COMP206-Library

An object file is the compiled result of one source file. There is still much work to produce running code: − Combine together with other source files or objects

− Resolve external references

*Libraries are the “completed” result of compilation: fully-resolved byte-code that’s ready to run on the system.* We will see two types of libraries:

− Static (.a)

− Shared (.so)

**[Linux Library Conventions]**

* All libraries start with “lib”
* Followed by the name of the library
* Followed by an extension:

• “.a” for static libraries

• “.so” for shared libraries

* You can see many of the default Linux libraries by looking in:

• /lib

• /usr/lib

• /usr/local/lib

**[Static and Shared Libraries]**

The difference is when the library byte-code is merged with the dependent program:

* Static libraries are copied by the linker into the executable

− Program now contains the library internally

* Shared libraries remain in separate files always

− Loaded dynamically as features are needed

* Create and Use A Static Library

− gcc -c swap.c (produces swap.o from swap.c)

− ar rcs libswap.a swap.o (produces libswap.a from swap.o)

− gcc main.c libswap.a OR

− gcc main.c -L. –lswap (CAUTION,ORDERMATTERS!)

*Quite similar to compiling against a “.o” file, just a bit more work is done in advance*

* Create and Use a Shared (dynamic) Library

− gcc -shared -fpic -o libswap.so swap.c

− gcc main.c -lswap -L.

At first your a.out will not be run-able. To fix this temporarily:

− export LD\_LIBRARY\_PATH=${LD\_LIBRARY\_PATH}:.

Environment variable for where to find .so (similar to PATH, which locates programs)

**[Linux Environment Variables]**

Recall: the shell is itself a programming language, just targeted to help you run and coordinate other programs. Its variables can be used to do math, manipulate data etc, but they often also effect the way other programs are run

* To set an environment variable for the remainder of the terminal session:

$ export <VARIABLE\_NAME>=<VALUE>

* To check or use an environment variable’s value, use

$<VARIABLE\_NAME>

Example: $ echo $LD\_LIBRARY\_PATH

* LD\_LIBRARY\_PATH

*A special variable whose job is to hold all the locations where shared libraries exist.* When you run a program, like a.out, ls, vim or any other, the shell searches LD\_LIBRARY\_PATH in order for every required .so and uses the first it finds.

Our command to let your program run with the newly created “.so” in the present working directory was export LD\_LIBRARY\_PATH=${LD\_LIBRARY\_PATH}:.

Note, “.” is the path we add on to the colon-separated list. Means “here!”

In order to set LD\_LIBRARY\_PATH permanently, add this line to “~/.bashrc”, the file that is run each time your terminal starts.

Likely use the full path instead of “.”

* LIBRARY\_PATH

*Similar to “-L” on the gcc command-line. Where are the libraries we want to compile against.*

Must find libraries listed with “-l<name>” in this list

Note that directories given with –L explicitly are searched before LIBRARY\_PATH

* CPATH

*Where to search for header files. Similar to “-I” (capital i) gcc flag*

**[Library Related Tools]**

* ldd - print shared object dependencies
* nm - list symbols from object files

Try these on the .a, .so and a.out produced during the two different compilation options shown above.

Pointers to functions

int \*fn(); this means a function that returns an int\*

int (\*fn)(); this means a pointer to a function that returns an integer and takes no arguments

The second creates a function pointer, which can be used to “point to” any existing function that matches it’s return and argument types, like this:

int return5() { return 5; } int (\*fn)() = return5;

Afterwards, the word fn is a valid way to execute the code in the return5 function (until fn might be pointed somewhere else later, which is OK):

int x = (\*fn)(); // x will equal 5

int y = return5(); // y will return 5

int z = fn(); // this syntax is fine too.

Provides an additional level of abstraction. You can pass around the pointer and use it to do different operations depending on context

Example: sorting integers vs sorting strings int (\*comparison)( void\*, void\* );

// Comparison can point to either

int compare\_strings( void\* a, void\* b ){ return strcmp( (char\*)a, (char\*)b); }

// OR

int compare\_ints( void\* a, void \*b ) { return a!=b; }

NOTE: The two functions must have the same specification, but can differ in implementation!

Buffer

System Calls

We so far seen how one core C function utilizes system calls to get its job done:

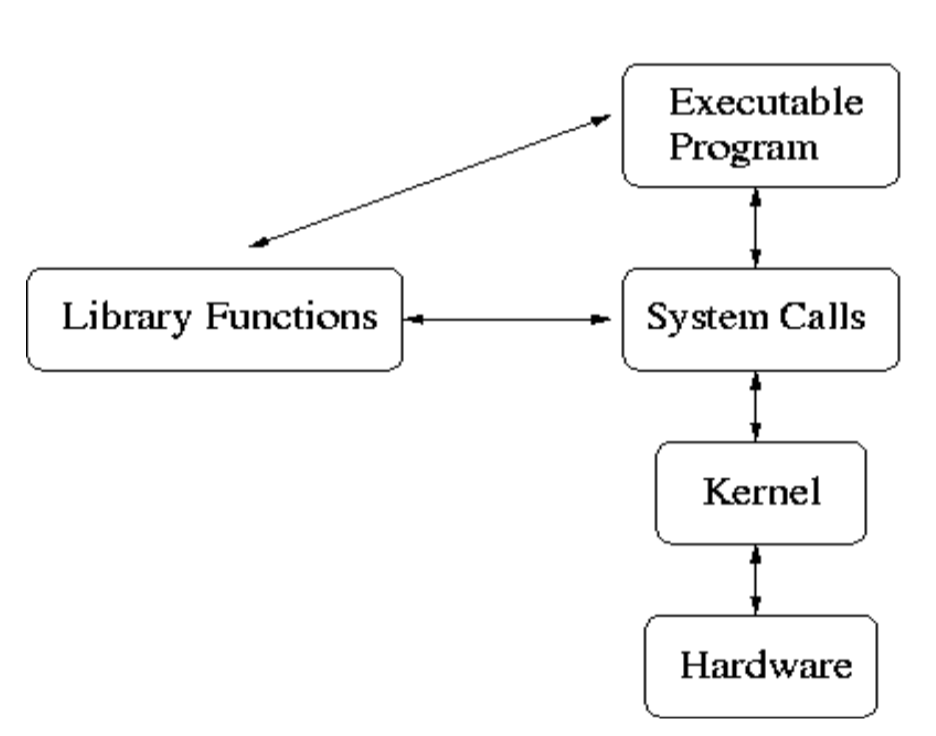
malloc : sbrk

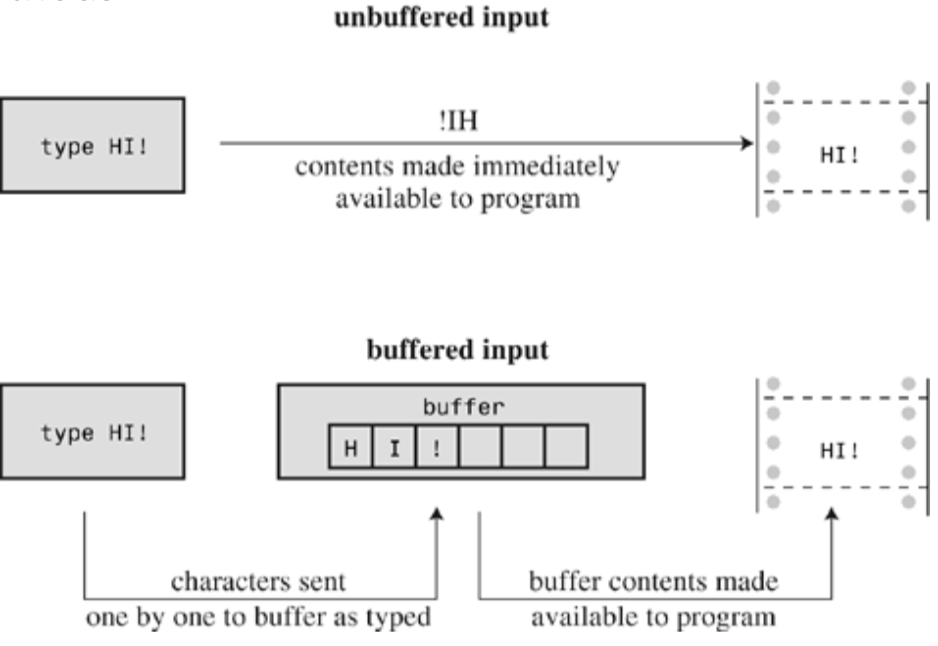
The same story holds for all other C functions that conduct system interaction:

fopen : unlink, access

fread and fwrite : read and write

In each case, we can ask what the C library does for us to know when to go directly to the system without our program slowing to a crawl indicates some "buffering" has happened





**Effects of Buffered IO**

* Keyboard mode for standard input:

The user types many characters, but is able to "go back" by pressing backspace or delete. Our program does not see these interactions, but only gets the full line when enter is pressed

* Efficiency for filesystem interactions:

Reading and writing to the hard drive is "slow" (relative to memory or CPU operations). The fact that we can interact with data byte by byte without our program slowing to a crawl indicates some "buffering" has happened

[Buffered VS Unbuffered IO]

