Administrivia

- Project 6 posted yesterday, due 12/10
  - public tests posted soon
- Read Chapter 7 of Bryant and O'Hallaron

Creating a static library

- To create a static library:
  - the UNIX utility `ar` creates a library from a group of object files
  - example rules in a Makefile to create a library `libavl.a` from two object files `avl.o` and `node.o`:
    ```
    LIBRARY_TO_CREATE = libavl.a
    OBJJS = avl.o node.o
    ...
    ar cru \$(LIBRARY_TO_CREATE) \$(OBJJS)
    ```
Using a static library

- To compile a program that uses a static library:
  - once a static library is created, you can add it to compilation commands for programs that use functions from the library; the library functions that are called will be linked into the application
  - suppose the program in main.c wants to use functions from the library libavl.a (which has the functions from avl.o and node.o) created above:
    ```
    gcc -o main main.o libavl.a
    ```

- To run a program that uses a static library:
  - it's a self-contained, standalone executable, so just run it (e.g., `main` in the example above).

More about shared libraries

- Standard libraries are in `/lib`, `/usr/lib`, and `/usr/local/lib`

- Standard library locations can be overridden using the environment variable `LD_LIBRARY_PATH`. It's a colon-separated list of directories (like `PATH`) that tells the linker/loader where to look for libraries.

- The UNIX utility `ldd` lists the shared libraries used by a program or shared library.

Creating a shared library

- To create a shared library:
  - use the special `gcc` flags
    ```
    -nostdlib -shared -fPIC -Wl,-soname,libraryname.so.1
    ```
    - `nostdlib` means that no standard C library is needed
    - `shared` says to generate a shared library
    - `fPIC` says to generate position-independent code
    - `Wl,-soname,libraryname.so.1` says to name the shared object `libraryname.so.1` (for whatever `libraryname` is)
  - example Makefile rules that do this, supposing we want to create a shared library `libavl.so` from two object files `avl.o` and `node.o`:
    ```
    LIBFLAGS = -nostdlib -shared -fPIC -Wl,-soname,$@.1
    
    libavl.so: avl.o node.o
    $(CC) $(LIBFLAGS) avl.o node.o -o libavl.so
    ln -s -f libavl.so.1 libavl.so
    ```

Using a shared library

- To compile a program that uses a shared library:
  - Assume the library file `libavl.so.1` is in the current directory, and the symbolic link `libavl.so` points to it, and the program in `main.c` wants to use functions from the library `libavl.so` (the functions from `avl.o` and `node.o`) created above:
    ```
    gcc -o main main.o -L. -lavl
    ```
    - the option `-L` tells the compiler to search the current directory during compilation for libraries (although not during runtime)
    - the option `-lavl` tells the compiler to look for a library file `libavl.so` (which in this case is a symlink to the actual library)
Using a shared library, con't.

- To run a program that uses a shared library:
  - setting the environment variable `LD_LIBRARY_PATH`, as in `setenv LD_LIBRARY_PATH`.
  - tells the program loader to look in a nonstandard location (the current directory) for shared libraries.
  - then just run `main` and the library is loaded when `main` begins to run (when it's first loaded into memory). Notice that the code in `avl.o` and `node.o` was never linked with the code in `main.o`, but it calls the functions in them via the shared library.

Dynamically loading a library

- C functions that support this:
  
  ```c
  void *dlopen(const char *pathname, int mode);
  
  • `pathname` is the name of a shared library
  • `mode` controls the function's operation
    
    - `RTLD_NOW`: when this shared library is loaded, indicate if there is anything that is not included which is needed immediately
    - `RTLD_LAZY`: wait and look for things only when they're actually needed from the library
  
  - returns a pointer or `handle` referring to the library, which can be used for subsequent calls to look up functions in the library.
  ```

Dynamically loading a library, con't.

```c
void *dlsym(void *handle, const char *name);

• looks up a function by name in the passed shared library
• returns a pointer to that function (or `NULL` if not found)
```

```c
int dlclose(void *handle);

• returns 0 on success
```

```c
const char *dlerror(void);

• returns a pointer to a string describing the error from the last call to any of the other functions, or `NULL` if no errors have occurred since initialization, or since it was last called
```

- To use these functions, `#include <dlfcn.h>`

Dynamically loading a library, con't.

- To compile a program that uses the above functions to dynamically load a library:
  
  ```bash
  gcc -rdynamic -ldl -o main main.c
  ```

  - for example, assume the program in `main.c` was modified to use the `dlfcn` functions above, and wants to dynamically load functions from the library `libavl.so.1` in the current directory:
  ```bash
  gcc -rdynamic -ldl -o main main.c
  ```
Dynamically loading a library, con't.

- To run a program that dynamically loads a library:
  - we again need to tell the program loader to look in the current directory for libraries, using `setenv LD_LIBRARY_PATH`.
  - then just run `main`
    - the library is opened when the program calls `dlopen()`
    - functions in it are loaded when it calls `dlsym()`, and can be executed via the returned function pointer.
  - Notice that the code in `avl.o` and `node.o` was never linked with the code in `main.o`, and `main.c` doesn't even contain regular calls to the functions, just their names in calls to `dlsym()`