Slide 1

Administrivia

- (In e-mail.)

Slide 2

The "Modules" Package

- You've maybe been told that to make use of some non-default versions of things you should "use the `module` command"? and that you could put such a command in your `.bashrc` file?
- `module` is the user interface to the Environment Modules package. Probably available in many Linux distributions' standard repositories. Project Web page `modules.sourceforge.net` seems to say it would work on other UNIX systems too.

Slide 3

Environment Modules

- Basic idea is to provide a clean method for modifying, in a reversible way, environment variables such as `$PATH`.
- Command `module` (actually a shell alias) performs various functions using "module files". Environment variable `$MODULEPATH` lists directories that will be searched.

Slide 4

`module` Command Subcommands

- `avail` to list available modules; `show` or `display` to show information about a particular module.
- `list` to show currently loaded modules.
- `load` or `add` to load a module.
- `unload` or `rm` to load a module; `purge` to unload all modules.
- `help` to — what it says.
- `use` to add to the search path for modules.

Slide 5

Module Files

- What happens when you "load" a module — system processes a "module file".
- System-wide modules are (typically?) in `/usr/share/Modules/modulefiles` (ones that come with packages) and `/etc/modulefiles` (installation-specific).
- Users can have their own modules as well.
- Syntax for writing module files is — well, look at examples?

Slide 6

Linking, Revisited

- Traditional method of getting from source code to something the processor can execute involves compiling (to object code) and linking.
- Linking combines object code and (references to) libraries to produce an executable.
- Libraries can be static (code merged into executable at link time) or dynamic (code loaded at runtime, potentially shared among processes). The latter are called DLLs in Windows, shared libraries in UNIX/Linux.

Slide 7

Shared Libraries

- One attraction is somewhat obvious: If code for library functions (e.g., `printf`) is statically linked into every program that uses it, programs need more memory — seems wasteful if processes can share one copy of code in memory.
- Another attraction is that library code can be updated independently of programs that use it. (But is there a downside to that?)

Slide 8

Shared Libraries, Continued

- A good-and-bad aspect is that if the shared code is updated, all programs that use it are affected.
- How to make this happen . . . At link time, programs get “stub” versions of functions. References to real versions resolved at load time.

Slide 9

Libraries in Linux

- You may remember that (sometimes?) when you call math-library functions in C you have to compile with the extra flag `-lm`? Actually a flag to the linker `ld`. What it means . . .
- `-l`*foobar* tells the linker to try to find functions in library file `lib`*foobar*`.a` (for static linking) or `lib`*foobar*`.so` (for dynamic linking).
- Somewhat elaborate scheme for naming shared libraries allows multiple versions to coexist. Programs that use them can reference latest version (default) or specify particular version.
- References to functions in shared libraries resolved when program is loaded into memory. Can also dynamically load functions at runtime. Both depend on system being able to find shared libraries.
- Standard places to find library code, or you can explicitly specify alternate places.

Slide 10

Libraries in Linux, Continued

- Creating a static library is relatively straightforward:
- Compile code as usual and then use `ar` to combine object code files into library.
- (Example.)

Slide 11

Libraries in Linux, Continued

- Creating a shared library is less so:
- Compile code with flag to generate “position-independent code”.

  Why? Depending on platform object code and executables can include machine instructions that depend on where in memory the program is loaded. This can work for regular programs (discussed more in CSCI 2321 and CSCI 3323) but won’t work for shared libraries. “Position-independent code” avoids such instructions.
- Generate shared library and set up symbolic links following naming conventions (in which a library has a “real name”, an “soname”, and a name by which the linker normally finds it).
- At runtime, must be sure system knows where to find library. Either “hardcode” in executable or use environment variable `LD_LIBRARY_PATH`.
- (Example.)

Slide 12

Minute Essay

- Have you used environment modules (even just routinely loading one) in other classes, or for something else?