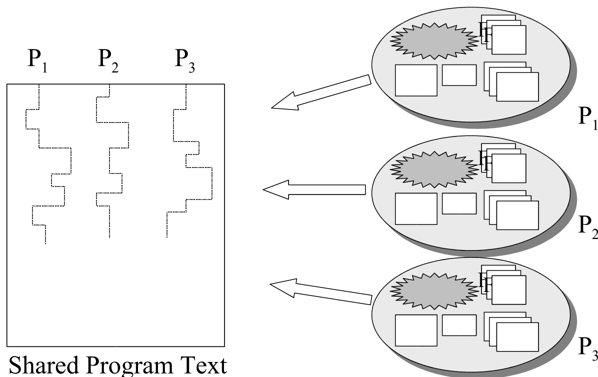
Re-entrant code and Libraries

- A library is a collection of code that implements commonly used methods or patterns with a public API. Libraries facilitate code reuse.
- Libraries can be shared (also known as dynamically linked libraries or DLLs) or be static.
- Is a shared library) part of the process or is it a resource? It should be viewed as a resource since a system utility has to find it on the fly.
- A static library, on the other hand, becomes part of the program text (at least, appropriate parts of it.)
- The concept of re-entrant code,i.e., programs that cannot modify themselves while running. Re-entrant code is necessary to write libraries.



Re-entrant Code

Useful for shared libraries (also known as Dynamically Linked Libraries or DLLs).

Creating a Shared Library

- Suppose we have three C files: `f1.c`, `f2.c`, and `f3.c` that we want to compile and add into a shared library that we will name `mylib`. First, we can compile the C files with the flags `-fPIC -shared` to the `gcc` compiler.

```
gcc  -Wall -fPIC -shared   -c -o f1.o f1.c
gcc  -Wall -fPIC -shared   -c -o f2.o f2.c
gcc  -Wall -fPIC -shared   -c -o f3.o f3.c
```

- Then we can combine, the three object files into one shared library using the `ld` linker/loader.

```
ld -fPIC -shared -o libmylib.so f1.o f2.o f3.o
```

- Now we can compile a program that invokes functions from the library by linking it with the shared library.

```
gcc test1.c -o test1 -lmylib
```

  The compiler will search for the shared library named `libmylib.so` in the current folder as well as a set of system library folders.
- If your shared library is in some other folder, you can specify that folder with the `-L` option. For example, if your library is in the sub-folder `lib` underneath the current folder, you can use

```
gcc test1.c -o test1 -Llib -lmylib
```

- When you run the executable, again the system has to be able to find the shared library. If it is not in the current folder (or installed in a system folder), then use the environment variable `LD_LIBRARY_PATH` to specify what set of folders to search in. For example:

```
export LD_LIBRARY_PATH=.:lib:$LD_LIBRARY_PATH
```

Creating a Static Library

Suppose we have three C files: `f1.c`, `f2.c`, and `f3.c` that we want to compile and add into a static library that we will name `mylib`.

First, we can compile the C files with the flag `-fPIC` to the `gcc` compiler.

```
gcc  -Wall -fPIC   -c -o f1.o f1.c
gcc  -Wall -fPIC   -c -o f2.o f2.c
gcc  -Wall -fPIC   -c -o f3.o f3.c
```

Then we can combine, the three object files into one static library using the `ar` archive program.

```
ar rcv libmylib.a f1.o f2.o f3.o
```

At this point, we can write a test program that invokes functions from the library and link it with the static library.

```
gcc -Wall -static -L. test1.c -lmylib -o test1.static
```

The rules for finding a static library are the same as for shared libraries. Note that for the above command to work, you will need to have a static version of the standard C library. You can install that with the command (on Fedora Linux):

```
yum install glibc-static
```

How to check for library dependency?

- **Linux**: Use the tool ldd.
- **MS Windows**: Use the tool depends (available from `http://www.dependencywalker.com`).
- **MacOSX**: Use the tool otool.

Plugins

Plugins are pieces of code that be loaded into or unloaded from a program upon demand without having to restart the program.

Device drivers are a type of plugin for the operating system that deals with hardware devices.

Examples of plugin use: Web browsers, Windows Media Player, Amarok, Eclipse etc.

Plugin Example

```
/* lab/plugins/ex1/runplug.c */
#include <stdio.h>
#include <string.h>
/* dll include file */
#include <dlfcn.h>
#define MAX_BUF 1024
/* dll variables */
void *handle;              /* handle of shared library */
void (*function)(void); /* pointer to the plug-in function */
const char *dlError;     /* error string */

int main(int argc, char **argv)
{
  char buf[MAX_BUF];
  char plugName[MAX_BUF];

  while (1) {
    /* get plug-in name */
    printf("Enter plugin name (exit to exit): ");
    fgets(buf, MAX_BUF, stdin);
    buf[strlen(buf)-1] = '\0';       /* change \n to \0 */
    sprintf(plugName, "./%s", buf);  /* start from current dir */
    /* ... next page ... */
```

Plugin Example (contd.)

```
    /* checks for exit */
    if (!strcmp(plugName, "./exit"))
      return 0;
    /* open a library */
    handle = dlopen(plugName, RTLD_LAZY);
    if ((dlError = dlerror())) {
      printf("Opening Error: %s\n", dlError);
      continue;
    }
    /* loads the plugin function */
    function = dlsym( handle, "plugin");
    if ((dlError = dlerror()))
      printf("Loading Error: %s\n", dlError);
    /* execute the function */
    (*function)();
    if ((dlError = dlerror()))
      printf("Execution Error: %s\n", dlError);
    /* close library 1 */
    dlclose(handle);
    if ((dlError = dlerror()))
      printf("Closing Error: %s\n", dlError);
  }
  exit(0);
}
```

Plugin Example (contd.)

```
/* lab/plugins/ex1/plugin1.c */
#include <stdio.h>
void plugin(void)
{
  printf("This is plug-in 1\n");
}
```

```
/* lab/plugins/ex1/plugin2.c */
#include <stdio.h>

void plugin(void)
{
  printf("This is the second plug-in\n");
}
```

Plugin Example (contd.)

```
[amit@kohinoor ex1]$ ls
Makefile  plugin1.c  plugin2.c  runplug.c
[amit@kohinoor ex1]$ make
gcc runplug.c -g   -ldl -o runplug
gcc -fpic -shared plugin1.c -o plugin1.so
gcc -fpic -shared plugin2.c -o plugin2.so
```