TEXAS ADVANCED COMPUTING CENTER

WWW.TACC.UTEXAS.EDU

TEXAS
The University of Texas at Austin

How Lmod Loads a Modulefile, Part 1

Robert McLay

June. 7, 2022

Outline

- Talk: How Lmod finds a modulefile to load
- Next Time: Part 2: How Lmod actually evaluates a modulefile
- There is a surprising amount to talk about before the modulefile gets evaluated.
- We will be talking about a user requested load.
- A module requested load is slightly different.

Outline (II)

- `main()`: How main() parses the command line
- `Load_Usr()`: Starts working your modules to load
- `l_usrLoad()`: Splits modules to loads or unloads
- `MName Class`: Maps module name to a filename
- `mcp`: Object to control kind of evaluation
- `MasterControl:load(mA)`: We are loading a modulefile
- `Master:load(mA)`: Where the heavy lifting is done
- `loadModuleFile()`: Where the modulefile is actually evaluated

`main()`: parsing command line

- main() is located in lmod.in.lua (installed as lmod)
- The command is: *module load foo*
- To parse Lmod uses the 2nd argument w/o a minus: *load*
- Lmod searches `lmodCmdA` to map string to command

A side note on Lmod Coding Conventions

- A variable w/ a trailing “A” means that an array
- A variable w/ a trailing “T” means that an table (or dictionary)
- A variable w/ a trailing “Tbl” means that an table (or dictionary)
- A routine with a l_name is a local function (file scope)
- Class Name are in CamelCase

lmodCmdA

```
local lmodCmdA = {
   {cmd = 'add',          min = 2, action = loadTbl     },
   {cmd = 'avail',        min = 2, action = availTbl    },
   {cmd = 'isloaded',     min = 3, action = isLoadedTbl },
   {cmd = 'is_loaded',    min = 4, action = isLoadedTbl },
   {cmd = 'is-loaded',    min = 4, action = isLoadedTbl },
   {cmd = 'load',         min = 2, action = loadTbl     },
   ...
}
local loadTbl      = { name = "load", cmd = Load_Usr    }
```

- So “load” matches “load” with more than 2 chars

Going from lmodCmdA to Load_Usr()

- Sets cmdT to loadTbl
- Sets cmdName to cmdT.name (forces standard name not user command name)
- cmdT.cmd(unpack(masterTbl.pargs)) $\Rightarrow$ Load_Usr

Calling Load_Usr()

- All functions implementing user commands are in src/cmdfunc.lua

```
function Load_Try(...)
   dbg.start"Load_Try(",concatTbl({...},", "),")"
   local check_must_load = false
   local argA            = pack(...)
   l_usrLoad(argA, check_must_load)
   dbg.fini("Load_Try")
end

function Load_Usr(...)
   dbg.start"Load_Usr(",concatTbl({...},", "),")"
   local check_must_load = true
   local argA            = pack(...)
   l_usrLoad(argA, check_must_load)
   dbg.fini("Load_Usr")
end
```

l_usrLoad(argA, check_must_load)

- Split argA into loads in lA, unloads in uA (-foo)
- Both uA and lA are an array of MName objects.
- unload modules in uA
- `lA[#lA+1] = MName:new("load",module_name)`
- `mcp:load_usr(lA)`
- src: src/cmdfunc.lua

MName class: Module Name class

- Maps name (“foo” or “foo/1.1”) to filename
- There are two kinds of searching “load” or “mt”
- Load: must search file system.
- mt: filename is in moduletable
- Evaluation must be lazy or just-in-time
- Software hierarchy means that
- `module load gcc mpich`
- mpich might not be in $MODULEPATH until after gcc is loaded
- src: src/MName.lua

MName key concepts

- **userName**: name on the command line
- It might be gcc or gcc/9.3.0
- **sn**: the shortName or a name without a version
- **fullName**: The full name of the module (sn/version)
- Examples:
  1. gcc/9.3.0 (sn: gcc, N/V)
  2. gcc/x86_64/9.3.0 (sn: gcc, N/V/V)
  3. compiler/gcc/9.3.0 (sn: compiler/gcc, C/N/V)
  4. compiler/gcc/x86_64/9.3.0 (sn: compiler/gcc, N/V/V)

mcp and MasterControl class

- MasterControl class is what controls whether a ”load” in a modulefile is a load or unload
- mcp is a global variable that is built to be in a mode() like load, unload, spider, etc.
- We talked about this in an earlier presentation.

MasterControl:load(mA)

```
function M.load(self, mA)
   local master = Master:singleton()
   local a      = master:load(mA)

   if (not quiet()) then
      self:registerAdminMsg(mA) -- nag msg registration.
   end
   return a
end
```

- MasterControl functions call Master Functions to do the work.
- src: src/MasterControl.lua

Master:load(mA)

```
function M.load(mA)
  for i = 1, #mA do
     repeat
        mname = mA[i]
        sn = mname:sn()  -- shortName
        fn = mname:fn()  -- file name
        -- if blank sn -> pushModule (might have to wait until
        -- compiler or mpi is loaded.
        -- and break (really continue)

        if (mt:have(sn,"active)) then
           -- Block 1: Check for previously loaded module with same sn

        elseif (fn) then
           -- Block 2: Load modulefile

        -- Check for family stack (e.g. compiler, mpi etc)
        if (mcp.processFamilyStack(fullName)) then
            -- Suppose gcc is loaded and it was "replaced" by intel
            -- unload gcc and reload intel
         end
      until true
   end

   -- Reload every module if change in MODULEPATH.

   -- load any modules on module stack
end
```

- This is where the heavy lifting is done.
- src: src/Master.lua

Block 1: Check for previous loaded module w/same sn

```
if (mt:have(sn,"active)) then
   -- if disable_same_name_autoswap -> error out
   -- Otherwise: unload previous module
    local mcp_old = mcp
    local mcp     = MCP
    unload_internalMName:new("mt",sn)
    mname:reset()  -- force a new lazyEval
    local status = mcp:load_usr{mname}
    mcp          = mcp_old
```

- Here we guarantee the right mcp
- Unload the old module
- Recursively mcp:load_usrmname

Block 2: Load modulefile

```
elseif (fn) then
   frameStk:push(mname)
   mt = frameStk:mt()
   mt:add(mname,"pending")
   local status = loadModuleFile{file = fn, shell = shellNm,
                    mList = mList, reportErr = true}
   frameStk:pop()
   loaded = true
end
```

loadModuleFile(t)

- This is where Lmod handle either *.lua files or TCL Modulefiles
- Once either read in as a block (for *.lua) or converted (TCL modulefile ⇒ Lua)
- src: src/loadModulefile.lua

```
-- Use the sandbox to evaluate modulefile text.
if (whole) then
   status, msg = sandbox_run(whole)
else
   status = nil
   msg    = "Empty or non-existent file"
end
-- report any errors
```

Next time

- What is a sandbox and how does it work?
- Why I want a sandbox?
- Next time handing control to modulefile

Conclusions

- It takes a lot to get to the point where Lmod is evaluating your modulefile.
- Lmod uses several “classes” to manage the loading of a module
- Plus a couple of Design Patterns such as Singletons

Future Topics

- Next Meeting: July 5th 9:30 US Central (14:30 UTC)
- What happens from the loadModulefile(t).
- This is where Lmod hands off control to the user.