

COMP 206 – Software Systems

Lecture 15 – “Advanced” C topics

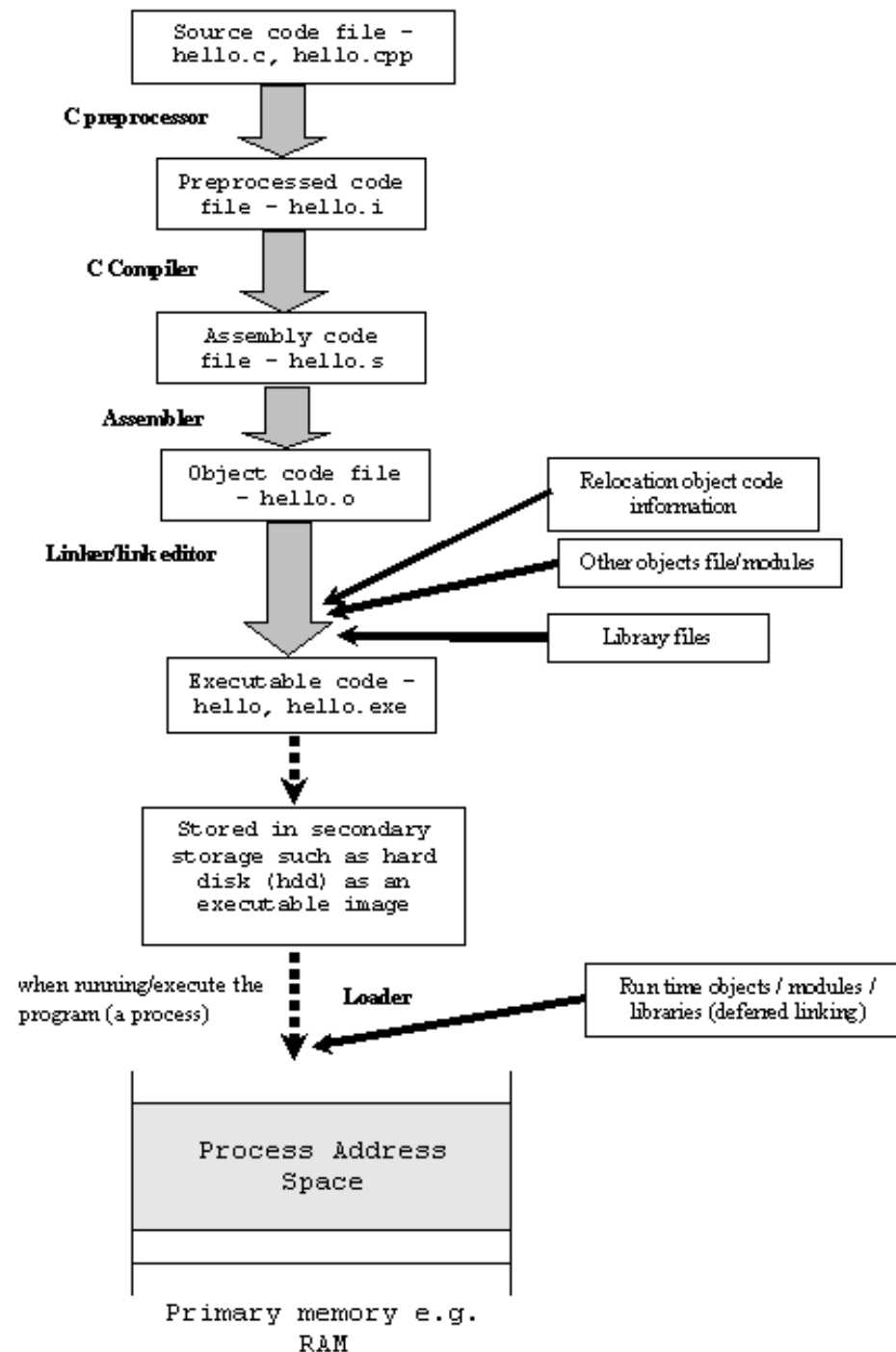
Last week of Oct, 2018

Today's outline

- Using and creating C libraries
- Function pointers
- System calls

Recall: the C Program Creation Pipeline

- Last time: stop at .o file you have done most of the work for one file only
- What about larger groups of code like all of the filesystem helpers within the kernel?
 - For this we need libraries



Introduction to Libraries

- 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
- Can you imagine the pros and cons of these approaches?


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, ORDER MATTERS!)

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`

Setting LD_LIBRARY_PATH

- We just saw the example, LD_LIBRARY_PATH is 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 “.”

Other important environment variables for coding

- `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.

Exercises

- Clone the Lectures Github repository and try compiling the “swap” functionality into an object, a static library and a shared library, and then build the main each way.

Writing Safe C programs

- As a software systems programmer, you must think about how to make your program meet its specifications even when the users and other programs it interacts with attempt to misbehave.
- Let's think about writing safe, secure C code....

~~Writing Safe C programs~~ Hacking C Code!

- ~~• As a software systems programmer, you must think about how to make your program meet its specifications even when the users and other programs it interacts with attempt to misbehave.~~

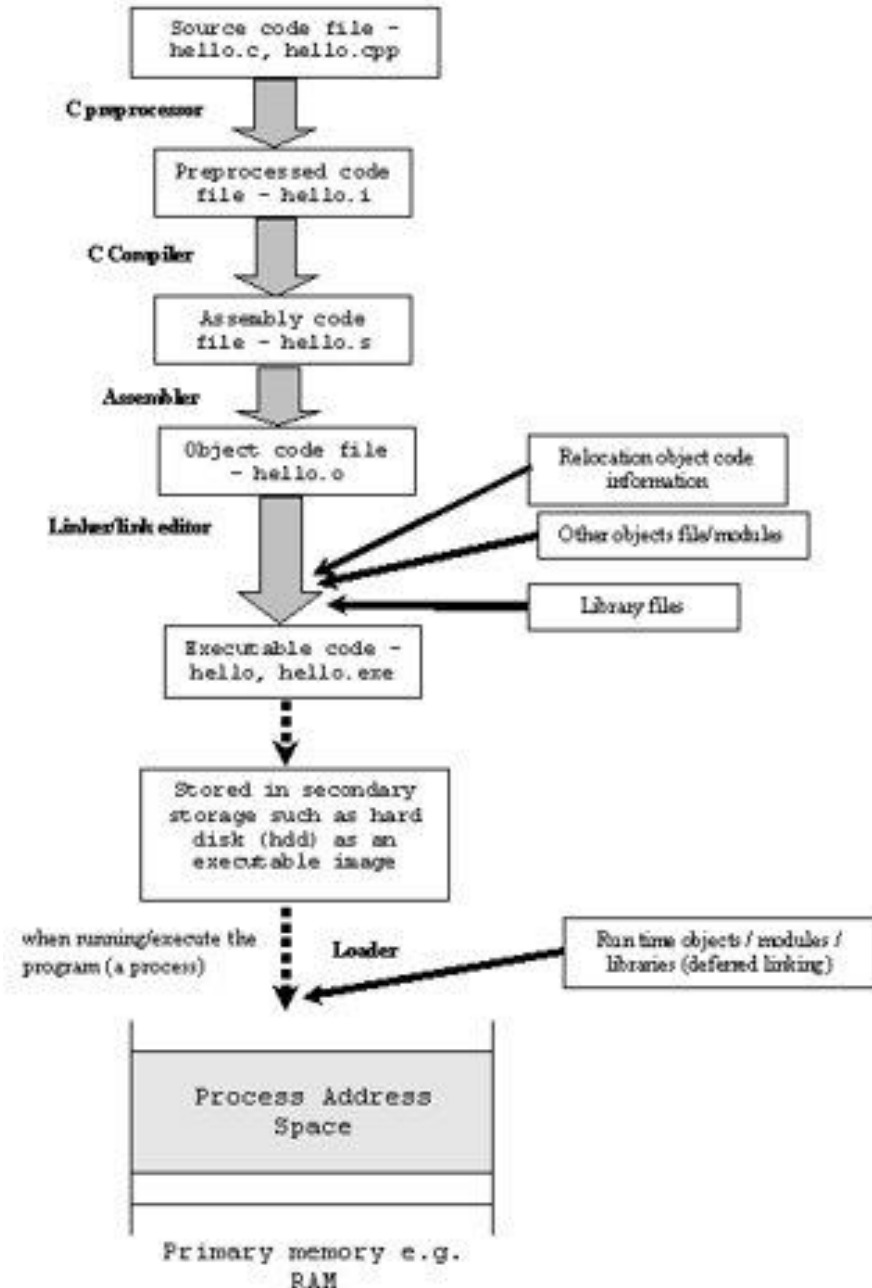
BORING!

- ~~• Let's think about writing safe, secure C code....~~

Or... we could just take over!

Exploit #1, over-write C's library functions

- Recall: dynamic libraries are not a part of the executable, but rather stored separately in "shared object" (.so) files
- If a C program depends on any external functions, we can find out what these are and replace them with our own version

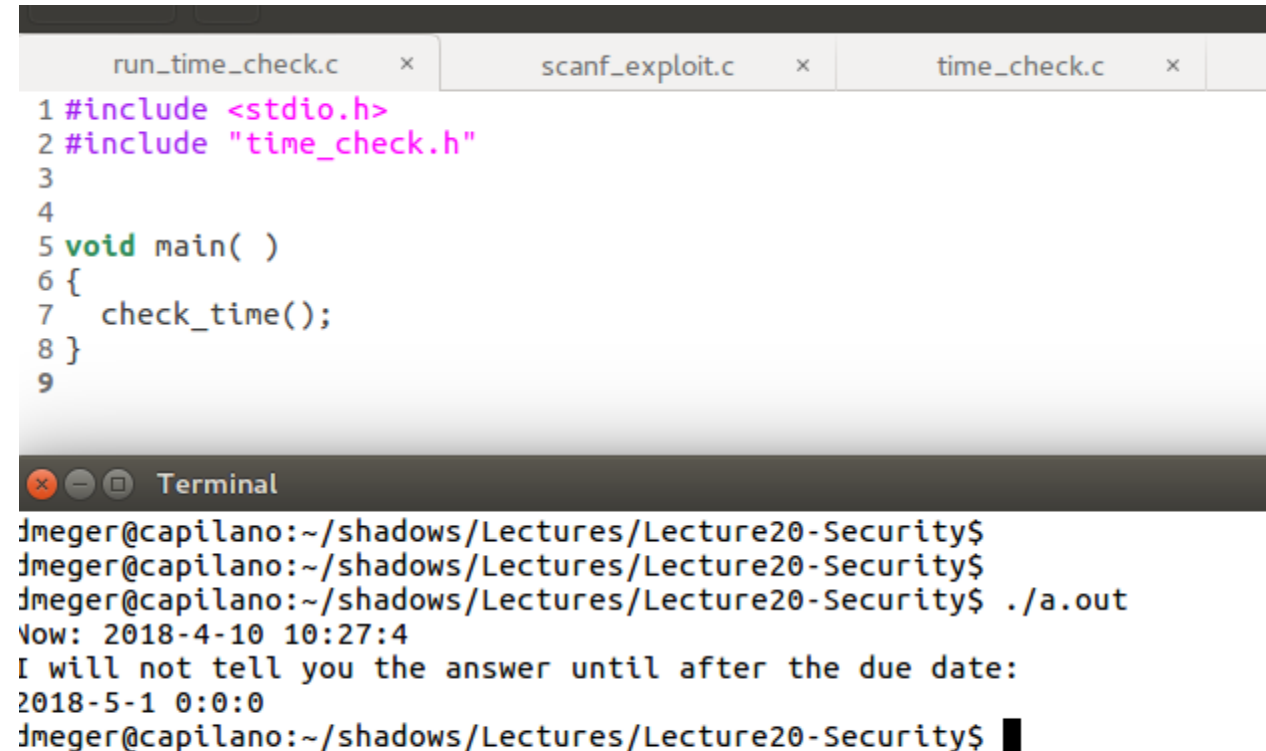


Over-writing Library Functions

- Where do they live? `$ locate libc.so`
 - You may already have multiple versions. We're not the first ones to have this idea, and many real programs install their own libc because they want to standardize a certain version or customize functionality.
 - Typically in `/lib`, `/usr/lib`, `/usr/local`
- So, which one is used?
 - The `LD_LIBRARY_PATH` environment variable lists paths in order. The first one that contains the needed library is used.
 - For any program, we can tell this with `$ ldd <program_name>`

"Time" for an example

- Suppose you receive a program that locks until a certain date
- For example, the code on the Lectures repository of GitHub will tell you the answer to the final exam... but only on Jan 1st, 2019



```
run_time_check.c x scanf_exploit.c x time_check.c x
1 #include <stdio.h>
2 #include "time_check.h"
3
4
5 void main( )
6 {
7     check_time();
8 }
9

Terminal
jmegeer@capilano:~/shadows/Lectures/Lecture20-Security$
jmegeer@capilano:~/shadows/Lectures/Lecture20-Security$
jmegeer@capilano:~/shadows/Lectures/Lecture20-Security$ ./a.out
Now: 2018-4-10 10:27:4
I will not tell you the answer until after the due date:
2018-5-1 0:0:0
jmegeer@capilano:~/shadows/Lectures/Lecture20-Security$
```

Let's see how the code is checking time

- The a.out depends on a shared object that hides the actual code:

```
$ ldd a.out
```

```
linux-vdso.so.1 => (0x00007fff9eac9000)
```

```
libtime_check.so => ./libtime_check.so (0x00007f414bc78000)
```

```
libc.so.6 => /lib/x86_64-linux-gnu/libc.so.6 (0x00007f414b8ae000)
```

```
/lib64/ld-linux-x86-64.so.2 (0x00007f414be7a000)
```

- However, we can see what that shared object does using "nm":

```
$ nm libtime_check.so
```

```
<many lines to ignore, but some important ones...>
```

```
U localtime@@GLIBC\_2.2.5
```

```
U printf@@GLIBC\_2.2.5
```

```
U puts@@GLIBC\_2.2.5
```

```
U \_\_stack\_chk\_fail@@GLIBC\_2.4
```

```
U time@@GLIBC\_2.2.5
```



This is our exploit
(key-hole)

Creating our key, need to know the spec

C library function - time()

Advertisements

⊕ Previous Page

Next Page ⊕

Description

The C library function **time_t time(time_t *seconds)** returns the time since the Epoch (00:00:00 UTC, January 1, 1970), measured in seconds. If **seconds** is not NULL, the return value is also stored in variable **seconds**.

Declaration

Following is the declaration for time() function.

```
time_t time(time_t *t)
```

Parameters

- **seconds** – This is the pointer to an object of type time_t, where the seconds value will be stored.

Return Value

The current calendar time as a time_t object.

Example

The following example shows the usage of time() function.

So, we must simply write our own "time()"

- This key is extremely simple.
 - time_t is simply a long int.
 - Returning any large number would work
 - We can do some quick math to get a sensible value, 1 second after the deadline
- Now we need to compile our code and ask our a.out to use it

```
1 #include <time.h>
2
3 time_t time( time_t *seconds )
4 {
5     return 1527825601;
6
7 }
8
9
```

Turning the key: LD_PRELOAD

- Recall: compiling to a .so file will allow our code to be dynamically loaded by the program
- We can force it to be prioritized over the standard C time() implementation with the LD_PRELOAD environment variable

```
$  
$ gcc -shared -fpic time_hack.c -o libtime.so  
$ LD_PRELOAD=./libtime.so; ./a.out  
Now: 2018-6-1 0:0:1  
The answer is 42.  
$ □
```

The answer!



Recall: The gdb debugger

- The GNU debugger, available everywhere that gcc exists
- Running with `gdb <prog_name>` loads the program and allows running with assistance and introspection
- Command-line user interface. Some key commands (see help for more):
 - `run <args>`
 - `break <line# or function name>`
 - `list`
 - `backtrace`
 - `print <var>`
 - `disp <var>`
 - `next`
 - `nexti`
 - `finish`



GDB

What does gdb tell us about programs?

- They can be run by more than just the operating system
 - GDB loads your program and runs it in a different way than usual
 - It captures the terminal output, steps the code one line at a time, starts and stops the program, can even change data as you become very experienced
 - In a way “owns” your programs execution -> so where is the OS in this process?
- How does this all work?
 - Some more answers throughout the term, for now, just something to think over

Pointers to functions

All of a program's code lives in memory

C gives us access to memory using pointers

In addition to “pointing” to data, we can also point to functions (yes really!)

These include those we code ourselves, as well as built-in functions in libraries

Pointers to functions

Know the Difference Between These Declarations:

<code>int *fn();</code>	this means a function that returns an int*
<code>int (*fn)();</code>	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 its 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.
```

A full C file example

```
#include <stdio.h>
void add(int a, int b){
    printf("%d\n", (a+b) );
}

int main(void)
{
    void (*p) (int,int);

    p=&add;
    p(5,5);

    return 0;
}
```

Why have function pointers?

- 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
```

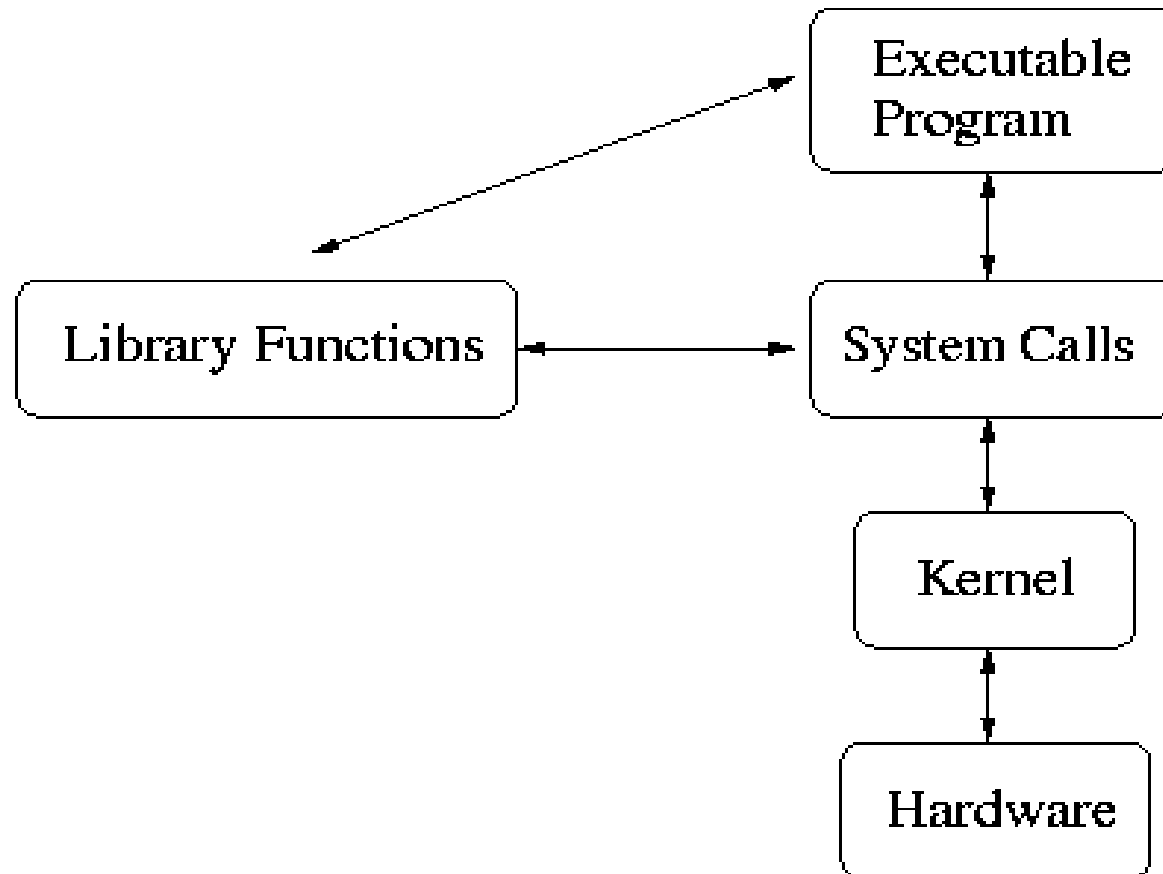
```
int compare_strings( void* a, void* b ){ return strcmp( (char*)a, (char*)b); }
```

```
// 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!

Performing Low-Level Interactions



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:

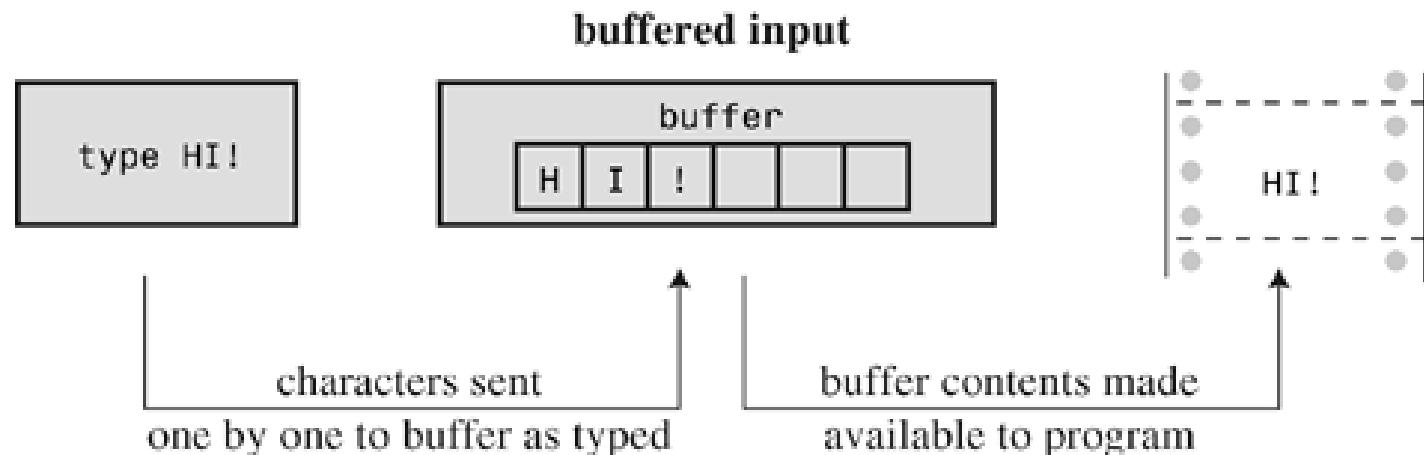
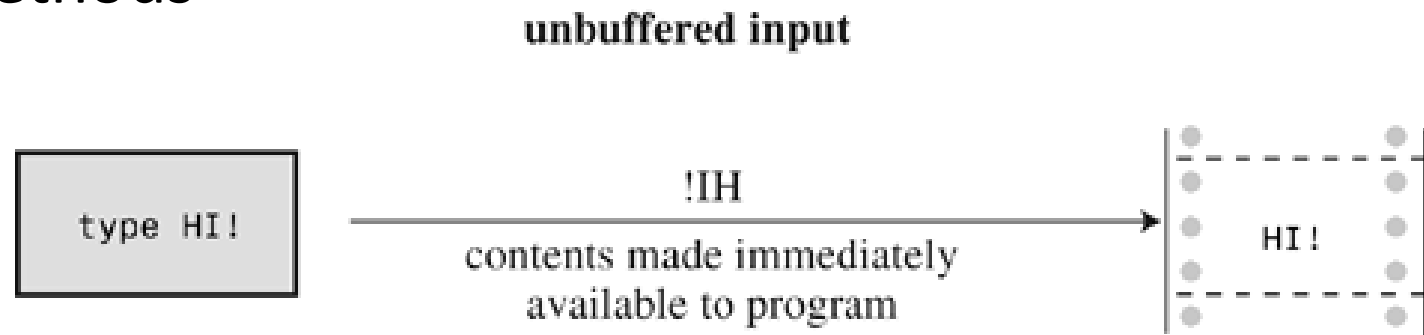
fopen : unlink, access

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

A common case: Buffered vs Un-buffered IO

- Each of the standard C IO methods we saw uses buffering
 - Why?

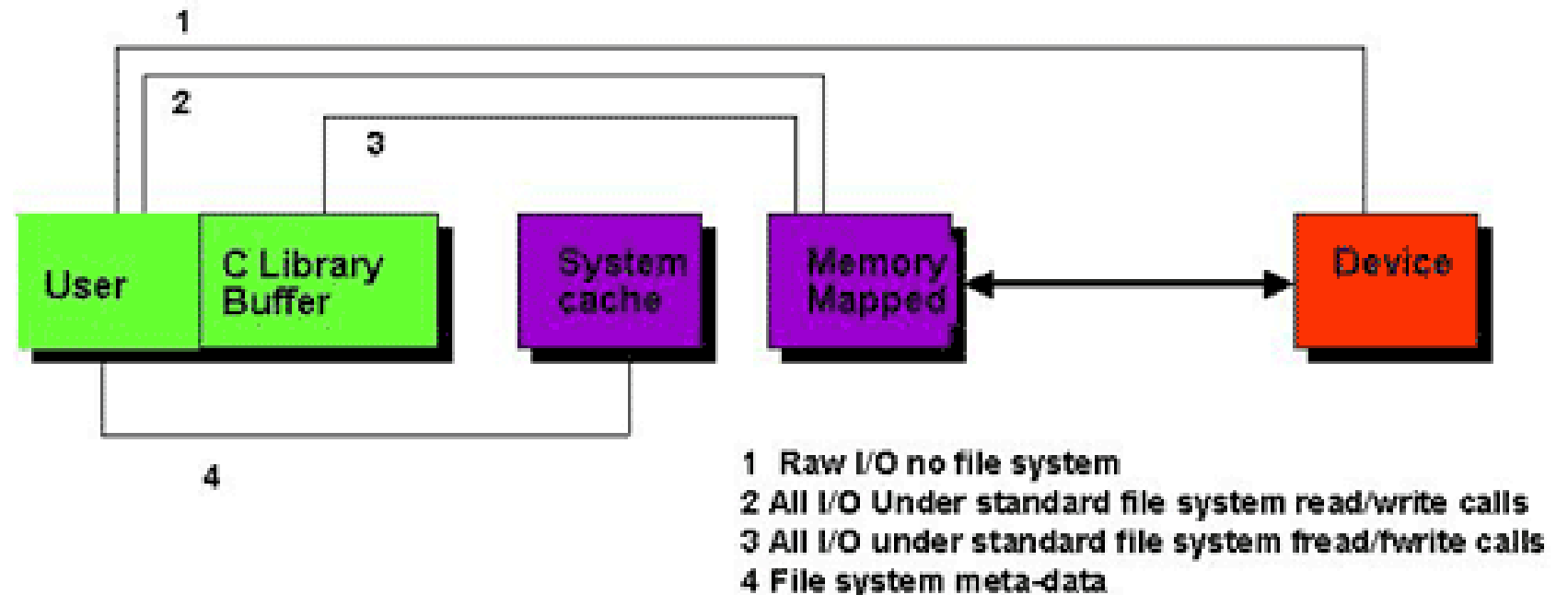


Effects of Buffered IO

- 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 Un-buffered IO

	Buffered-Formatted	Buffered-Unformatted	Unbuffered
C	<code>fprintf()</code> <code>fscanf()</code>	<code>fread()/fgets()</code> <code>fwrite()/fputs()</code>	<code>read()/getc()</code> <code>write()/putc()</code>



Exercises for today

- Practice making libraries:
 - Go back to the swap examples where we created and used a .o. Ensure you can build the same code by making a static or dynamic library.
 - Take any C code you've written previously, split it up, build part into a library, make sure you can compile the whole thing together.
- Challenge with function pointers:
 - Write a sort function that accepts a function pointer called "compare"
 - Call it once with the pointer set to the function "less than" that you have written, see that you sort in ascending order
 - Call again with the pointer set to the function "greater than", again written by you. Now you should see descending order.
- Not that easy to practice with system calls yet. This is mostly "knowledge" for 206.