This function is called after a modulefile is loaded.

****unload**(...):** This function is called after a modulefile is unloaded.

****parse_updateFn**(...):** This hook returns the time on the timestamp file.

****writeCache**(...):** This hook return whether a cache should be written.

****SiteName**(...):** This hook is used to specify Site Name. It is used to generate family prefix: `site_FAMILY_`

****msgHook**(...):** Hook to print messages after avail, list, spider, LmodError and LmodWarning.

****groupName**(...):** This hook adds the arch and os name to moduleT.lua to make it safe on shared filesystems.

****avail**(...):** Map directory names to labels

****restore**(...):** This hook is run after restore operation

****startup(UsrCmd):** This hook is run when Lmod is called

****packagebasename(s_patDir, s_patLib):** This hook gives you a table with the current patterns that spider uses to construct the reverse map.

TOPICS YET TO BE WRITTEN

1. Optional Software layout, two digit rule
2. Module naming conventions
3. Advanced Topics: priority path, .modulerc tricks
4. settarg
5. tracking module usage
6. converting shell scripts into modulefiles
7. module command and a parallel a file system.
8. inherit
9. internal structure of lmod.

INDICES AND TABLES

- `genindex`
- `modindex`
- `search`