Exercise Session 8

Systems Programming and Computer Architecture

Fall Semester 2024

Disclaimer

- **Website**: n.ethz.ch/~falkbe/
- **(Extra) Demos on GitHub**: github.com/falkbe
- My exercise slides have **additional slides** (which are not official part of the course) **having a blue heading**: they are there to complement and go into more depth where I found appropriate
- For the exam **only** the official exercise slides are relevant, if in doubt always check the ones on the official moodle page

Agenda

- Assignment 5 (Bomb Lab)
  - Questions
- UNIX File System, Paths and Command Line Recap
- Linking Theory (Static and shared libraries)
- Linking
- - Theory recap
  - Library linking demo
- Assignment 6
  - Introduction
- Exam problem example

UNIX FHS

Filesystem, Paths, Binaries etc.

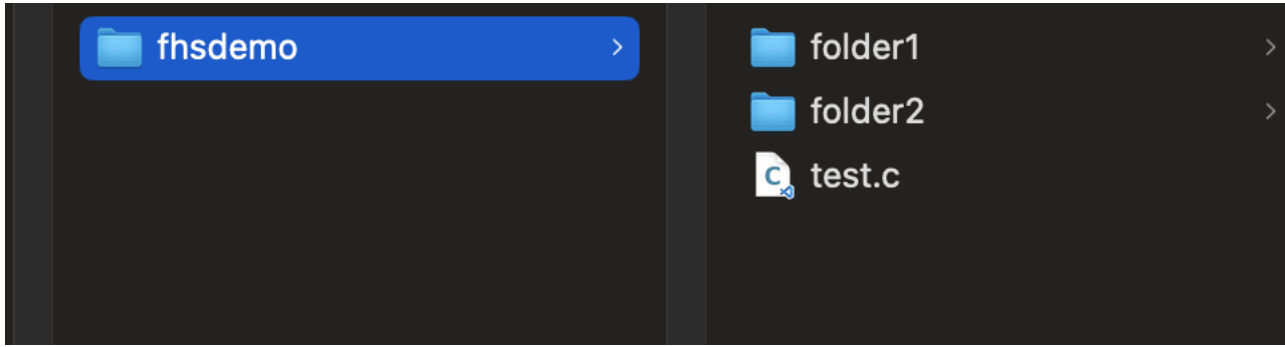
Why look UNIX FHS again?

Systems@ETH Zürich

- **Fundamental** to understand, not really taught in course though
- **Will** help you a lot later on, although not super relevant for the exam its **super relevant for other courses, understanding and praxis**

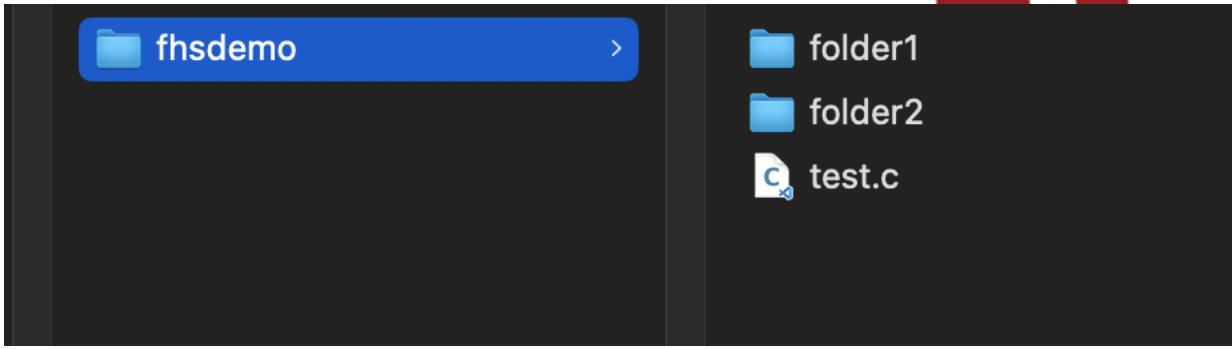
UNIX FHS

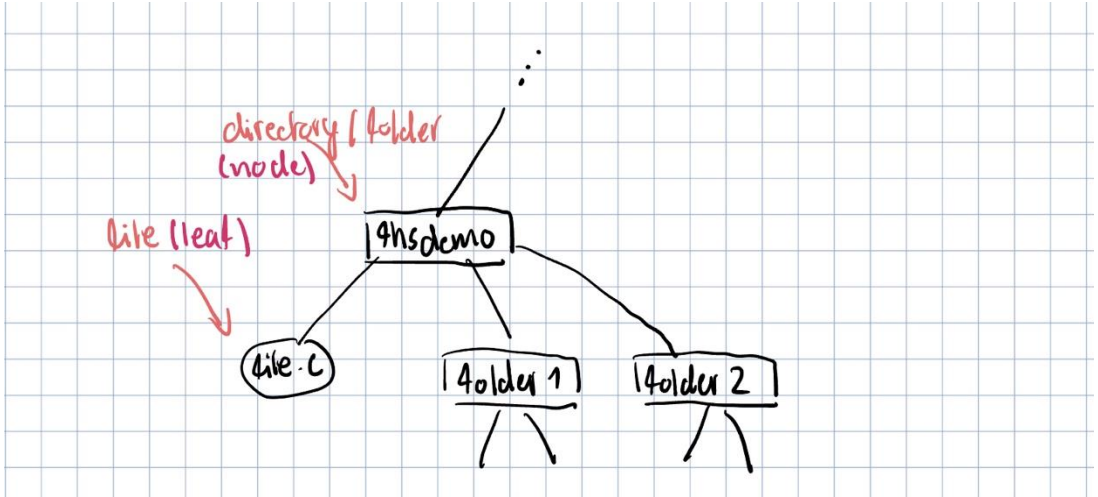
- Looked at one/multiple C files, mostly in one directory (Folder)



UNIX FHS

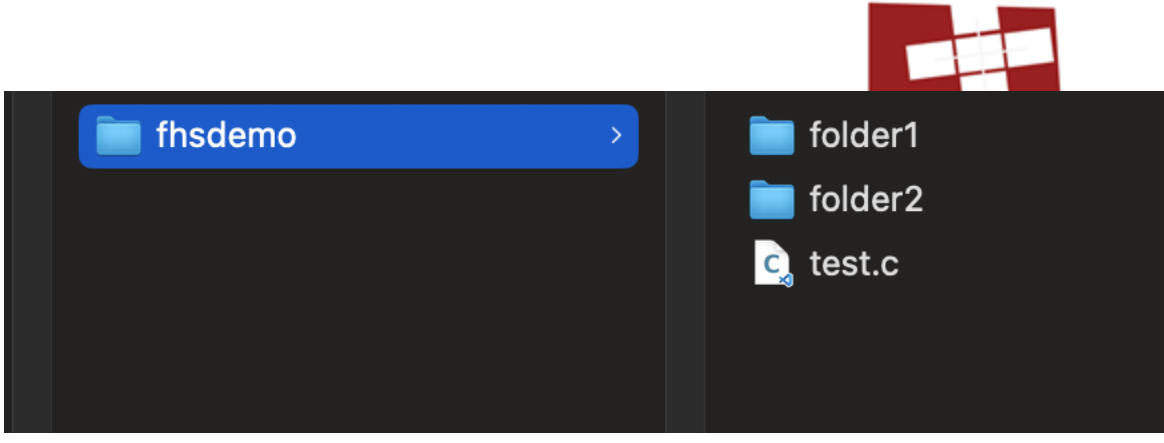
- Linux FHS is a Graph: you are just at a particular node

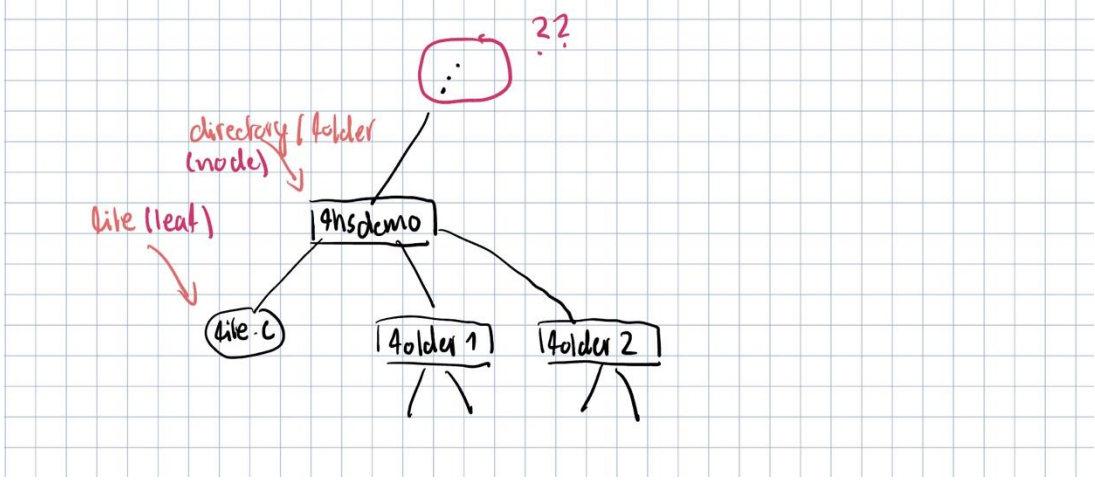




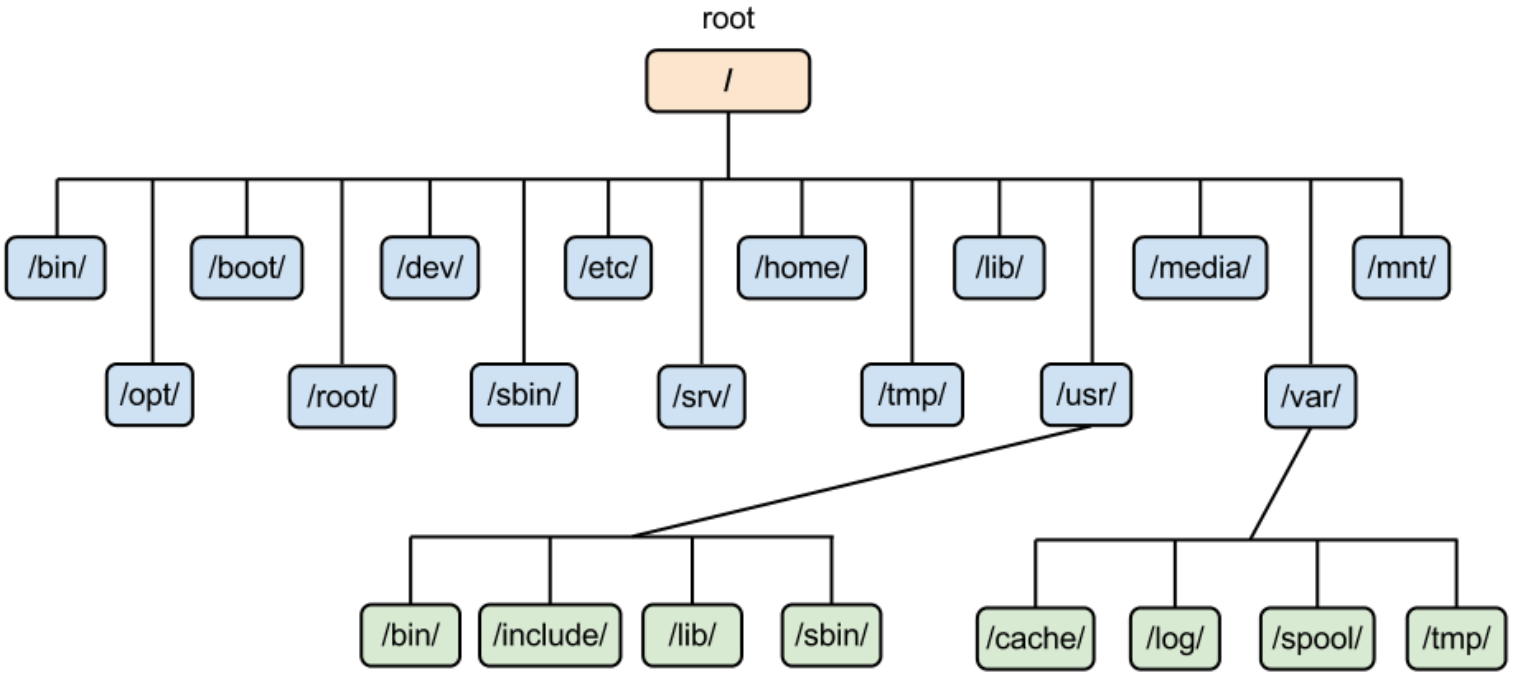
UNIX FHS

- Linux FHS is a Graph: you are just at a particular node
- What is at “...”?

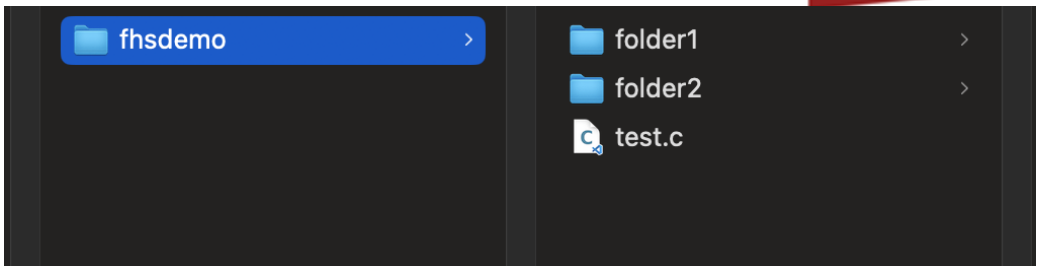




UNIX FHS

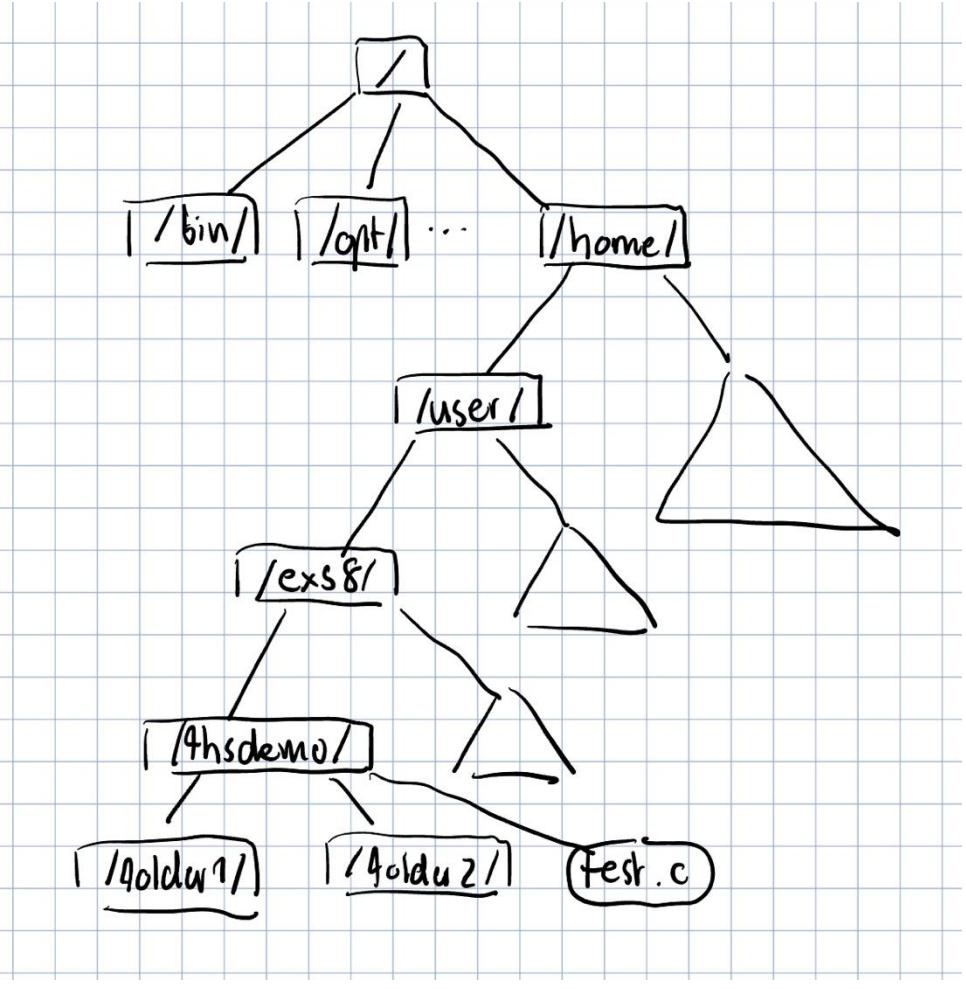

root
/
/bin/
/boot/
/dev/
/etc/
/home/
/lib/
/media/
/mnt/
/opt/
/root/
/sbin/
/srv/
/tmp/
/usr/
/var/
/bin/
/include/
/lib/
/sbin/
/cache/
/log/
/spool/
/tmp/

UNIX FHS

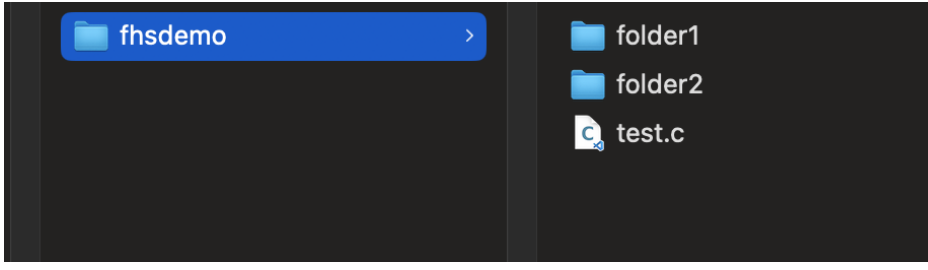


- Terminal?
- “**pwd**” shows where you are in the graph

```
user@4865b4f533e3:~/exs8/fhsdemo$ pwd
/home/user/exs8/fhsdemo
```



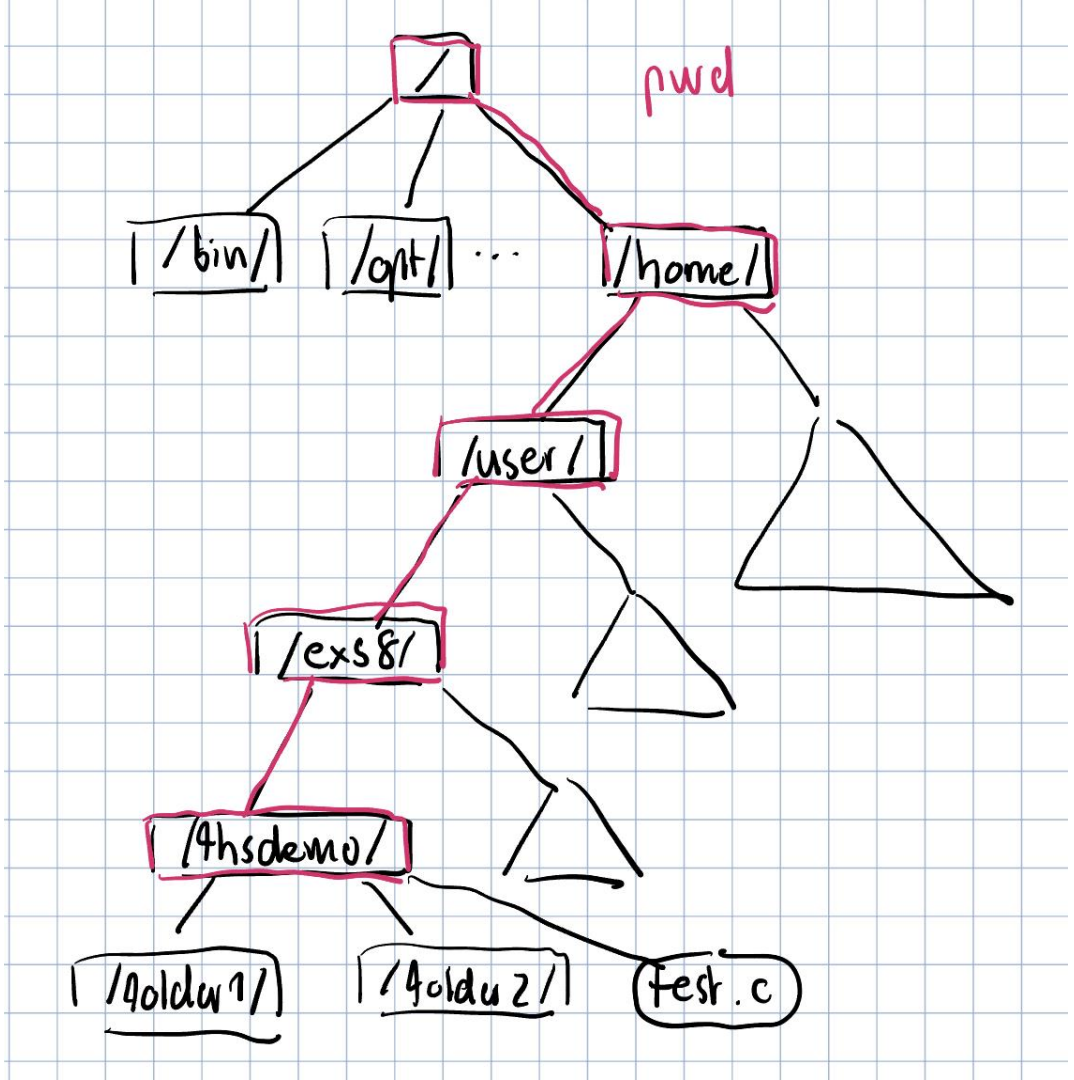
Systems@ETH Zürich




```
user@4865b4f533e3:~/exs8/fhsdemo$ pwd
/home/user/exs8/fhsdemo
```

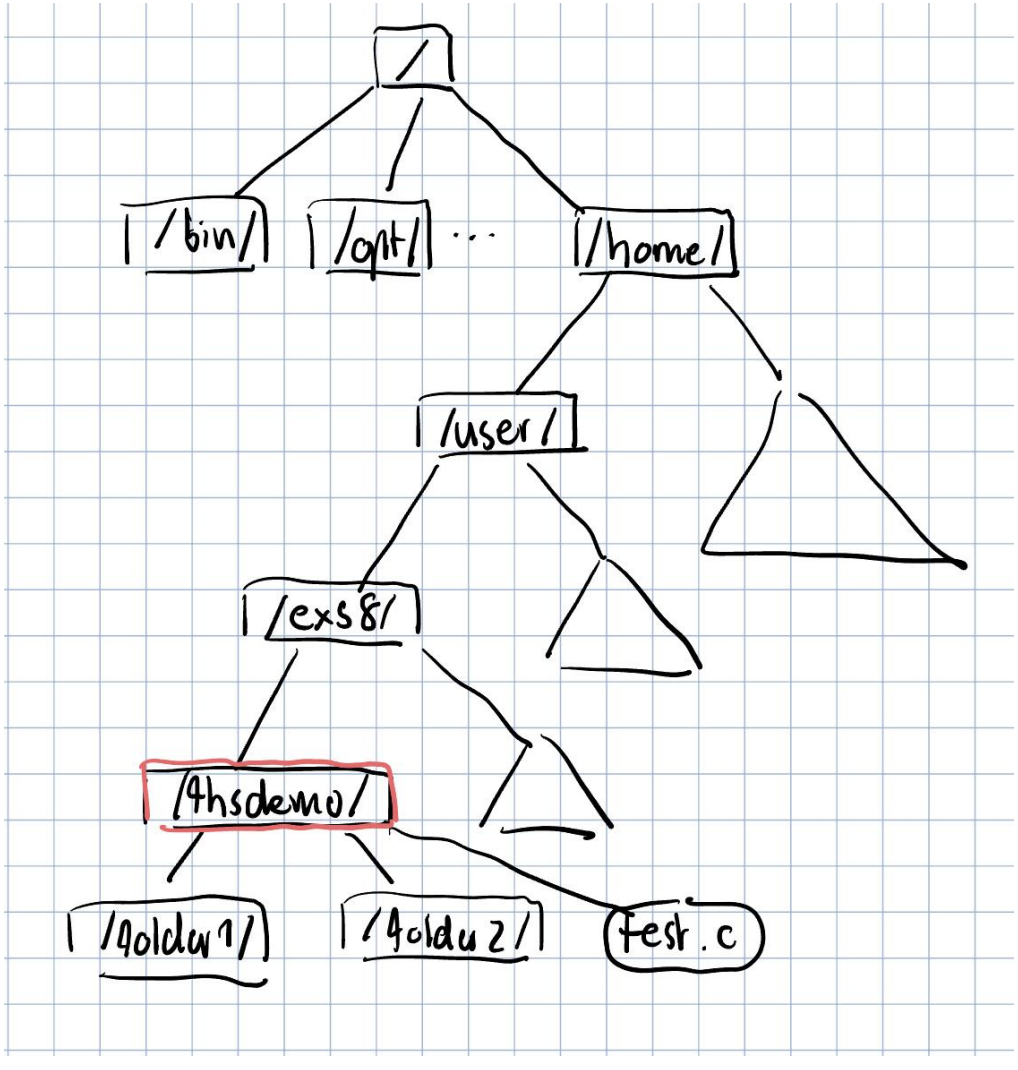
UNIX FHS

- **Pwd** shows you current path, by concatenating names of nodes to where you are



UNIX FHS

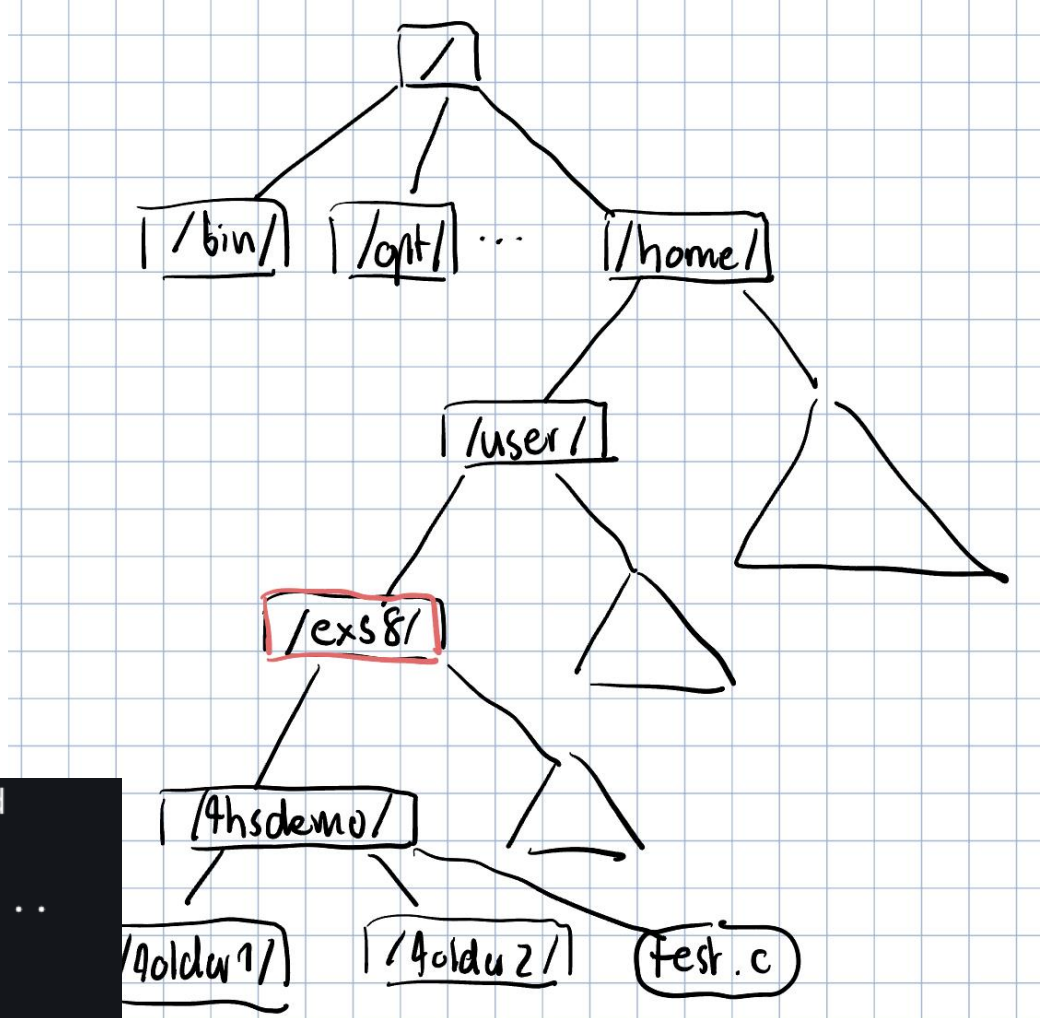
- **You** are always somewhere in this graph, the current **working directory**
- Refer to your own position as “.”



UNIX FHS

- You can move relative to your current position
- **Up**: “cd ..”
- Now check pos with “pwd”

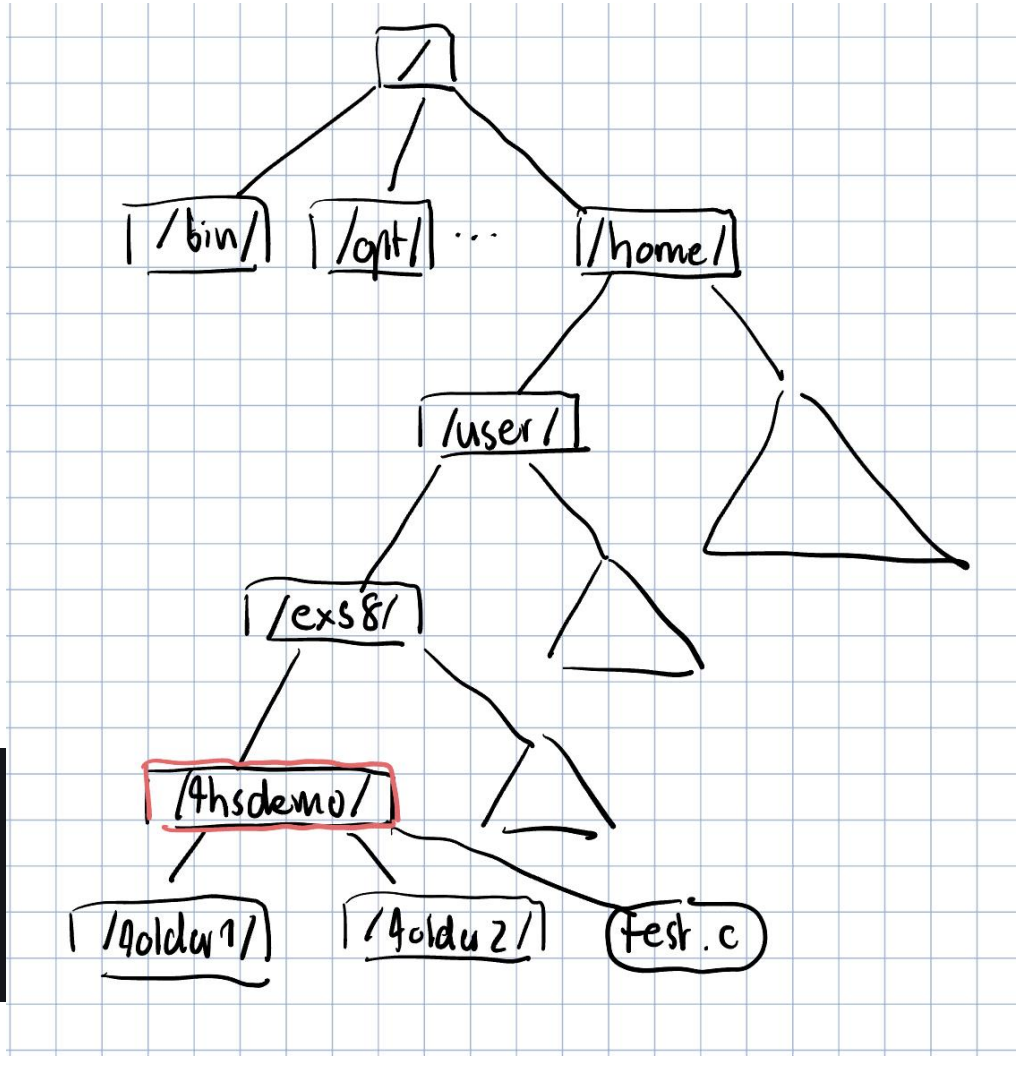
```
user@4865b4f533e3:~/exs8/fhsdemo$ pwd
/home/user/exs8/fhsdemo
user@4865b4f533e3:~/exs8/fhsdemo$ cd ..
user@4865b4f533e3:~/exs8$ pwd
/home/user/exs8
```



UNIX FHS

- You can move relative to your current position
- **Up**: “cd ..”
- **Down**: “cd fhsdemo/”
- Now check pos with “pwd”

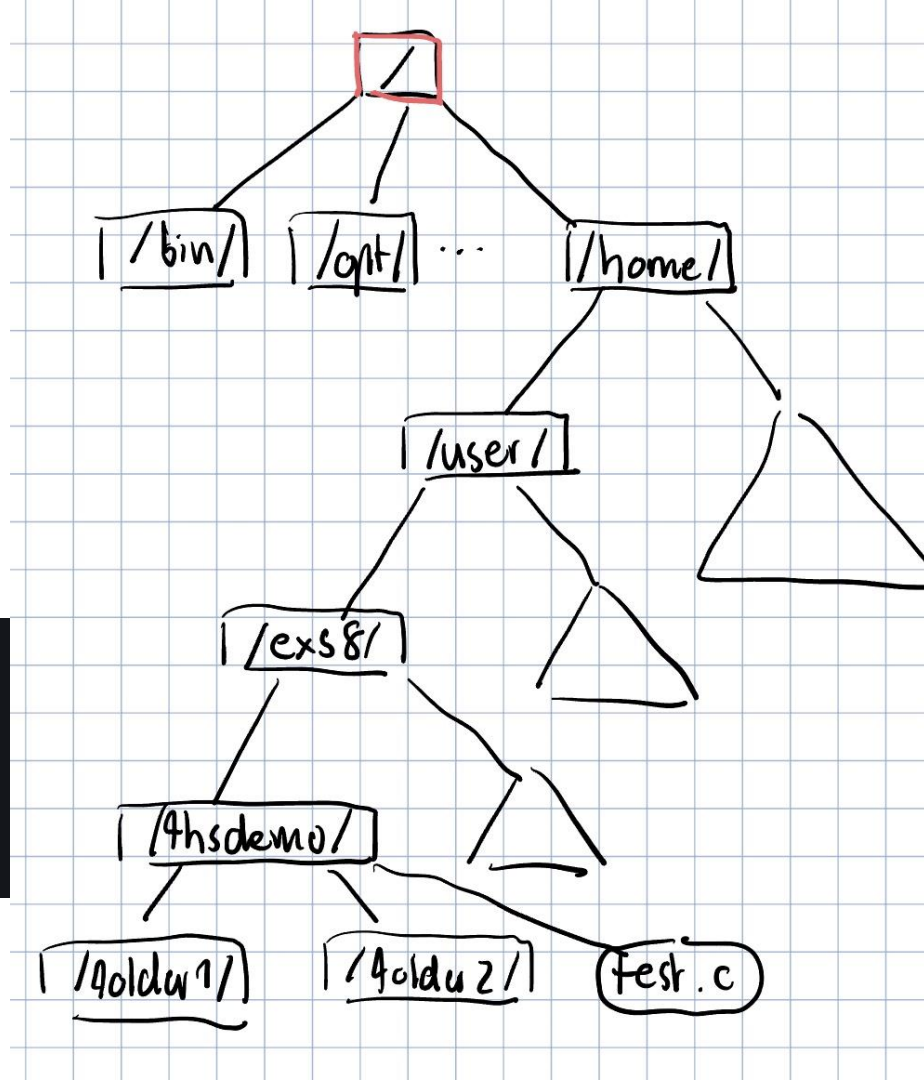
```
user@4865b4f533e3:~/exs8$ pwd
/home/user/exs8
user@4865b4f533e3:~/exs8$ cd fhsdemo/
user@4865b4f533e3:~/exs8/fhsdemo$ pwd
/home/user/exs8/fhsdemo
```



UNIX FHS

- You can move to a **absolute position**
- **“cd <absolutepath>”**
- Example: “cd /"

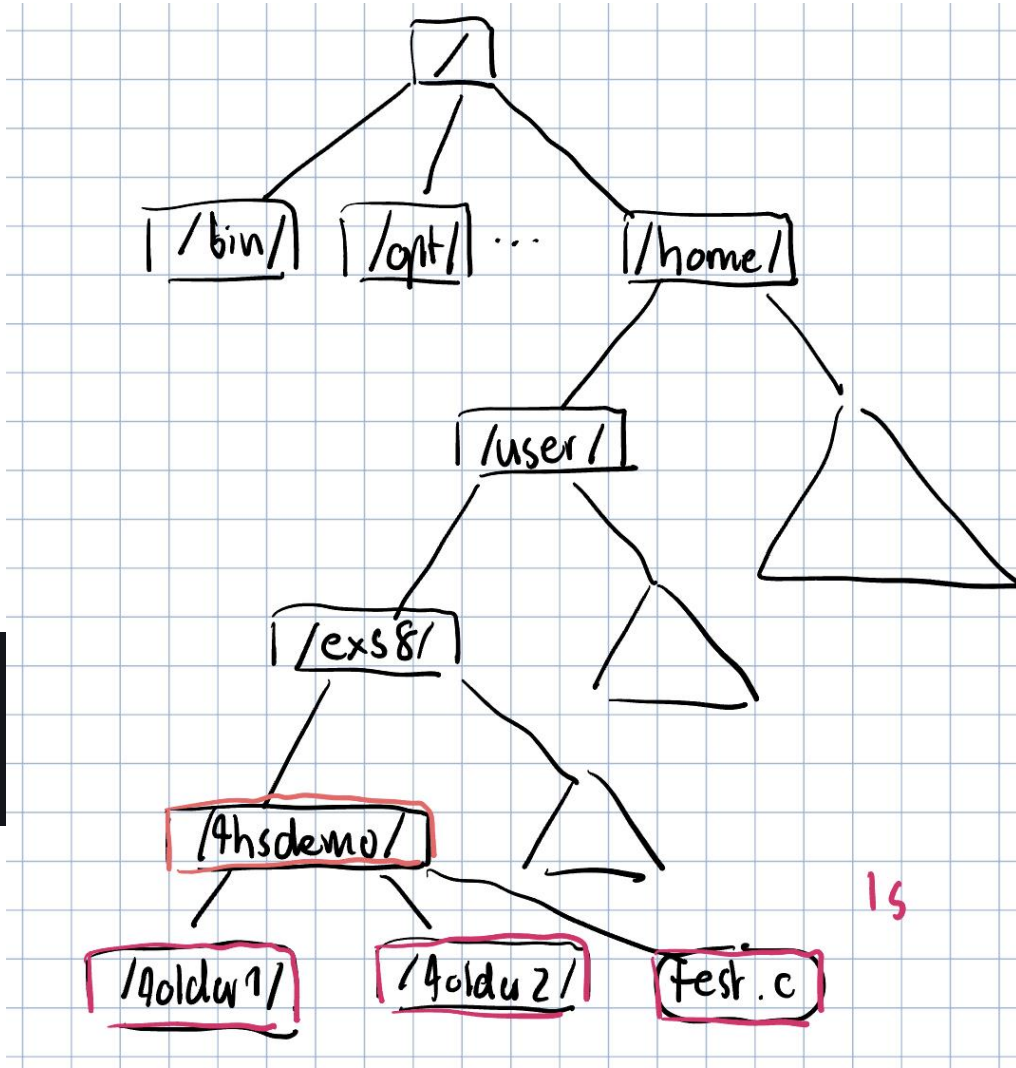
```
user@4865b4f533e3:~/exs8/fhsdemo$ pwd
/home/user/exs8/fhsdemo
user@4865b4f533e3:~/exs8/fhsdemo$ cd /
user@4865b4f533e3:/$ pwd
/
```



UNIX FHS

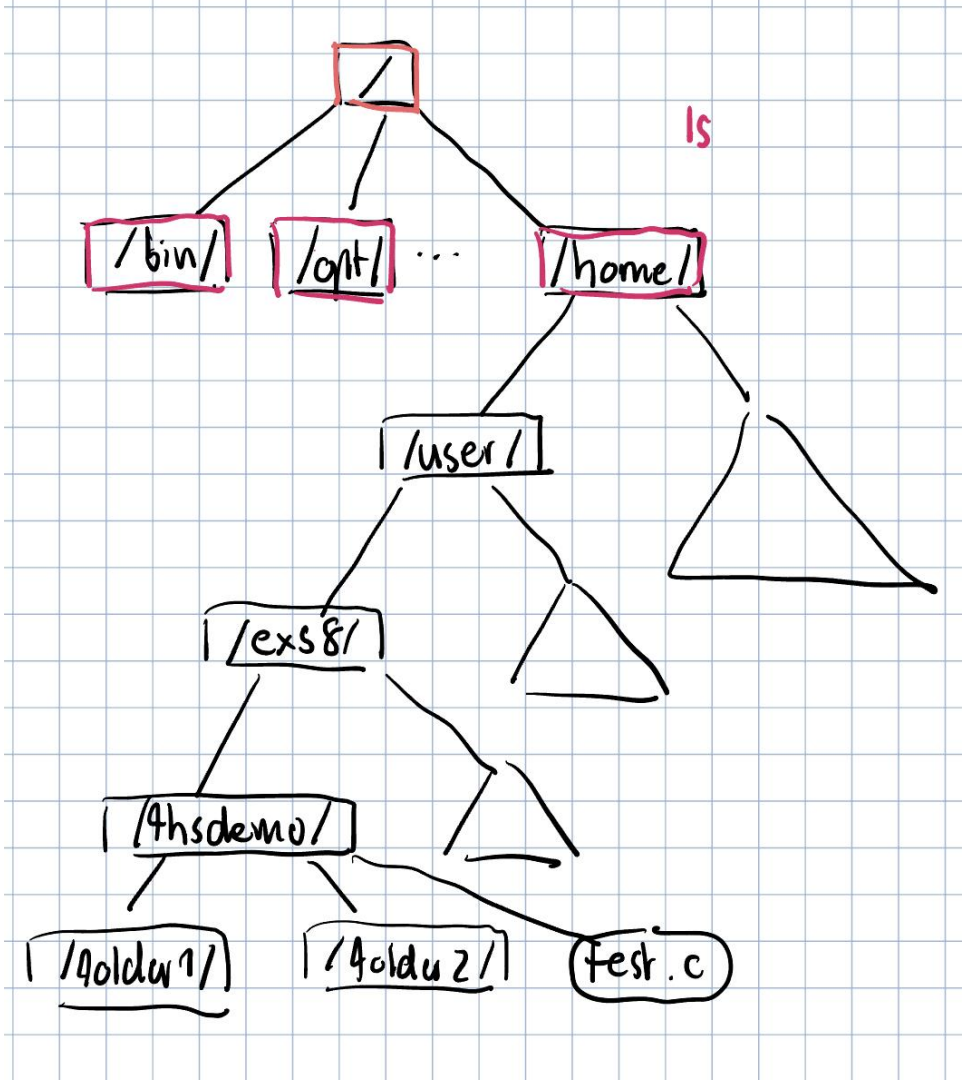
- Check whats inside your current directory (node)?
- “ls” for list

```
user@4865b4f533e3:~/exs8/fhsdemo$ pwd
/home/user/exs8/fhsdemo
user@4865b4f533e3:~/exs8/fhsdemo$ ls
folder1  folder2  test.c
```

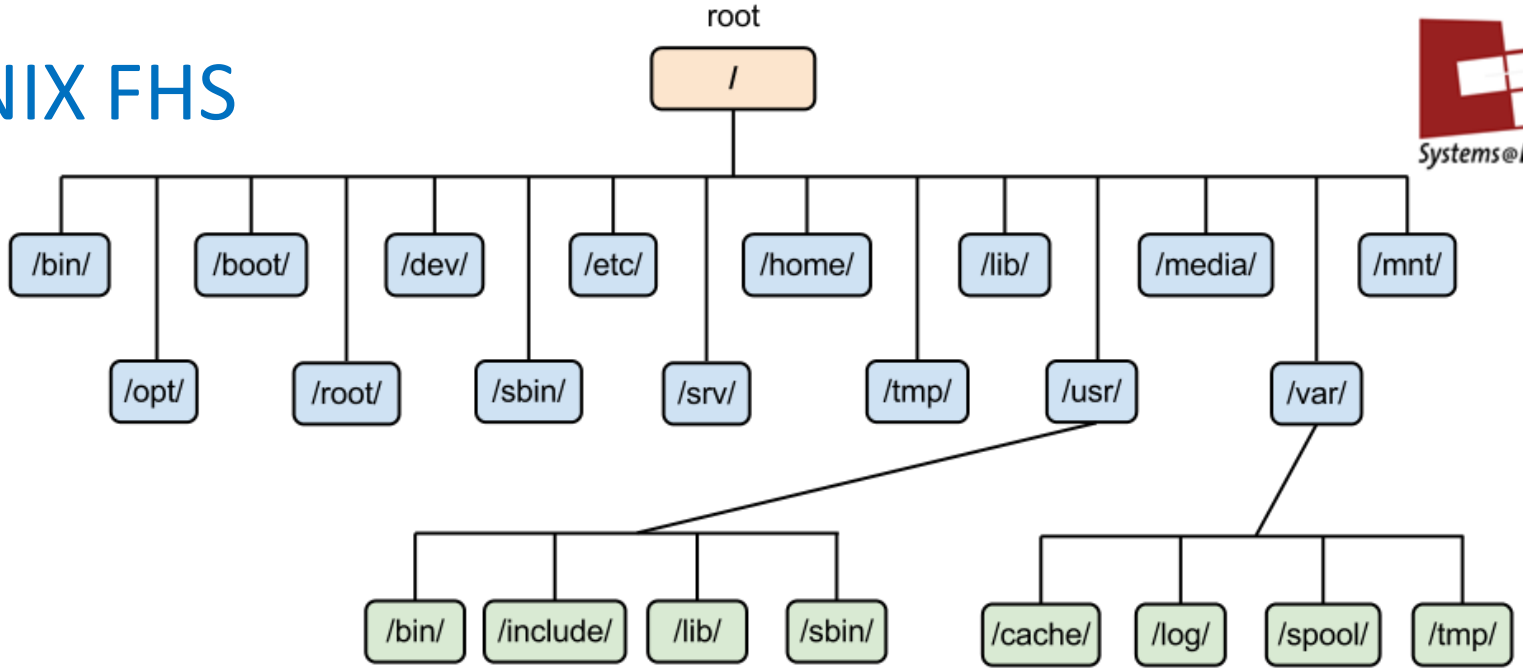


UNIX FHS

- Lets check whats inside “/”?



UNIX FHS



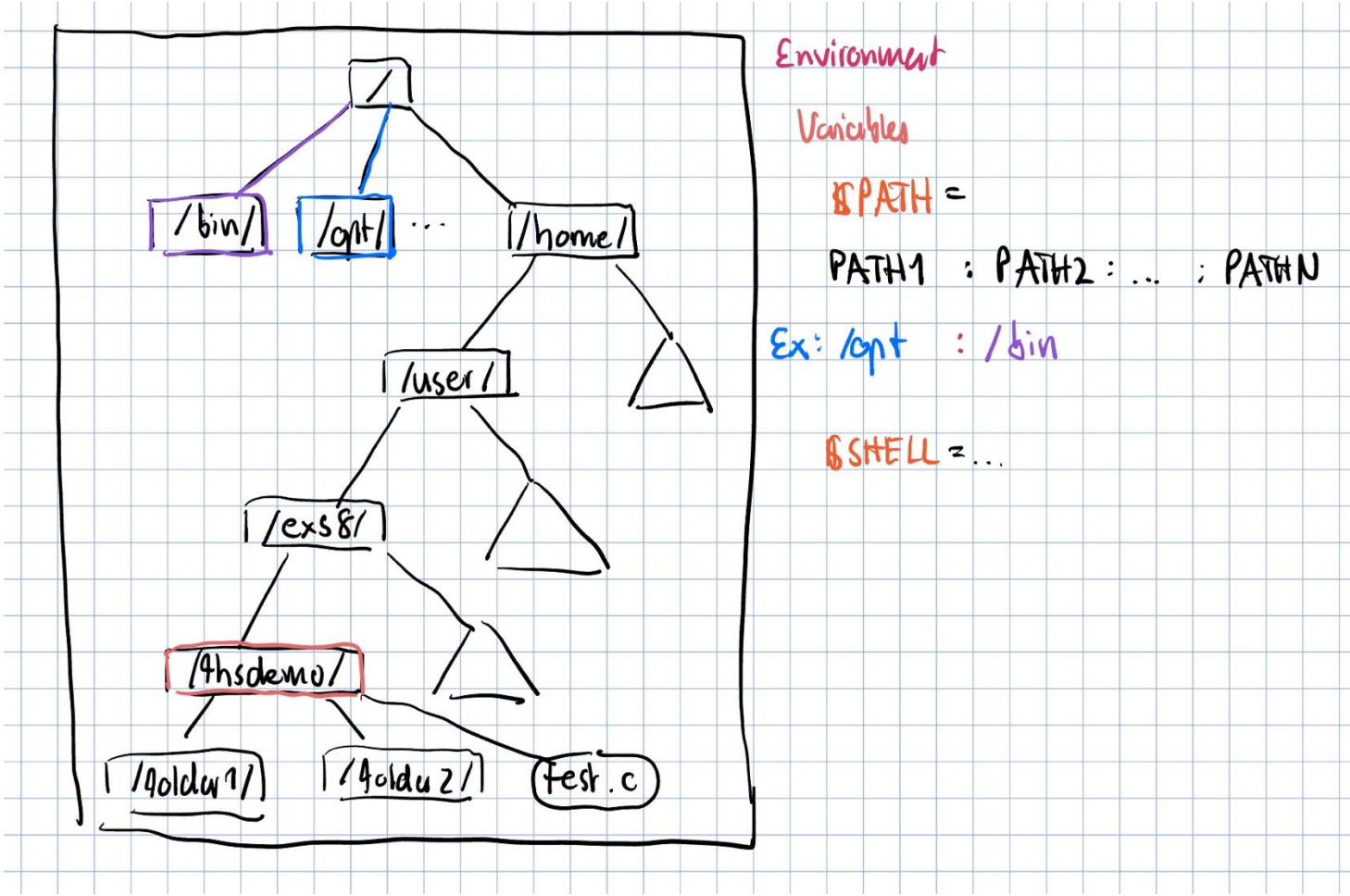

```
user@4865b4f533e3:/$ pwd
/
user@4865b4f533e3:/$ ls
bin   dev  home  lib32  libx32  mnt  proc  run   srv  tmp  var
boot  etc  lib   lib64  media   opt  root  sbin  sys  usr
```

UNIX FHS

Binaries in the FHS

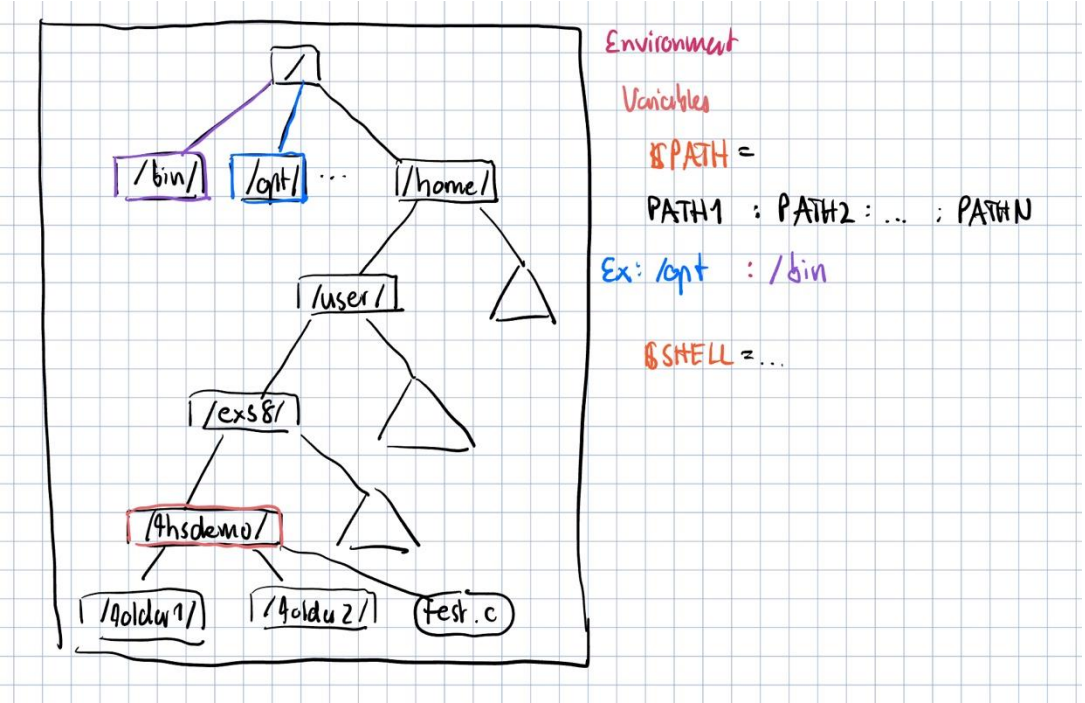
UNIX FHS

- We have an **environment** : think of having variables next to the graph
- Can store **paths** for instance



UNIX FHS

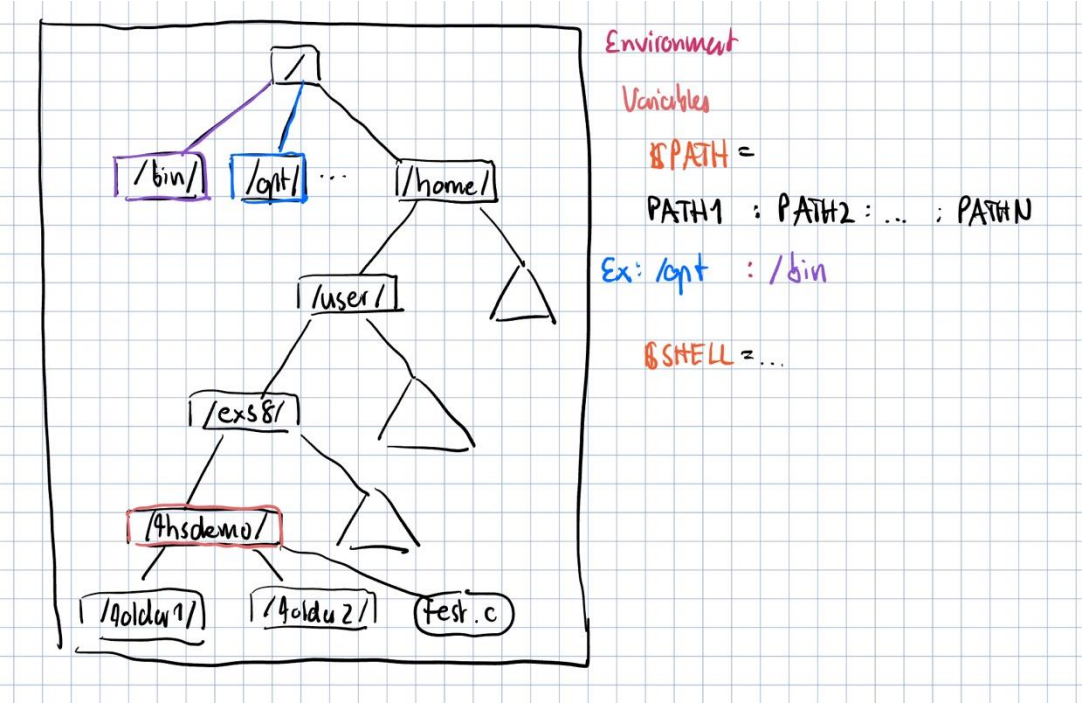
- **$PATH**: Stores paths PATH1 : PATH2 : … : PATHN
- Each Path separated by “:” in between
- Path1= /opt
- Path2= /bin
- $PATH=/opt:/bin




```
user@4865b4f533e3:/$ echo $PATH
/usr/local/sbin:/usr/local/bin:/usr/sbin:/usr/bin:/sbin:/bin
```

UNIX FHS

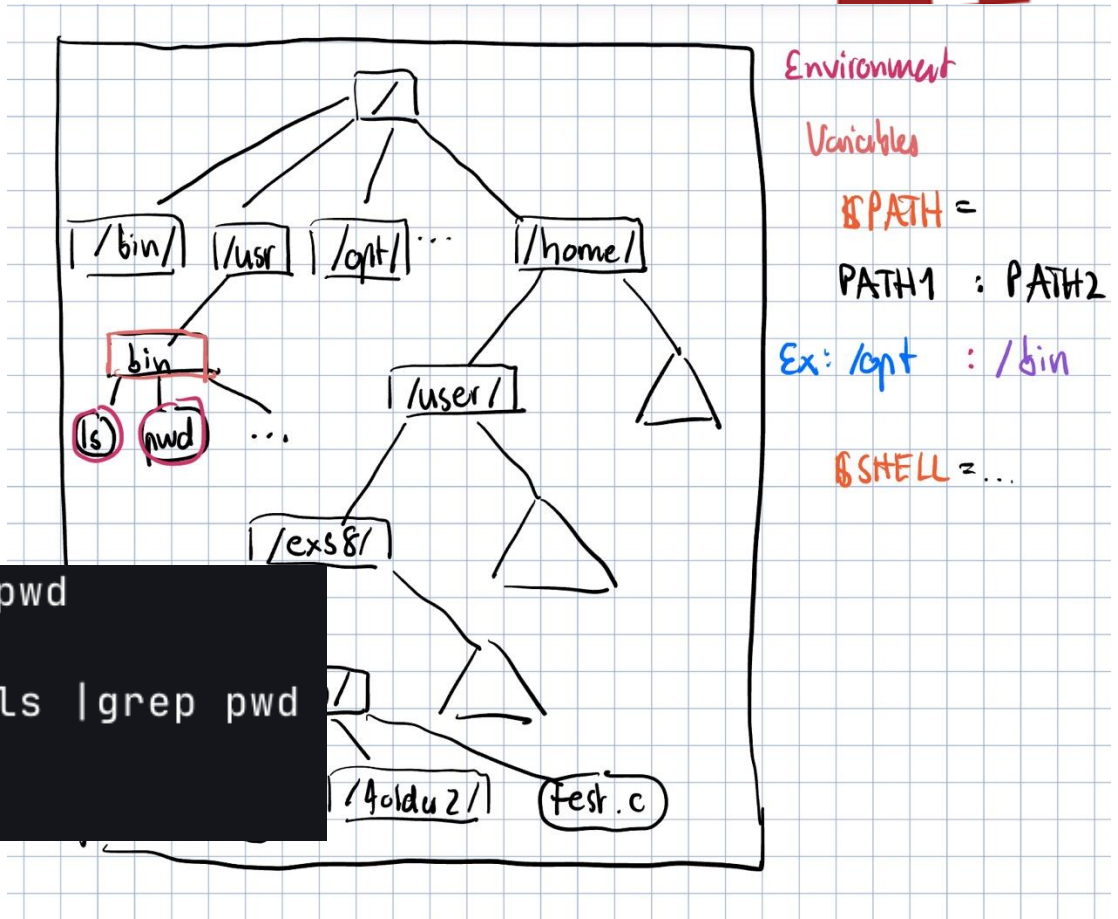
- Every “ls” or ”pwd” etc. executes a **binary** (program)
- It finds it by looking in **all directories** specified by **path variable $PATH**
- “which <binary>” tells us where it is in the graph




```
user@4865b4f533e3:/$ which ls
/usr/bin/ls
user@4865b4f533e3:/$ which pwd
/usr/bin/pwd
```

UNIX FHS

- Check out the path, and check if its actually here? **IT IS**




```
user@4865b4f533e3:/usr/bin$ pwd
/usr/bin
user@4865b4f533e3:/usr/bin$ ls |grep pwd
pwd
pwdx
```

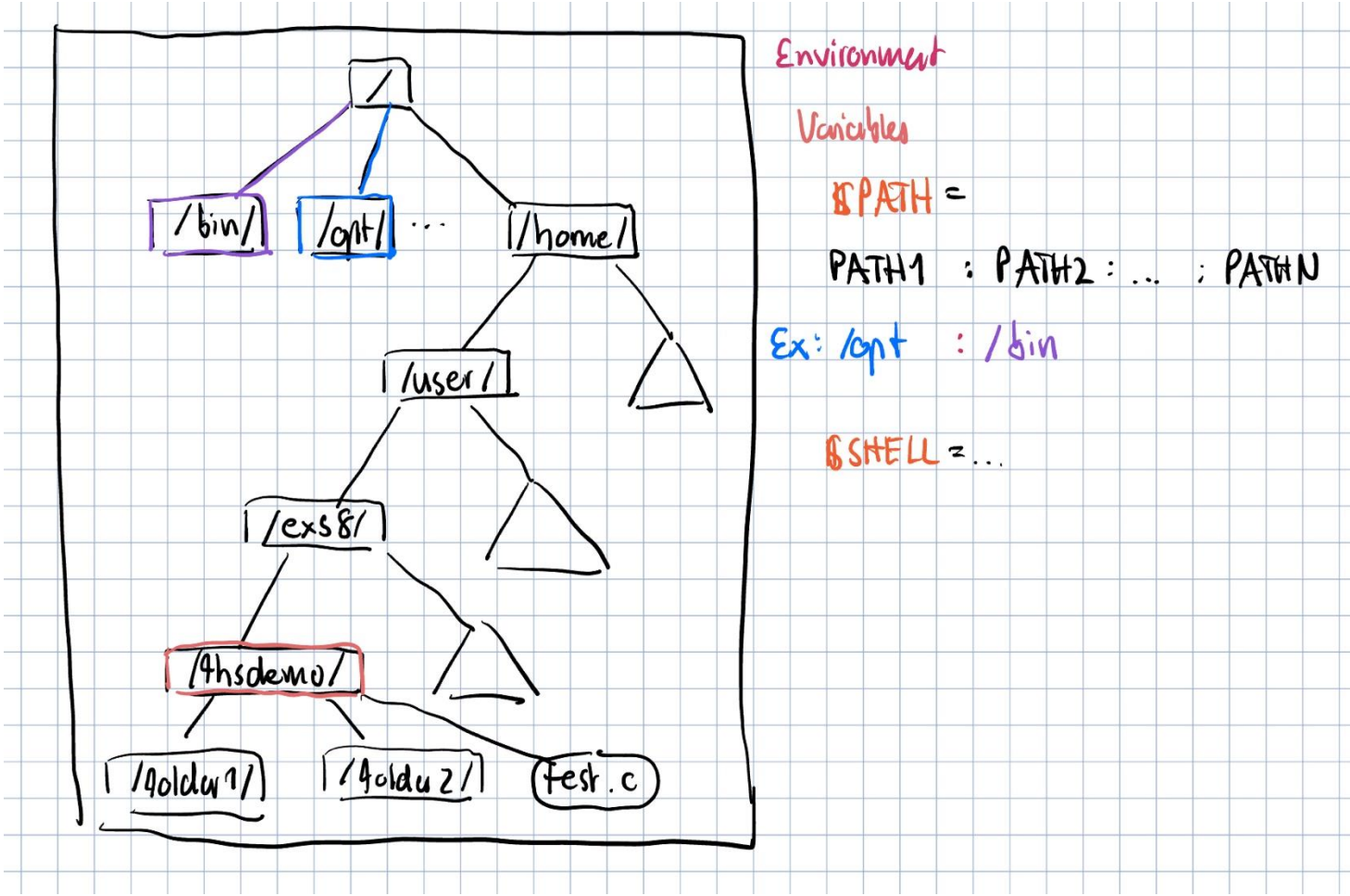
UNIX FHS

- We can do this for **any binary** specified in the **$PATH**
- Lets check out **gcc**

```
user@4865b4f533e3:/usr/bin$ which gcc
/usr/bin/gcc
user@4865b4f533e3:/usr/bin$ cd /usr/bin/
user@4865b4f533e3:/usr/bin$ pwd
/usr/bin
user@4865b4f533e3:/usr/bin$ ls |grep gcc
c89-gcc
c99-gcc
gcc
gcc-11
gcc-ar
gcc-ar-11
gcc-nm
gcc-nm-11
gcc-ranlib
gcc-ranlib-11
x86_64-linux-gnu-gcc
x86_64-linux-gnu-gcc-11
x86_64-linux-gnu-gcc-ar
x86_64-linux-gnu-gcc-ar-11
x86_64-linux-gnu-gcc-nm
x86_64-linux-gnu-gcc-nm-11
x86_64-linux-gnu-gcc-ranlib
x86_64-linux-gnu-gcc-ranlib-11
user@4865b4f533e3:/usr/bin$
```

UNIX FHS

- **Note**: we can be **anywhere** in the FHS, and execute those commands as **$PATH** specifies where to look for it
- **Do not have to be in the directory /bin to execute it**



UNIX FHS

- Write own C program printing “hello, world”
- **Compile it**

```
[bfalk@piora1 ~]$ pwd
/home/bfalk
[bfalk@piora1 ~]$ cat hello.c
#include <stdio.h>

int main(int argc, char** argv){
        printf("hello, world\n");
        return 0;
}
[bfalk@piora1 ~]$ gcc -o hello hello.c
[bfalk@piora1 ~]$ ls
hello  hello.c  simpleadd  simpleadd.c
[bfalk@piora1 ~]$
```

UNIX FHS

- Write own C program (you know how to!)
- **Compile it**
- Execute via “./hello”

- Difference between “./hello” and “/home/bfalk/hello”?

```
[bfalk@piora1 ~]$ ./hello
hello, world
[bfalk@piora1 ~]$ pwd
/home/bfalk
[bfalk@piora1 ~]$ /home/bfalk/hello
hello, world
[bfalk@piora1 ~]$
```

UNIX FHS

- Write own C program (you know how to!)
- **Compile it**
- Execute via “./hello”

- Difference between “./hello” and “/home/bfalk/hello”?

```
[bfalk@piora1 ~]$ ./hello
hello, world
[bfalk@piora1 ~]$ pwd
/home/bfalk
[bfalk@piora1 ~]$ /home/bfalk/hello
hello, world
[bfalk@piora1 ~]$
```

UNIX FHS

- Why doesn’t “hello” work? “pwd” “ls” etc. worked without the “./”?

```
[bfalk@piora1 ~]$ hello
-bash: hello: command not found
[bfalk@piora1 ~]$
```

UNIX FHS

- Why doesn’t “hello” work? “pwd” “ls” etc. worked without the “./”?

- **Issue**: for non absolute path it checks $PATH variable, there is no “hello” executable anywhere

```
[bfalk@piora1 ~]$ hello
-bash: hello: command not found
[bfalk@piora1 ~]$ which hello
/usr/bin/which: no hello in (/cm/local/apps/gcc/13.1.0/bin:/home/bfalk/.local/bin:/home/bfalk/bin:/
cm/local/apps/environment-modules/4.5.3//bin:/usr/local/bin:/usr/bin:/usr/local/sbin:/usr/sbin:/sbi
n:/usr/sbin:/cm/local/apps/environment-modules/4.5.3/bin)
[bfalk@piora1 ~]$
```

UNIX FHS

- Why doesn't "hello" work? "pwd" "ls" etc. worked without the "./"?
- **Issue**: for non absolute path it checks $PATH variable, there is no "hello" executable anywhere
- **Solution**: add directory which has "hello" executable to path

```
[bfalk@piora1 ~]$ echo $PATH
/cm/local/apps/gcc/13.1.0/bin:/home/bfalk/.local/bin:/home/bfalk/bin:/cm/local/apps/environment-mod
ules/4.5.3//bin:/usr/local/bin:/usr/bin:/usr/local/sbin:/usr/sbin:/sbin:/usr/sbin:/cm/local/apps/en
vironment-modules/4.5.3/bin
[bfalk@piora1 ~]$ pwd
/home/bfalk
[bfalk@piora1 ~]$ export PATH=/home/bfalk:$PATH
[bfalk@piora1 ~]$ echo $PATH
/home/bfalk:/cm/local/apps/gcc/13.1.0/bin:/home/bfalk/.local/bin:/home/bfalk/bin:/cm/local/apps/env
ironment-modules/4.5.3//bin:/usr/local/bin:/usr/bin:/usr/local/sbin:/usr/sbin:/sbin:/usr/sbin:/cm/l
ocal/apps/environment-modules/4.5.3/bin
[bfalk@piora1 ~]$ hello
hello, world
[bfalk@piora1 ~]$
```

UNIX FHS

- Now we can execute hello from **anywhere** just as “pwd” and “ls” etc.
- It appears in **which**, so we know its accessible form **everywhere**
- **“~”** is short for my home, i.e. /home/bfalk

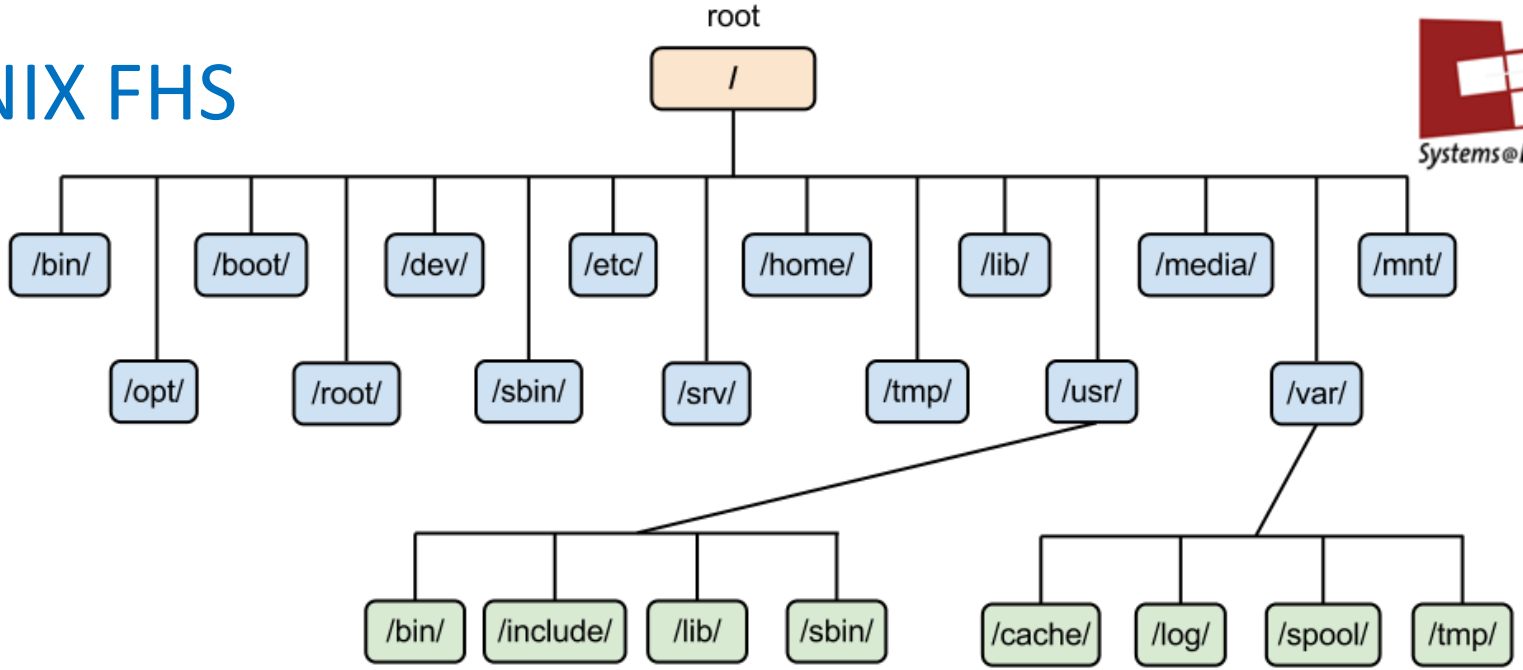
```
[bfalk@piora1 ~]$ cd /
[bfalk@piora1 /]$ pwd
/
[bfalk@piora1 /]$ hello
hello, world
[bfalk@piora1 /]$
```

```
[bfalk@piora1 /]$ which hello
~/hello
[bfalk@piora1 /]$
```

UNIX FHS

Libraries in the FHS

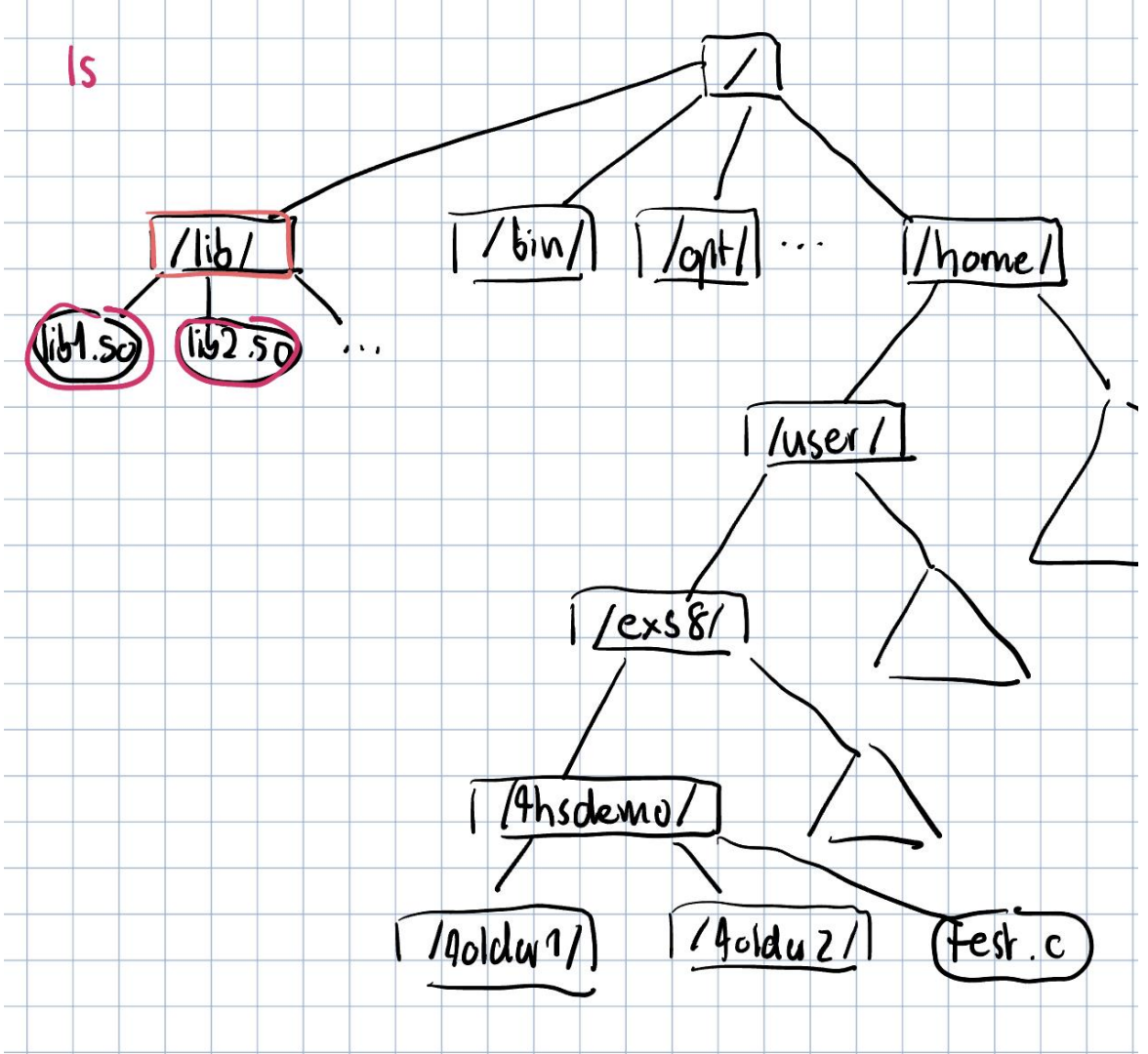
UNIX FHS



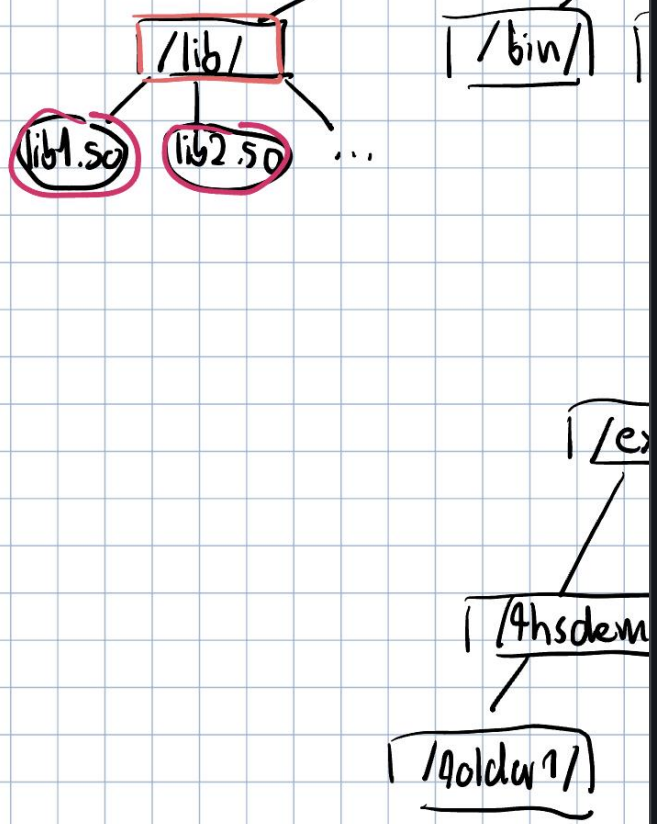

```
user@4865b4f533e3:/$ pwd
/
user@4865b4f533e3:/$ ls
bin   dev  home  lib32  libx32  mnt  proc  run   srv  tmp  var
boot  etc  lib   lib64  media   opt  root  sbin  sys  usr
```

UNIX FHS

- **Just** as binaries have their own locations in FHS (mostly /bin), **libraries** are also somewhere in **/lib** or **/usr/lib**



ls




```
[bfalk@piora1 lib]$ pwd
/lib
[bfalk@piora1 lib]$ ls
audit                                libnl-3.so.200
bfd-plugins                          libnl-3.so.200.26.0
binfmt.d                             libnl-genl-3.so.200
cmake                                libnl-genl-3.so.200.26.0
cpp                                  libnl-idiag-3.so.200
crt1.o                               libnl-idiag-3.so.200.26.0
crti.o                               libnl-nf-3.so.200
crtn.o                               libnl-nf-3.so.200.26.0
cups                                 libnl-route-3.so.200
debug                                libnl-route-3.so.200.26.0
dkms                                 libnl-xfrm-3.so.200
dracut                               libnl-xfrm-3.so.200.26.0
eclipse                              libnss_compat.so.2
engines-3                            libnss_dns.so.2
enroot                               libnss_files.so.2
environment.d                        libnuma.so.1
fipscheck                            libnuma.so.1.0.0
firewalld                            libnvcuvid.so
firmware                             libnvcuvid.so.1
fontconfig                           libnvcuvid.so.550.90.07
games                                libnvidia-allocator.so
gcc                                  libnvidia-allocator.so.1
gconv                                libnvidia-allocator.so.550.90.07
gcrt1.o                              libnvidia-eglcore.so.550.90.07
gems                                 libnvidia-encode.so
gio                                  libnvidia-encode.so.1
girepository-1.0                     libnvidia-encode.so.550.90.07
```

Assignment 5 (Bomb Lab)

Questions?

Linking – Theory Recap

- Compilation Pipeline
- ELF Object File
- Symbols
- Relocation
- Linking Libraries

UNIX FHS

Linking Theory: Static linking

Linking and Loading

- Whats the “issue” with this C program?

Example C program

main.c

```
int buf[2] = {1, 2};

int main()
{
  swap();
  return 0;
}
```

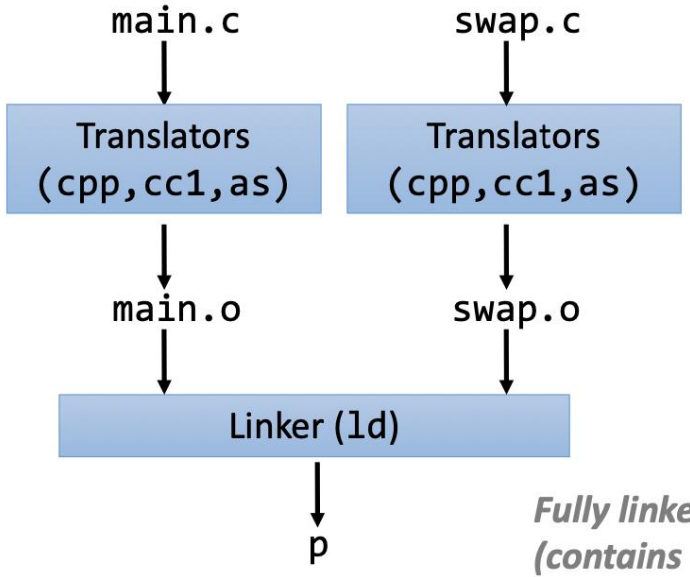
swap.c

```
extern int buf[];

static int *bufp0 = &buf[0];
static int *bufp1;

void swap()
{
  int temp;

  bufp1  = &buf[1];
  temp   = *bufp0;
  *bufp0 = *bufp1;
  *bufp1 = temp;
}
```

Static linking

- Programs are translated and linked using a *compiler driver*:

```
unix> gcc -O2 -g -o p main.c swap.c
unix> ./p
```



Source files

Separately compiled relocatable object files

Fully linked executable object file (contains code and data for all functions defined in main.c and swap.c

Static Linking in 2 Steps

Systems@ETH Zürich

- Step 1: Symbol resolution
  - Programs define and reference *symbols* (variables and functions):
    - `void swap() {…}  /* define symbol swap */`
    - `swap();  /* reference symbol swap */`
    - `int *xp = &x;  /* define xp, reference x */`
  - Symbol definitions are stored (by compiler) in *symbol table*.
    - Symbol table is an array of structs
    - Each entry includes name, type, size, and location of symbol.
  - Linker associates each symbol reference with exactly one symbol definition.

Static Linking in 2 Steps

Systems@ETH Zürich

What do linkers do?

- Step 2: Relocation
  - Merges separate code and data sections into single sections
  - Relocates symbols from their relative locations in the `.o` files to their final absolute memory locations in the executable.
  - Updates all references to these symbols to reflect their new positions.

Object files

3 kinds of object files (modules)

- Relocatable object file (.o file)
  - Contains code and data in a form that can be combined with other relocatable object files to form executable object file.
  - Each .o file is produced from exactly one source (.c) file
- Executable object file
  - Contains code and data in a form that can be copied directly into memory and then executed.
- Shared object file (.so file)
  - Special type of relocatable object file that can be loaded into memory and linked dynamically, at either load time or run-time.
  - Called *Dynamic Link Libraries* (DLLs) by Windows

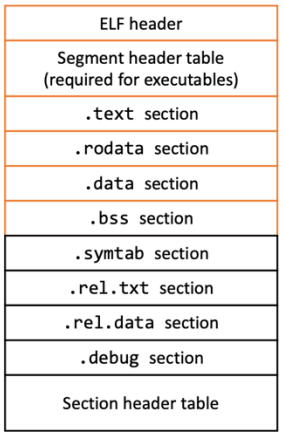
Object files: when a .c -> .o, how does this single .o get stored? That's a **relocatable object file**

Systems@ETH Zürich

- Relocatable object file (.o file)
  - Contains code and data in a form that can be combined with other relocatable object files to form executable object file.
  - Each .o file is produced from exactly one source (.c) file

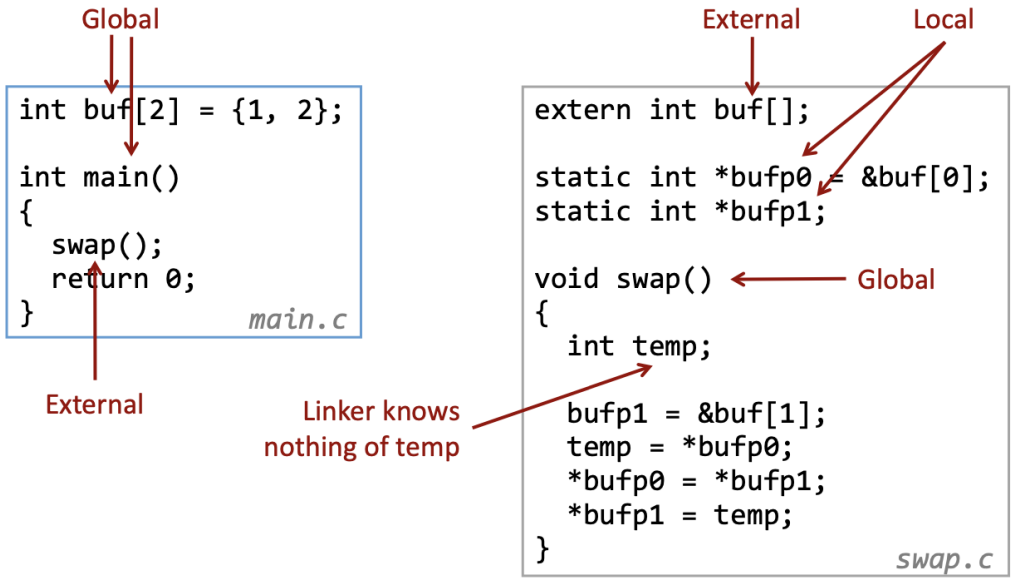
ELF object file format

- Elf header
  - Word size, byte ordering, file type (`.o`, `exec`, `.so`), machine type, etc.
- Segment header table
  - Page size, virtual addresses memory segments (sections), segment sizes.
- `.text` section
  - Code
- `.rodata` section
  - Read only data: jump tables, ...
- `.data` section
  - Initialized global variables
- `.bss` section
  - Uninitialized global variables
  - "Block Started by Symbol"
  - "Better Save Space"
  - Has section header but occupies no space



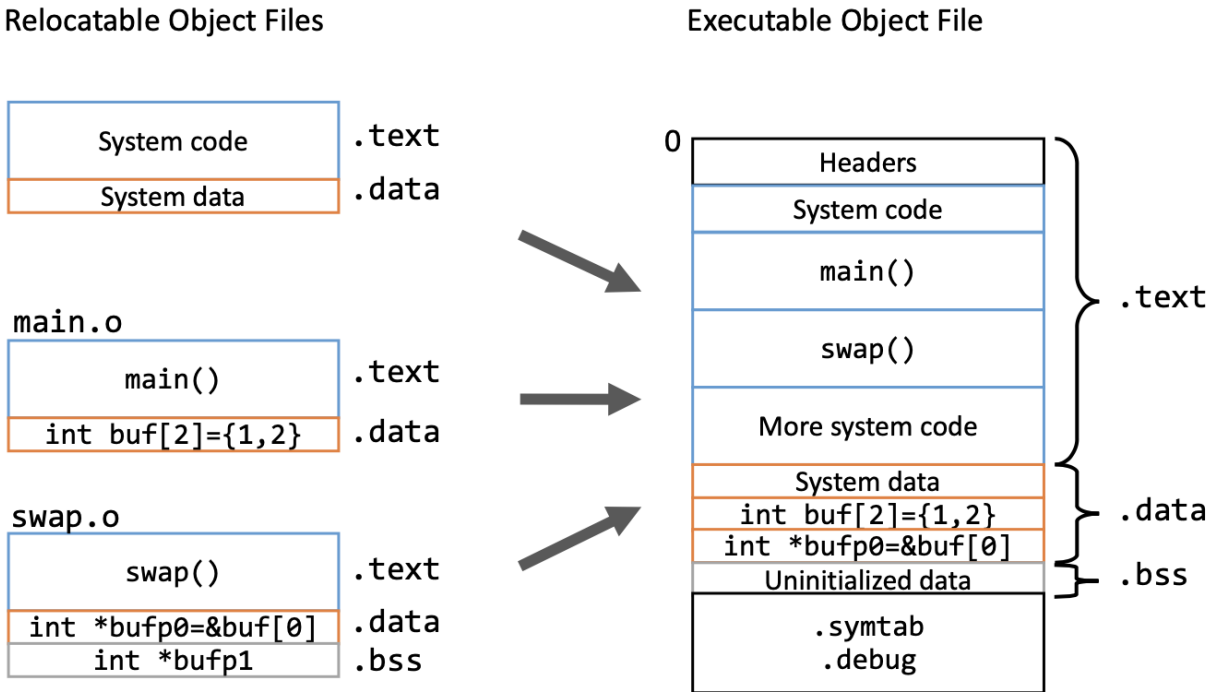
Resolving symbols: Global, External and Local

Resolving symbols



Put all the .o files (which are in ELF format) inside **ONE** big executable

Relocating code and data



Inside each .o file we are **still missing the references**: currently 0 as placeholder

Systems@ETH Zürich

Relocation info (`main`)

main.c

```
int buf[2] = {1,2};

int main()
{
  swap();
  return 0;
}
```

main.o

```
Disassembly of section .text:

0000000000000000 <main>:
   0:   48 83 ec 08      sub    $0x8,%rsp
   4:   b8 00 00 00 00   mov    $0x0,%eax
   9:   e8 00 00 00 00   callq  e <main+0xe>
                a: R_X86_64_PC32 swap-0x4
   e:   b8 00 00 00 00   mov    $0x0,%eax
  13:   48 83 c4 08      add    $0x8,%rsp
  17:   c3               retq
```

```
Disassembly of section .data:

0000000000000000 <buf>:
   0:   01 00 00 00 02 00 00 00
```

Source: `objdump -D -r <file>`

After merging them: we can **check** in the symbol table where the function is in the final executable

- Now the linker can actually input the address of <swap> , but only after **everything got merged** since only then the final addresses are clear

Executable after relocation (.text)

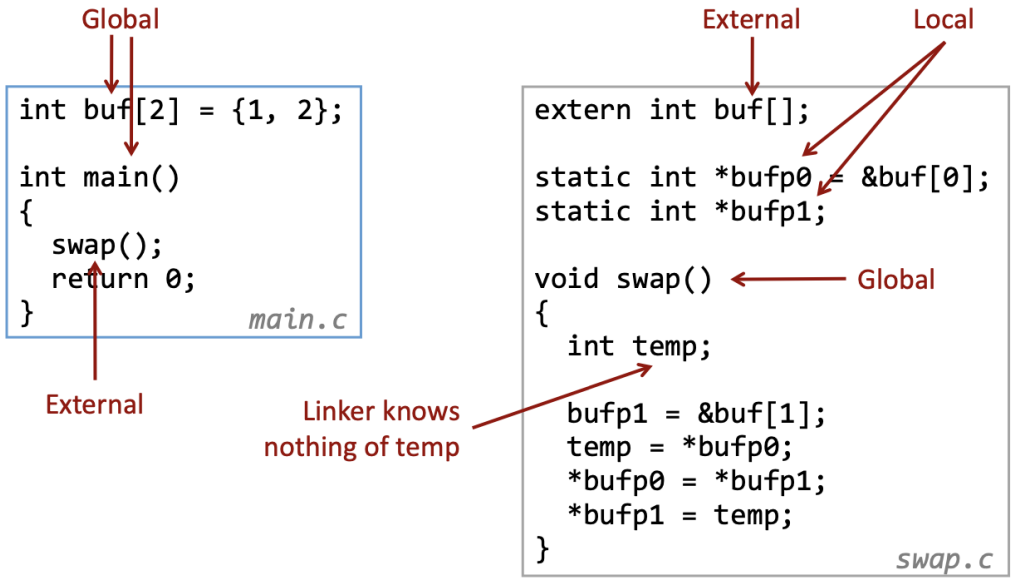
```
00000000004004ed <main>:
  4004ed: 48 83 ec 08              sub    $0x8,%rsp
  4004f1: b8 00 00 00 00           mov    $0x0,%eax
  4004f6: e8 0a 00 00 00           callq  400505 <swap>
  4004fb: b8 00 00 00 00           mov    $0x0,%eax
  400500: 48 83 c4 08              add    $0x8,%rsp
  400504: c3                       retq
```

Lecture Recap

Linking: Issues with duplicate symbol definitions

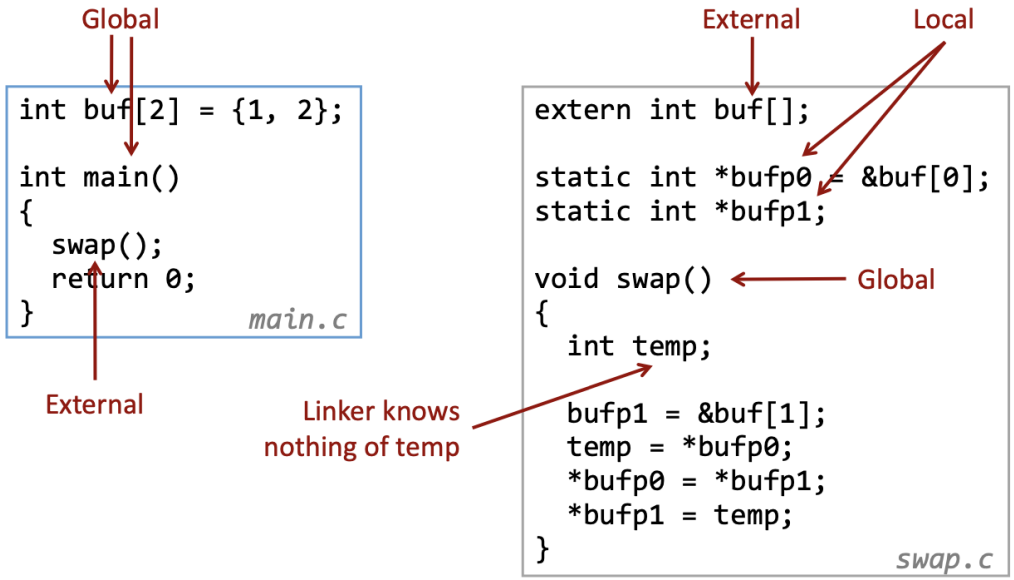
Recall 3 Type of Symbols

Resolving symbols



Recall 3 Type of Symbols

Resolving symbols



Weak and Strong Symbols

Systems@ETH Zürich

- Sometimes, **compiler** doesn't know if **global variable** (uninitialized) should be a **global symbol** or an **external** symbol (if not explicitly defined as external)
- So there is a concept of **strong and weak** symbols, for global variables **only!**

Weak and Strong Symbols

Systems@ETH Zürich

- **Note**: the concept of weak and strong linker symbols are related **exclusively** to **uninitialized global variables -f-common (old behaviour)**: puts uninitialised globals into a common block and they are weak symbols, allowing multiple uninitialized declerations across different .c files
- **-f-nocommon (new default)**: this is not the case anymore, uninitialized globals are **also classified as strong symbols**
- That means: the only weak to get weak symbols in C is to either compile with -f-common compiler flag or with #pragma weak

3 Linker Rules

Systems@ETH Zürich

The linker's symbol rules

1. Multiple strong symbols are not allowed
   - Each item can be defined only once
   - Otherwise: Linker error

2. Given a strong symbol and multiple weak symbol, choose the strong symbol
   - References to the weak symbol resolve to the strong symbol

3. If there are multiple weak symbols, pick an arbitrary one
   - Can override this with `gcc -fno-common`

Will this yield an error?

Systems@ETH Zürich

What about this?

main.c:

```
int count;
int main(int argc, char *argv[])
{
    count = 42;
    print_count();
    return 0;
}
```

other.c:

```
#include <stdio.h>
int count = 1;
void print_count()
{
    printf("Count is %d\n", count);
}
```

Will this yield an error?

- **With –fcommon** the unitialised global is a weak symbol, as the other.c has definition for int count, the linker will turn “int count” i.e. the weak symbol into an external symbol and it links fine.
- **With –f-nocommon**: all global vars are strong, so we have 2 strong symbols which yields an error

With `-fno-common`
(default on very new compilers)

main.c:

```
int count;
int main(int argc, char *argv[])
{
    count = 42;
    print_count();
    ret
}
```

other.c:

```
#include <stdio.h>
int count = 1;
void print_count()
{
    printf("Count is %d\n", count);
```

Link-time error: symbol multiply defined.

main.o:

```
0000000000000000 B count
0000000000000000 T main
                 U print_count
```

other.o:

```
0000000000000000 D count
0000000000000000 T print_count
                 U printf
```

With `-fcommon`
(default pre-COVID)

main.c:

```
int count;
int main(int argc, char *argv[])
{
    count = 42;
    print_count();
    ret
}
```

other.c:

```
#include <stdio.h>
int count = 1;
void print_count()
{
    printf("Count is %d\n", count);
```

No errors: program links and runs

main.o:

```
0000000000000000 C count
0000000000000000 T main
                 U print_count
```

other.o:

```
0000000000000000 D count
0000000000000000 T print_count
                 U printf
```

“Common Block”: *weak symbol*

That's why!

- **Static**: means it's a local variable now, linker doesn't care anymore
- **Initialise**: no ambiguity (both in –fcommon and –fno-common counts as d**definition**)
- **Extern**: make it explicitly to an external variable

Global variables

- Avoid if you can!
- Otherwise
  - Use `static` if you can
  - Initialize if you define a global variable
  - Use `extern` if you use external global variable

Lecture Recap

Static Libraries

Why do we need libraries?

Packaging commonly-used functions

- How to package functions commonly used by programmers?
  - Math, I/O, memory management, string manipulation, etc.

- Awkward, given the linker framework so far:
  - **Option 1:** Put all functions into a single source file
    - Programmers link big object file into their programs
    - Space and time inefficient
  - **Option 2:** Put each function in a separate source file
    - Programmers explicitly link appropriate binaries into their programs
    - More efficient, but burdensome on the programmer

What are (static) libraries

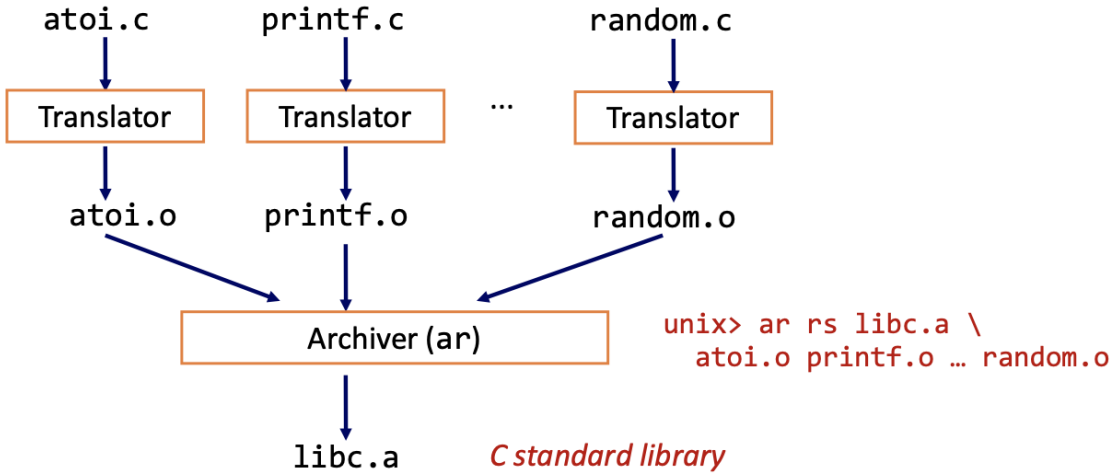
Systems@ETH Zürich

Solution: static libraries

- Static libraries (`.a` archive files)
  - Concatenate related relocatable object files into a single file with an index (called an *archive*).
  - Enhance linker so that it tries to resolve unresolved external references by looking for the symbols in one or more archives.
  - If an archive member file resolves reference, link into executable.

What are (static) libraries

Creating static libraries



Archiver allows incremental updates

Recompile function that changes and replace .o file in archive.

Example: check your linux system

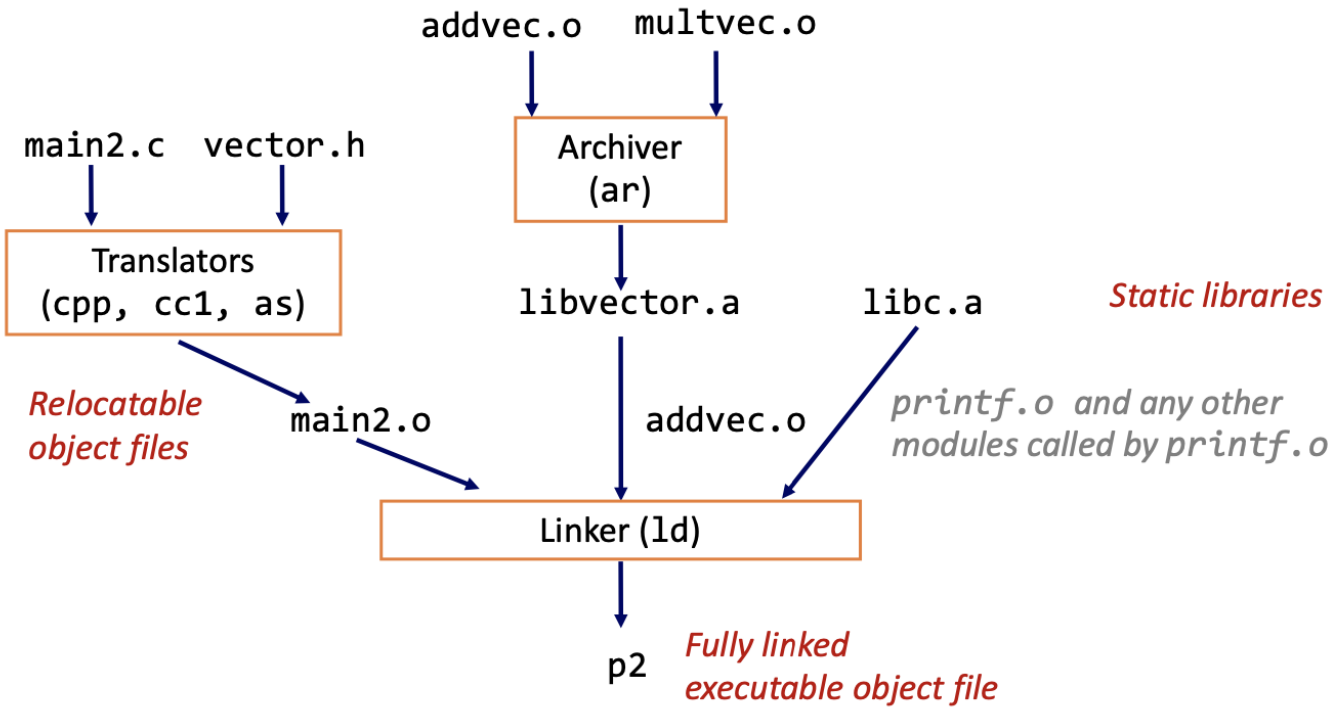
Commonly-used libraries

- `libc.a` (the C standard library)
  - 8 MB archive of 900 object files.
  - I/O, memory allocation, signal handling, string handling, data and time, random numbers, integer math
- `libm.a` (the C math library)
  - 1 MB archive of 226 object files.
  - floating point math (sin, cos, tan, log, exp, sqrt, …)

```
% ar -t /usr/lib/libc.a | sort
…
fork.o
…
fprintf.o
fpu_control.o
fputc.o
freopen.o
fscanf.o
fseek.o
fstab.o
…
```

```
% ar -t /usr/lib/libm.a | sort
…
e_acos.o
e_acosf.o
e_acosh.o
e_acoshf.o
e_acoshl.o
e_acosl.o
e_asin.o
e_asinf.o
e_asinl.o
…
```

Linking with static libraries



Lecture Recap

Shared Libraries (.so files)

Issue with static libraries?

Shared libraries

- Static libraries have the following disadvantages:
  - Duplication in the stored executables
    (every function needs the standard libc)
  - Duplication in the running executables
  - Minor bug fixes of system libraries require each application to explicitly relink

- 100 files which use printf? Then in 100 executables, the “printf” code was pulled out of the library and **inserted** into the executables **(duplication in stored executables)**
- **Change something in a library**: as we pulled the code out and compiled it inside the executable: the old code **stays in the exceutable**

Solution: Shared libraries!

Systems@ETH Zürich

- Solution: shared libraries
  - Object files that contain code and data that are loaded and linked into an application *dynamically,* at either *load-time* or *run-time*
  - Also called: dynamic link libraries, DLLs, .so files

- **Load time**: when program is loaded into memory to be executed
- **Run time**: when program already runs you get the code (as a function pointer)

Solution: Shared libraries!

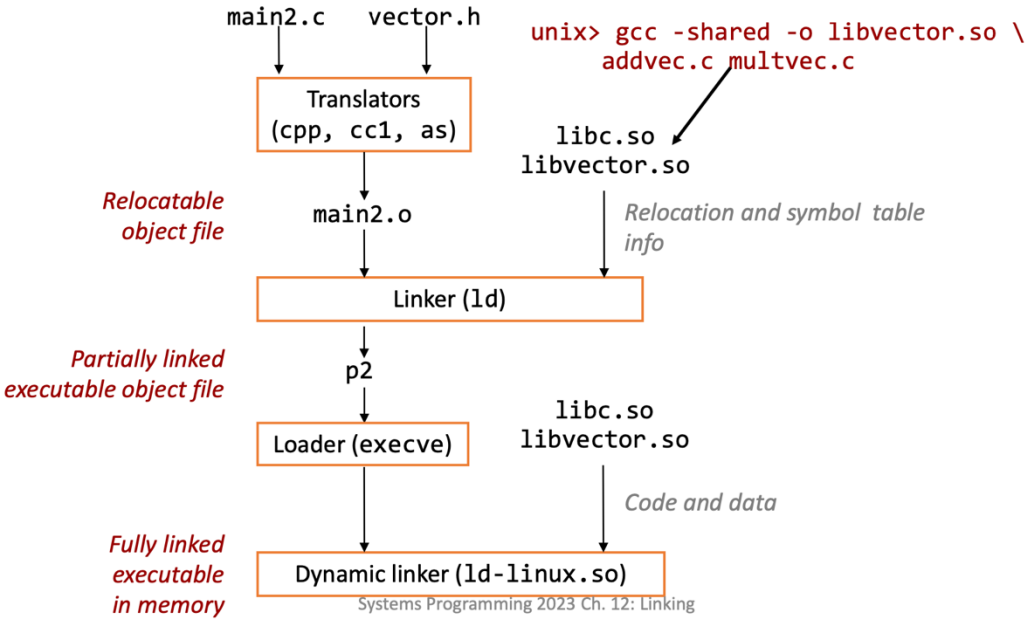
Systems@ETH Zürich

Shared libraries

- Dynamic linking can occur when executable is first loaded and run (load-time linking).
  - Common case for Linux, handled automatically by the dynamic linker (`ld-linux.so`).
  - Standard C library (`libc.so`) usually dynamically linked.
- Dynamic linking can also occur after program has begun (run-time linking).
  - In Unix, this is done by calls to the `dlopen()` interface.
    - High-performance web servers.
    - Runtime library interpositioning
- Shared library routines can be shared by multiple processes.
  - More on this when we learn about virtual memory

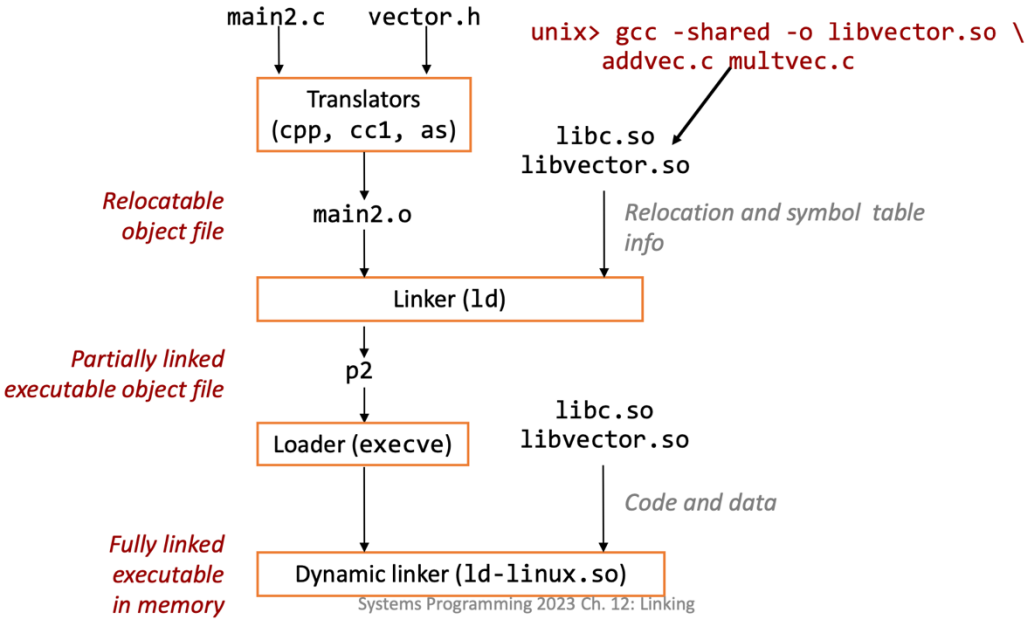
Load time Linking

Dynamic linking at load-time



Load time Linking

Dynamic linking at load-time



Load Time Linking

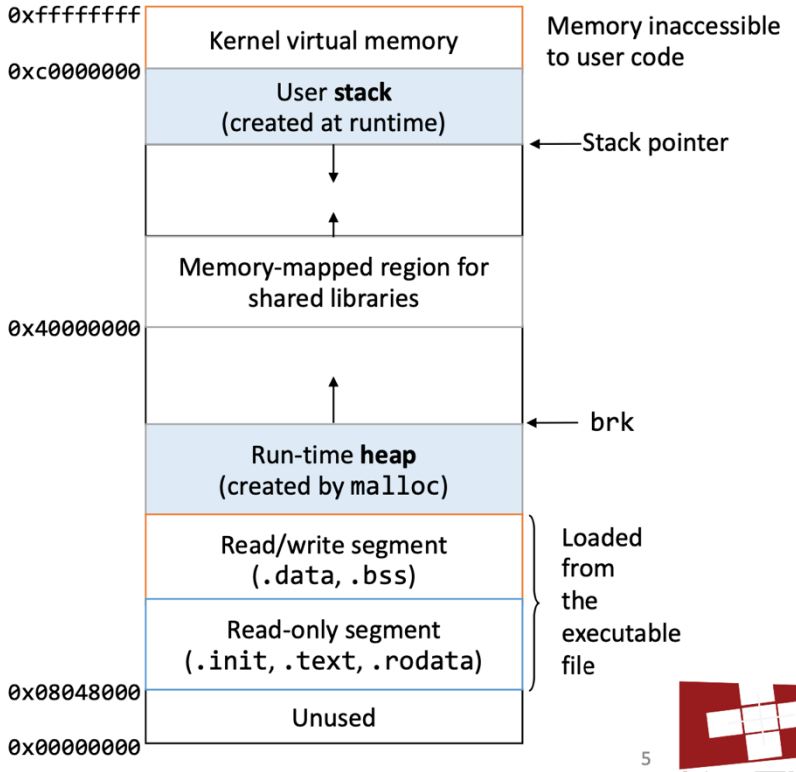
Systems@ETH Zürich

- **Since** the code is not compiled into the executable (as in the static version): the library functions used **cannot be in the .text segment** as the linker didn't have the code
- **Instead**: the dynamic linker simply puts them into "memory mapped regions for shared libraries": between **stack and heap**

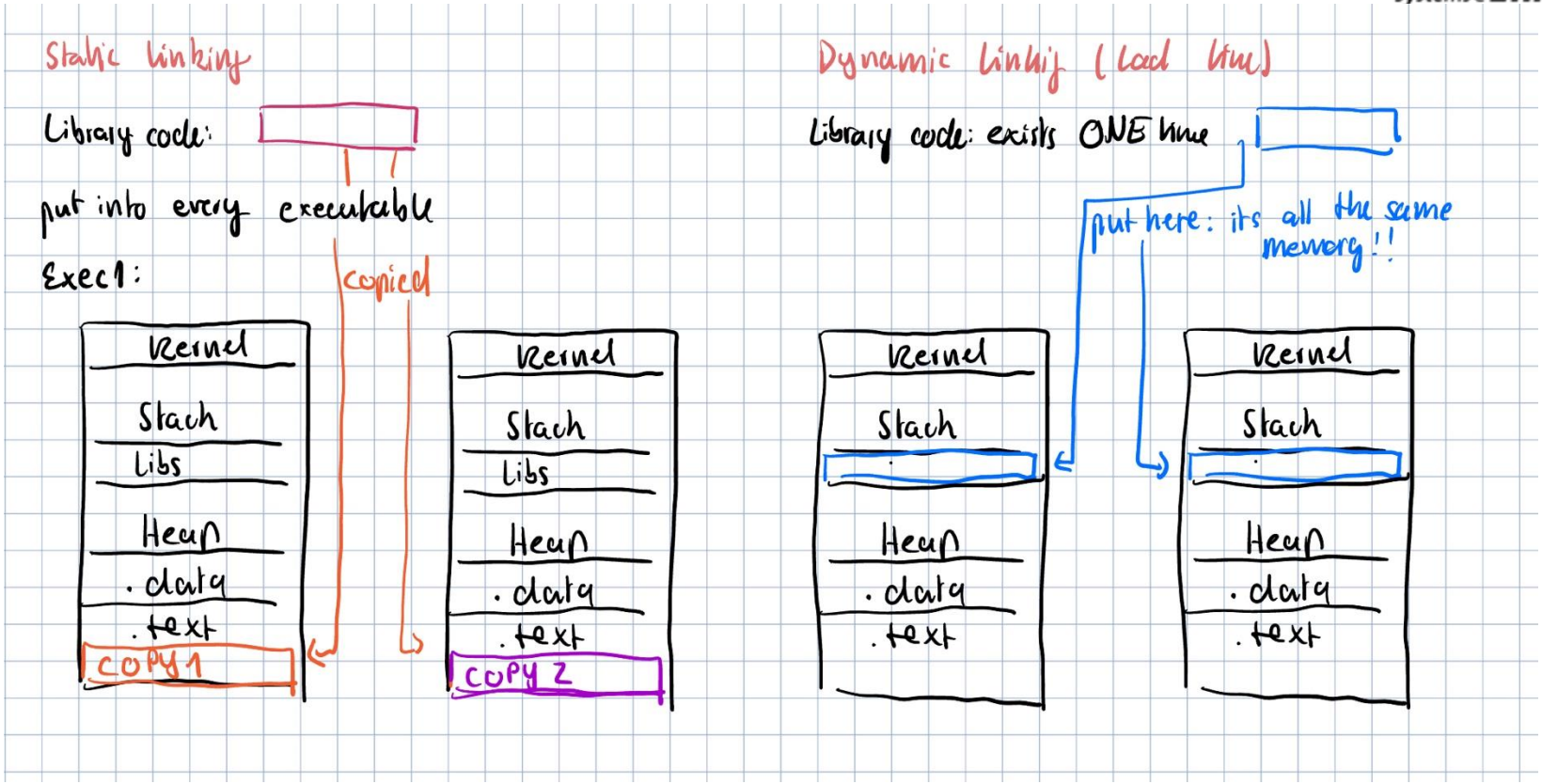
Load time Linking

Loading

- When the OS loads a program, it:
  - creates an address space
  - inspects the executable file to see what's in it
  - (lazily) copies regions of the file into the right place in the address space
  - does any final linking, relocation, or other needed preparation



Load time Linking



Shared Libraries Demo

A few old exam tasks on Linking

HS21 Question 4

Linking Exam Task (a)

Systems@ETH Zürich

a) Which of the following is the correct ordering (left to right) of a file's compilation cycle? Note: a filename without an extension is an executable.

(2 points)

- ☐ `foo.c` → `foo.o` → `foo.s` → `foo`
- ☐ `foo.h` → `foo.o` → `foo.s` → `foo`
- ☐ `foo.c` → `foo.so` → `foo`
- ☐ `foo.c` → `foo.s` → `foo.o` → `foo`
- ☐ `foo.c` → `foo.o` → `foo.so` → `foo`
- ☐ None of the above.

Linking Exam Task (a)

Systems@ETH Zürich

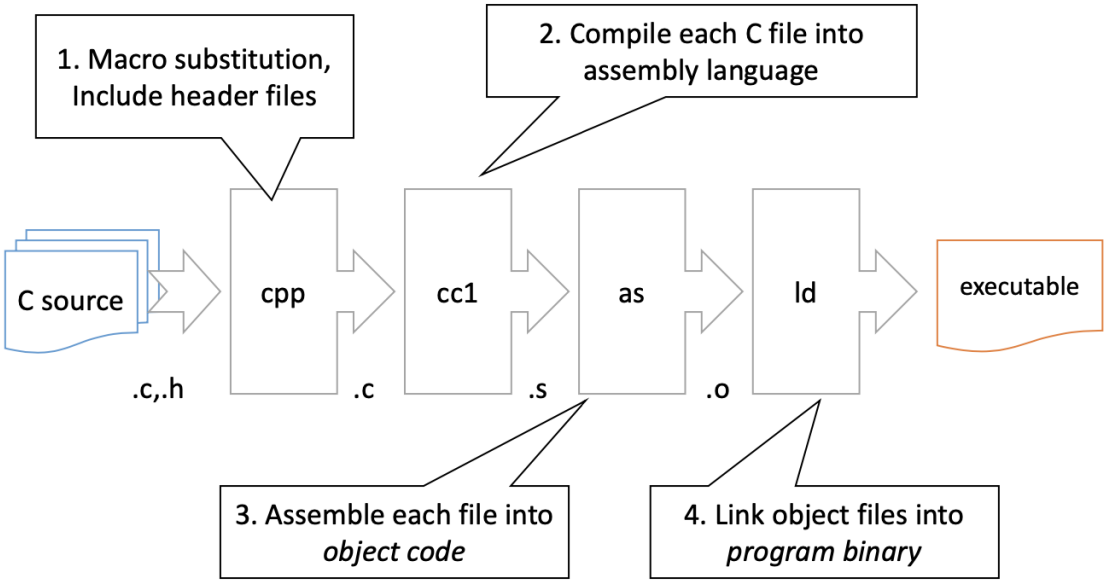
a) Which of the following is the correct ordering (left to right) of a file's compilation cycle? Note: a filename without an extension is an executable.

(2 points)

- [ ] `foo.c` → `foo.o` → `foo.s` → `foo`
- [ ] `foo.h` → `foo.o` → `foo.s` → `foo`
- [ ] `foo.c` → `foo.so` → `foo`
- [x] `foo.c` → `foo.s` → `foo.o` → `foo`
- [ ] `foo.c` → `foo.o` → `foo.so` → `foo`
- [ ] None of the above.

Linking Exam Task (a)

GNU **gcc** Toolchain



Linking Exam Task (b)

Systems@ETH Zürich

b) Which of the following is not a section of an ELF file?

(2 points)

- ☐ `.code`
- ☐ `.data`
- ☐ `.bss`
- ☐ `.text`
- ☐ `.rodata`

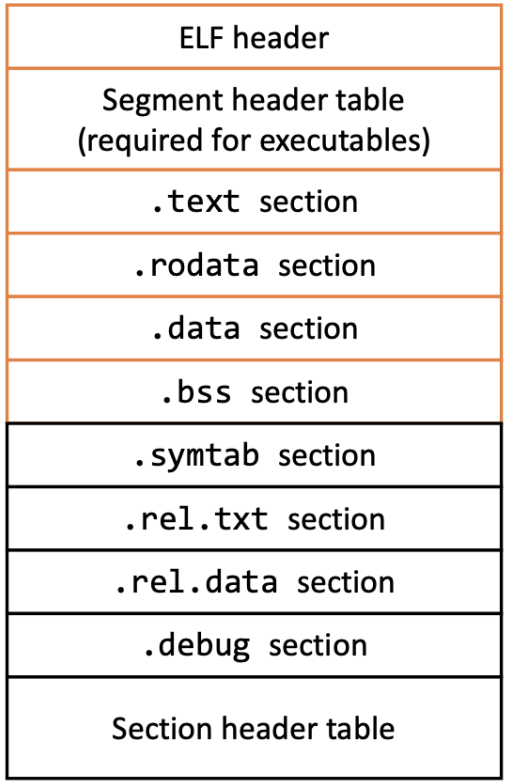
Linking Exam Task (b)

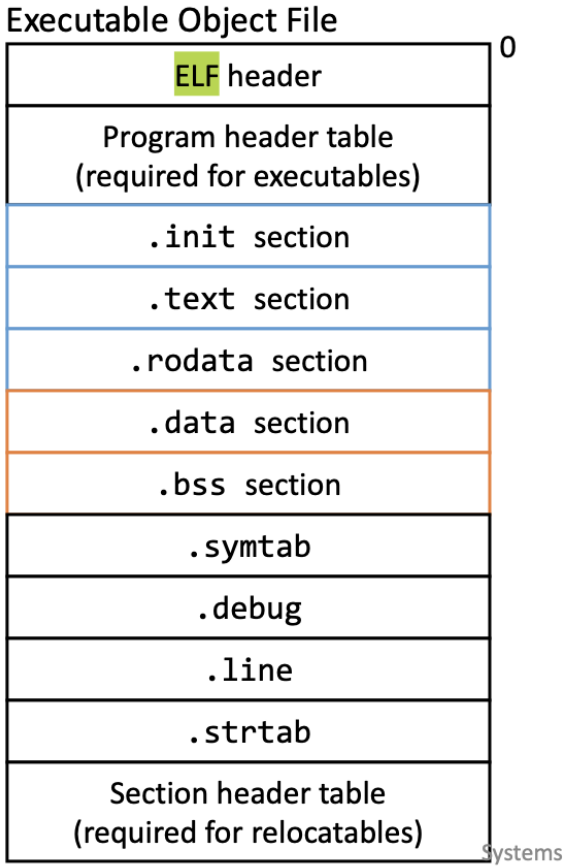
b) Which of the following is not a section of an ELF file?

(2 points)

- [x] `.code`
- [ ] `.data`
- [ ] `.bss`
- [ ] `.text`
- [ ] `.rodata`

Linking Exam Task (b)





Linking Exam Task (d)

Systems@ETH Zürich

d) Consider the following two blocks of code, which are contained in separate files:

```
// main.c
#include <stdio.h>

int i = 0;
int main() {
   foo();
   return 0;
}
```

```
// foo.c
#include <stdio.h>

int i = 1;
void foo() {
  printf("%d", i);
}
```

What will happen when you attempt to compile, link, and run this code?

(2 points)

- ☐ Compilation error
- ☐ Linking error
- ☐ Segmentation fault
- ☐ It will print 0
- ☐ It will print 1
- ☐ It may print 0 or it may print 1

Linking Exam Task (d)

Systems@ETH Zürich

d) Consider the following two blocks of code, which are contained in separate files:

```
// main.c
#include <stdio.h>

int i = 0;
int main() {
   foo();
   return 0;
}
```

```
// foo.c
#include <stdio.h>

int i = 1;
void foo() {
  printf("%d", i);
}
```

What will happen when you attempt to compile, link, and run this code?

(2 points)

- [ ] Compilation error
- [x] Linking error
- [ ] Segmentation fault
- [ ] It will print 0
- [ ] It will print 1
- [ ] It may print 0 or it may print 1

Linking Exam Task (d)

Systems@ETH Zürich

- **Compilation**: only a warning that we used a undeclared function foo() in main (as we dindnt #include): but this still compiles
- **Linking error**: as we have two strong initialized ints

Linking Exam Task (e)

Systems@ETH Zürich

e) Which of the following is an advantage of using static vs. dynamic libraries?

(2 points)

- ☐ Changes to the library do not require recompiling or re-linking all programs that use the library.
- ☐ Program executables can be smaller
- ☐ Program executables are self-contained
- ☐ Library code can be shared by multiple processes; the OS can map different program virtual addresses to the same physical address to share pages with library code.
- ☐ Most programs will run noticeably faster

Linking Exam Task (e)

Systems@ETH Zürich

e) Which of the following is an advantage of using static vs. dynamic libraries?

(2 points)

- [ ] Changes to the library do not require recompiling or re-linking all programs that use the library.
- [ ] Program executables can be smaller
- [x] Program executables are self-contained
- [ ] Library code can be shared by multiple processes; the OS can map different program virtual addresses to the same physical address to share pages with library code.
- [ ] Most programs will run noticeably faster

A few old exam tasks on Linker Symbols

HS20 Question 8

For each of the following identifiers, state whether they are a **strong linker symbol** definition, a **weak linker symbol** definition, a pure **declaration**, a **local** definition, or **none** of these.

Linker Symbols Exam Task

Consider the following C source code:

```
#include <stdio.h>

struct el { int i; struct el *next; };

extern unsigned long rr;
extern unsigned calc( int top, int bot );
int max_th = 42;
int min_th;

int main(int argc, char *argv[] )
{
    int count;

    for( count=0; count < 100; count++ )
        rr += calc( max_th, min_th );
    printf("%lu\n", rr);
    return 0;
}
```

el:

rr:

argv:

calc:

count:

main:

max_th:

min_th:

top:

Linker Symbols Exam Task

Consider the following C source code:

```
#include <stdio.h>

struct el { int i; struct el *next; };

extern unsigned long rr;
extern unsigned calc( int top, int bot );
int max_th = 42;
int min_th;

int main(int argc, char *argv[] )
{
    int count;

    for( count=0; count < 100; count++ )
        rr += calc( max_th, min_th );
    printf("%lu\n", rr);
    return 0;
}
```

- el is **none**. We're just declaring a struct type of name el, but there is no variable being declared here.
- rr is a **declaration**; it is declared as an extern.
- argv is **none**
- calc is a **declaration**; there is no definition of this procedure in this file, hence the extern and declaration.
- count is a **local** definition; it is a local variable inside the main function.
- main is a **strong** linker symbol
- max_th is a **strong** linker symbol
- min_th is a **weak** linker symbol
- top is **none**

Linker Symbols Exam Task

Systems@ETH Zürich

- **Structs**: are just a **type definition**, (like enums, typdefs etc.) they are not linker symbols because they don't allocate memory or have linkage themselves
- Defining struct in header file makes it usable in different files, but is still **no linker symbol**

- **Only** was we **instantiate** (define) the struct and then have a variable, linker symbols apply

Linker Symbols Exam Task

Systems@ETH Zürich

```
4   #include "mystruct.h"
5   struct el shared_element;  // This creates a strong symbol for shared_element
6
7   // file2.c
8   #include "mystruct.h"
9   extern struct el shared_element;
```

```
3   // mystruct.h
4   struct el {
5       int i;
6       struct el *next;
7   };
8
9   // file1.c
10  #include "mystruct.h"
11  struct el element1;  // file1.c can use struct el
12
13  // file2.c
14  #include "mystruct.h"
15  struct el element2;  // file2.c can also use struct el
```

Compilation Pipeline

We have the following C files:

```
int buf[2] = {1, 2};

int main()
{
  swap();
  return 0;
}
```

main.c

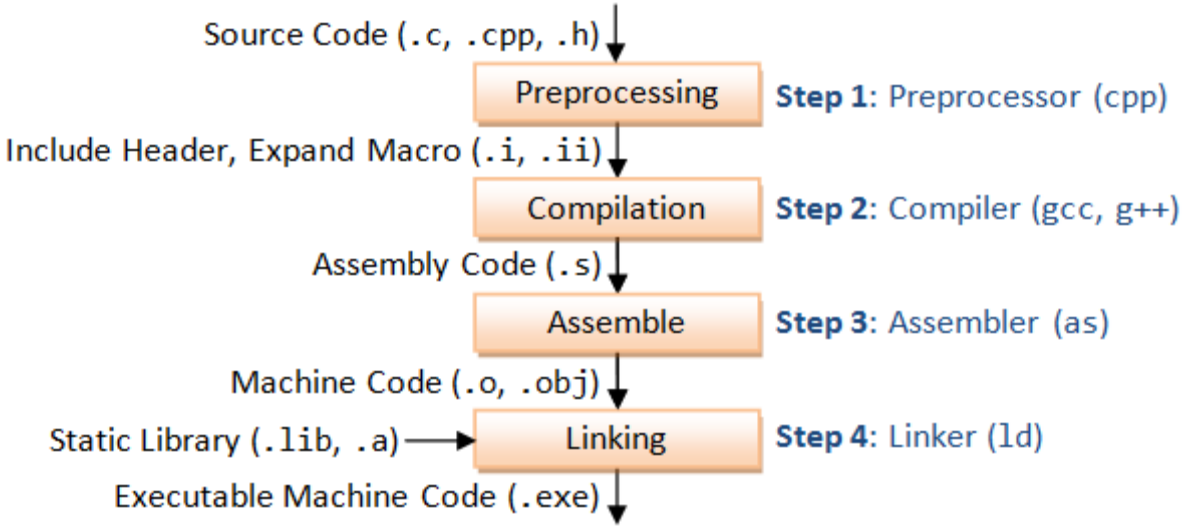
```
extern int buf[];

static int *bufp0 = &buf[0];
static int *bufp1;

void swap()
{
  int temp;

  bufp1 = &buf[1];
  temp = *bufp0;
  *bufp0 = *bufp1;
  *bufp1 = temp;
}
```

swap.c

Compilation Pipeline

Programs are translated and linked using a *compiler driver*:

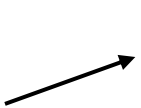
```
unix> gcc -O2 -g -o main main.c swap.c
unix> ./main
```



Compilation Pipeline


```
$ gcc -v main.c swap.c -o main
/usr/lib/gcc/x86_64-linux-gnu/4.8/cc1 -quiet -v -imultiarch x86_64-linux-gnu
main.c -quiet -dumpbase main.c -mtune=generic -march=x86-64 -auxbase main -
version -fstack-protector -Wformat -Wformat-security -o /tmp/ccmA4h2y.s
[…]
as -v --64 -o /tmp/cc6h3tRf.o /tmp/ccmA4h2y.s
[…]
/usr/lib/gcc/x86_64-linux-gnu/4.8/cc1 -quiet -v -imultiarch x86_64-linux-gnu
swap.c -quiet -dumpbase swap.c -mtune=generic -march=x86-64 -auxbase swap -
version -fstack-protector -Wformat -Wformat-security -o /tmp/ccmA4h2y.s
[…]
as -v --64 -o /tmp/ccecqv3W.o /tmp/ccmA4h2y.s
[…]
```

Compilation Pipeline




```
/usr/lib/gcc/x86_64-linux-gnu/4.8/collect2 --sysroot=/ --build-id --eh-
frame-hdr -m elf_x86_64 --hash-style=gnu --as-needed -dynamic-linker
/lib64/ld-linux-x86-64.so.2 -z relro -o main /usr/lib/gcc/x86_64-linux-
gnu/4.8/../../../x86_64-linux-gnu/crt1.o /usr/lib/gcc/x86_64-linux-
gnu/4.8/../../../x86_64-linux-gnu/crti.o /usr/lib/gcc/x86_64-linux-
gnu/4.8/crtbegin.o -L/usr/lib/gcc/x86_64-linux-gnu/4.8 -
L/usr/lib/gcc/x86_64-linux-gnu/4.8/../../../x86_64-linux-gnu -
L/usr/lib/gcc/x86_64-linux-gnu/4.8/../../../../lib -L/lib/x86_64-linux-gnu -
L/lib/../lib -L/usr/lib/x86_64-linux-gnu -L/usr/lib/../lib -
L/usr/lib/gcc/x86_64-linux-gnu/4.8/../../.. /tmp/cc6h3tRf.o /tmp/ccecqv3W.o
-lgcc --as-needed -lgcc_s --no-as-needed -lc -lgcc --as-needed -lgcc_s --no-
as-needed /usr/lib/gcc/x86_64-linux-gnu/4.8/crtend.o /usr/lib/gcc/x86_64-
linux-gnu/4.8/../../../x86_64-linux-gnu/crtn.o
```

* https://gcc.gnu.org/onlinedocs/gccint/Collect2.html

A Closer Look into a Binary

```
$ file main
```

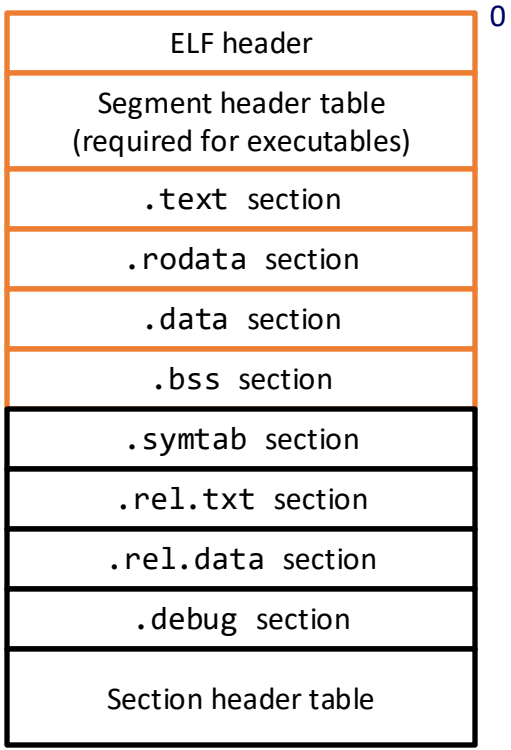
main: ELF 64-bit LSB executable, x86-64, version 1 (SYSV), dynamically linked (uses shared libs), for GNU/Linux

```
$ hexdump -C main
```

```
00000000  7f 45 4c 46 02 01 01 00  00 00 00 00 00 00 00 00
|.ELF............|
00000010  02 00 3e 00 01 00 00 00  90 48 40 00 00 00 00 00
|..>.....H@.....|
[…]
```

ELF Object File Format

- ELF header
- Segment header table
- `.text` section
- `.rodata` section
- `.data` section
- `.bss` section

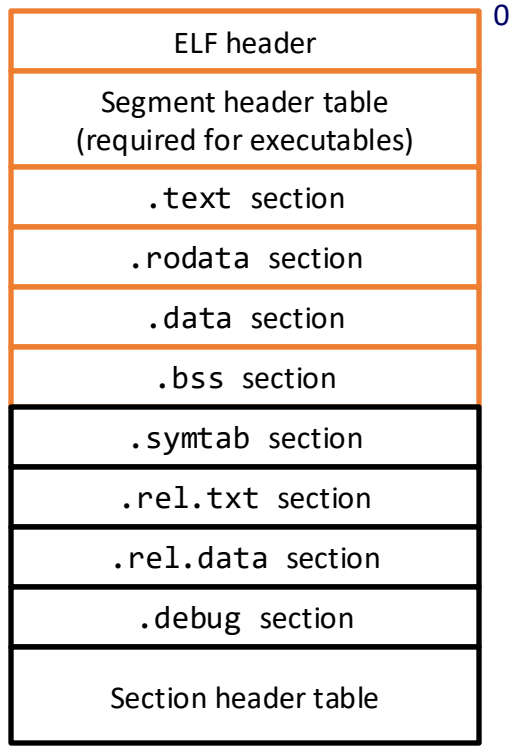


ELF Sections

- .data
  - contains global tables, variables
- .text
  - this is where the code of your program is
- .bss
  - “better save space” i.e. all your uninitialized variables

ELF Header Format

```
typedef struct {
        unsigned char e_ident[EI_NIDENT];
        Elf32_Half    e_type;
        Elf32_Half    e_machine;
        Elf32_Word    e_version;
        Elf32_Addr    e_entry;
        Elf32_Off     e_phoff;
        Elf32_Off     e_shoff;
        Elf32_Word    e_flags;
        Elf32_Half    e_ehsize;
        Elf32_Half    e_phentsize;
        Elf32_Half    e_phnum;
        Elf32_Half    e_shentsize;
        Elf32_Half    e_shnum;
        Elf32_Half    e_shstrndx;
        Elf32_Ehdr;
}
```

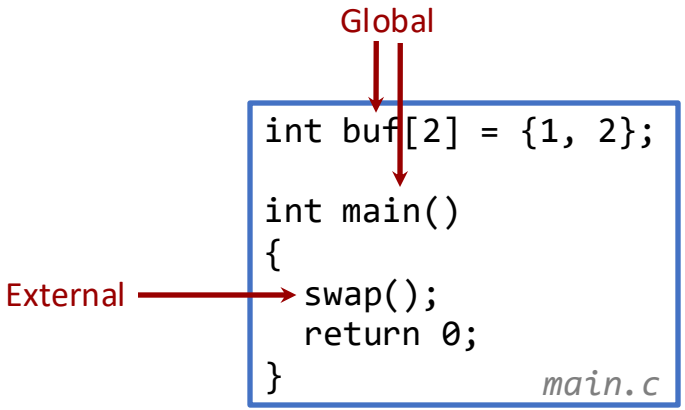


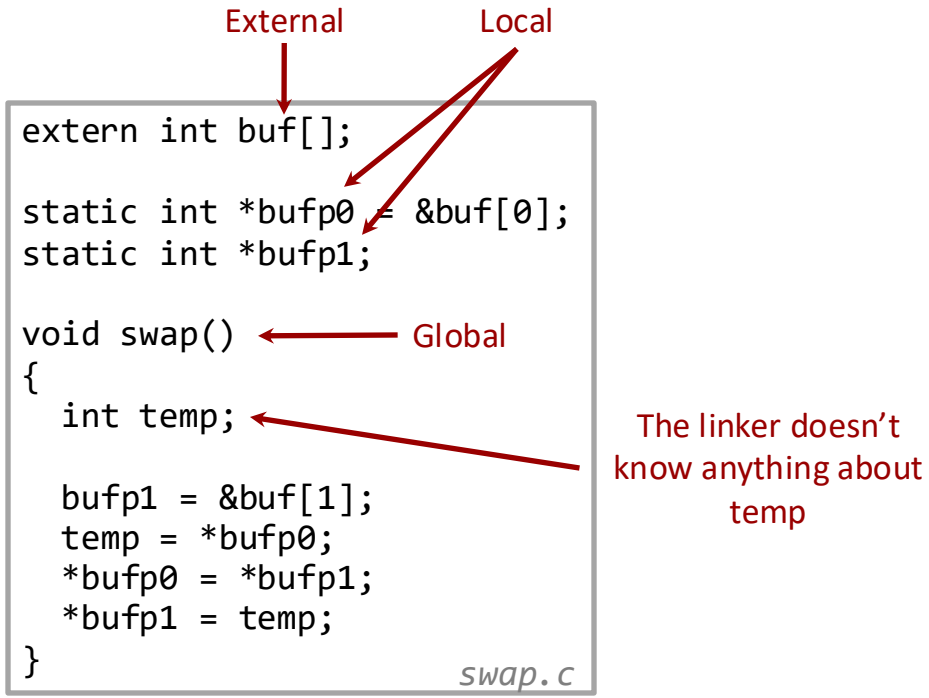
ELF Header Format

```
$ readelf --header main
```

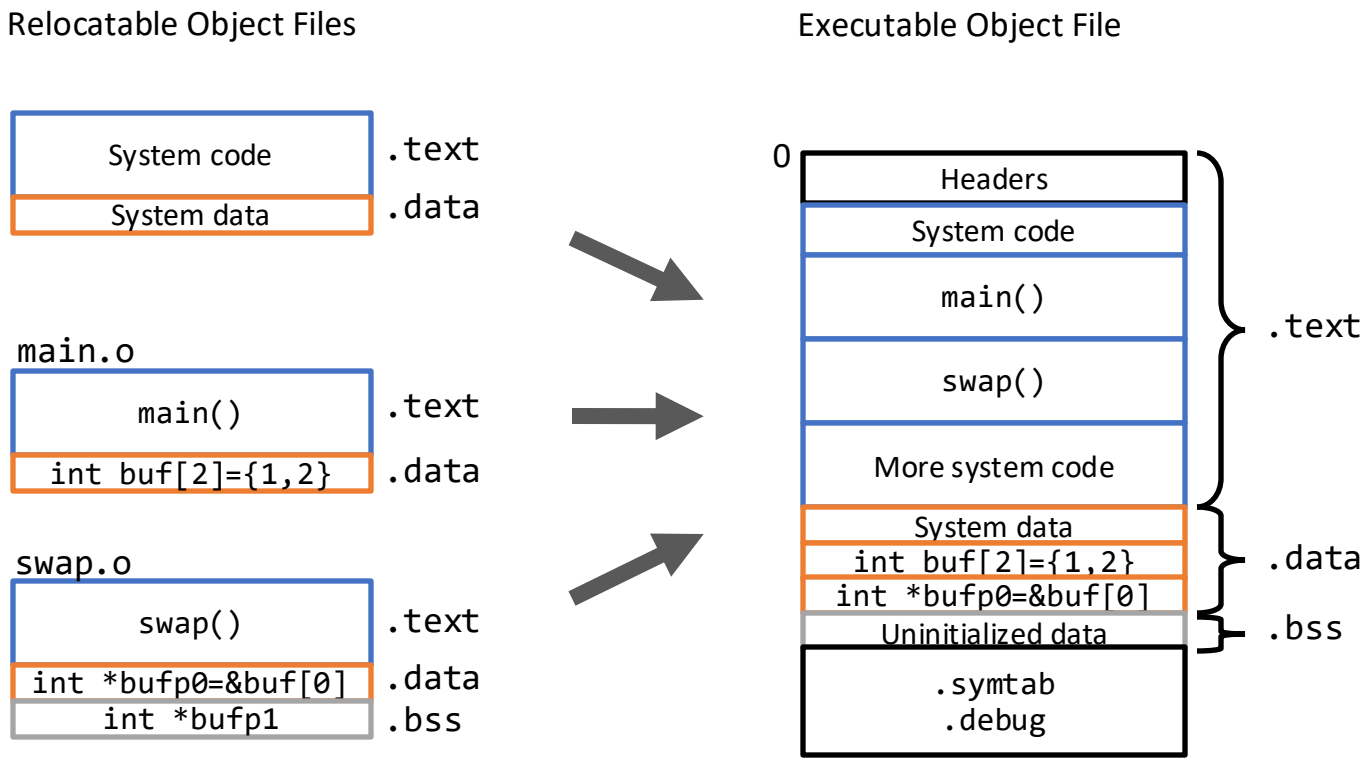
```
ELF Header:
Magic:   7f 45 4c 46 02 01 01 00 00 00 00 00 00 00 00 00
Class:                             ELF64
Data:                              2's complement, little endian
Version:                           1 (current)
OS/ABI:                            UNIX - System V
[…]
Entry point address:               0x404890
[…]
Start of program headers:          64 (bytes into file)
Start of section headers:          108288 (bytes into file)
[…]
```

Resolving Symbols



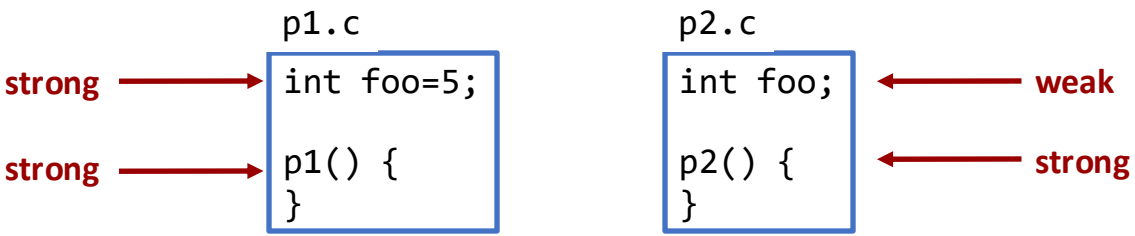


Relocating Code and Data



Strong and Weak Symbols

- Program symbols are either strong or weak
  - ***Strong***: procedures and initialized globals
  - ***Weak***: uninitialized globals



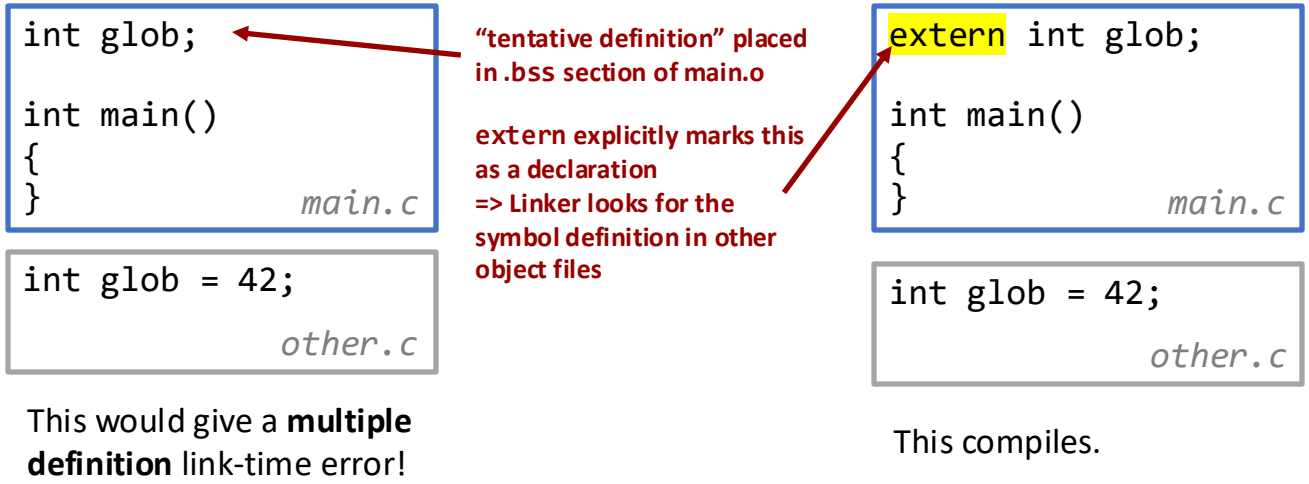
The Linker's Symbol Rules

*Assume the compiler flag `-fcommon` is used

- **Rule 1:** Multiple strong symbols are not allowed
  - Each item can be defined only once
  - Otherwise: Linker error
- **Rule 2:** Given a strong symbol and multiple weak symbol, choose the strong symbol
  - References to the weak symbol resolve to the strong symbol
- **Rule 3:** If there are multiple weak symbols, pick an arbitrary one

New Default: -fno-common

- Recent versions of `gcc` and `clang` use the `-fno-common` flag by default.



This would give a **multiple definition** link-time error!

This compiles.

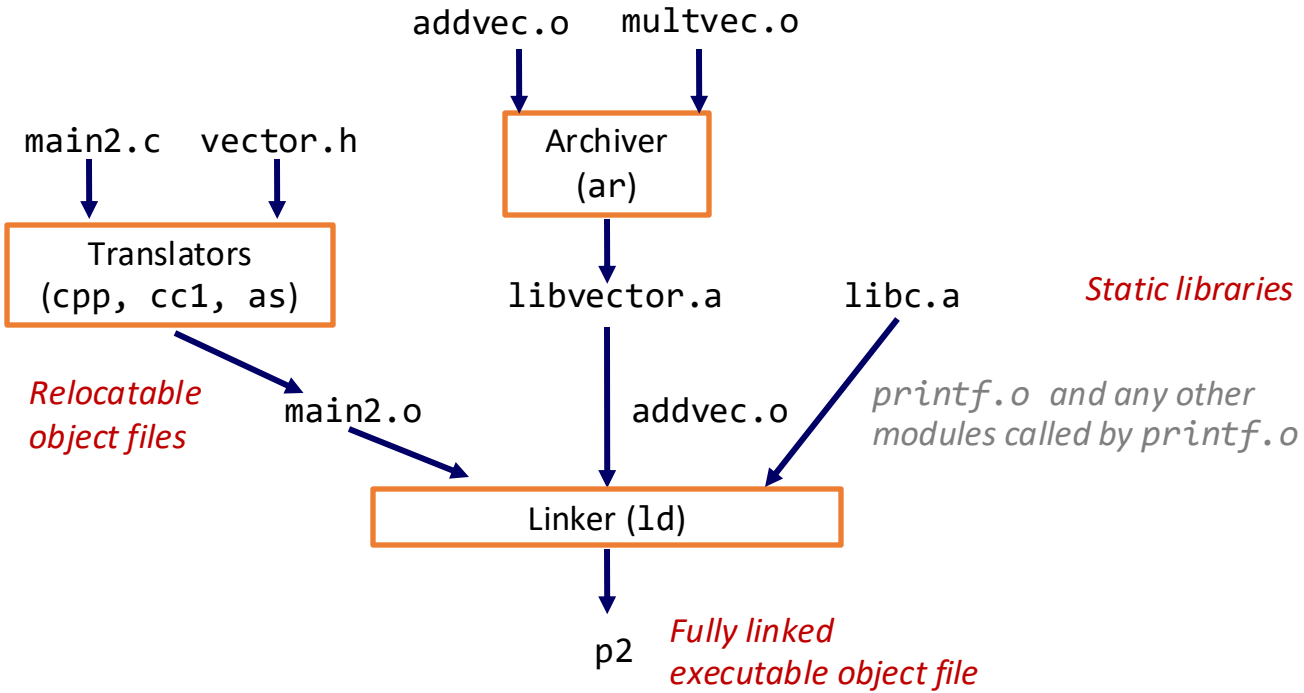
- Use `extern` in every declaration.
  The only place without `extern` must be the (possibly tentative) definition.

Static Libraries

Systems@ETH Zürich

- Static libraries (`.a` archive files)
  - Concatenate related relocatable object files into a single file with an index (called an *archive*).
  - Enhance linker so that it tries to resolve unresolved external references by looking for the symbols in one or more archives.
  - If an archive member file resolves reference, link into executable.

Linking with Static Libraries



Using Static Libraries

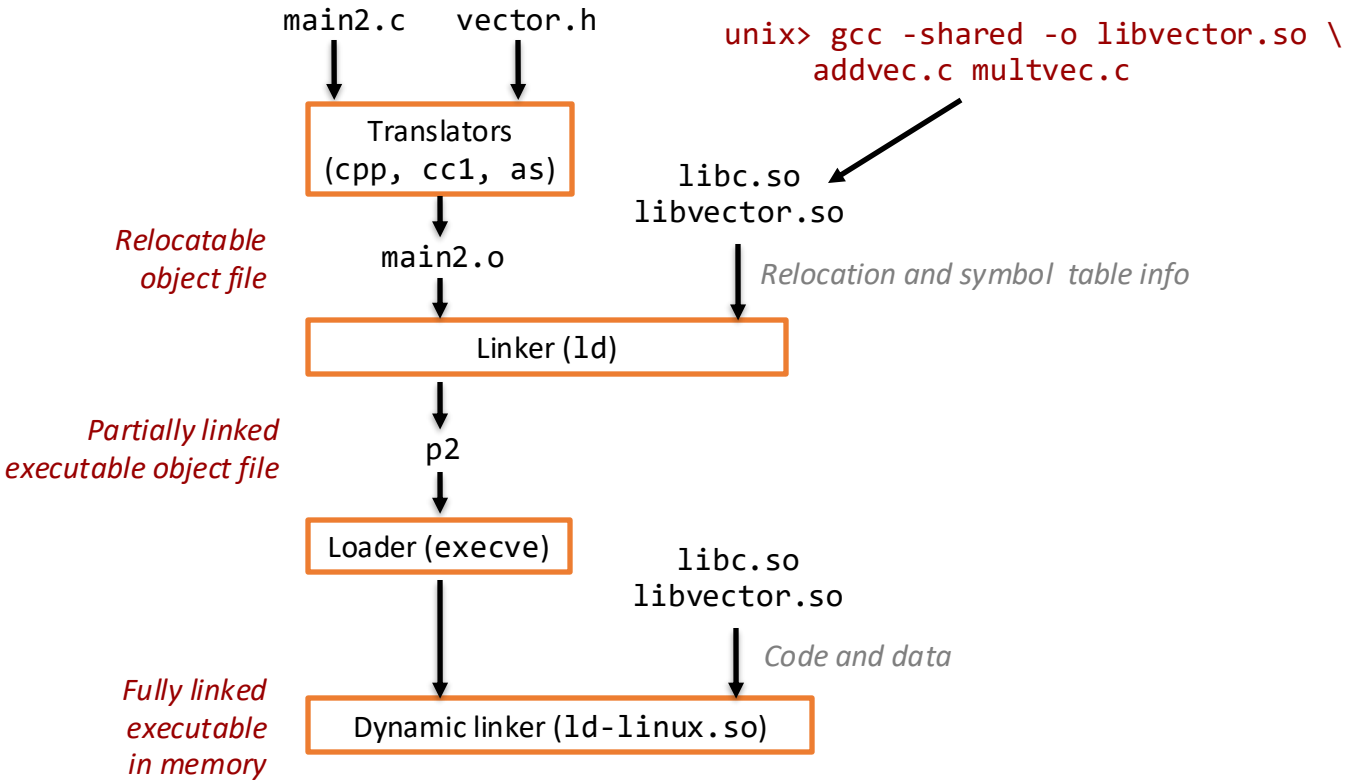
Systems@ETH Zürich

- Linker's algorithm for resolving external references:
  - Scan `.o` files and `.a` files in the command line order.
  - During the scan, keep a list of the current unresolved references.
  - As each new `.o` or `.a` file, *obj*, is encountered, try to resolve each unresolved reference in the list against the symbols defined in *obj*.
  - If any entries in the unresolved list at end of scan, then error.

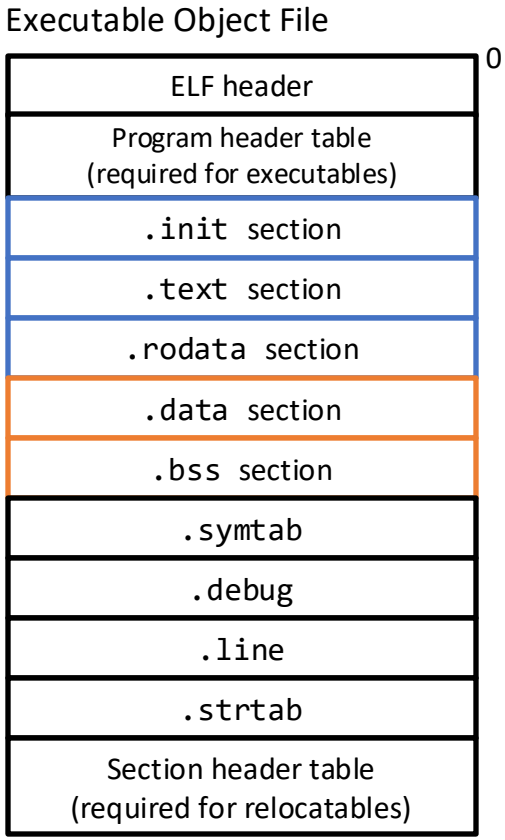
- Problem:
  - Command line order matters!
  - Moral: put libraries at the end of the command line.

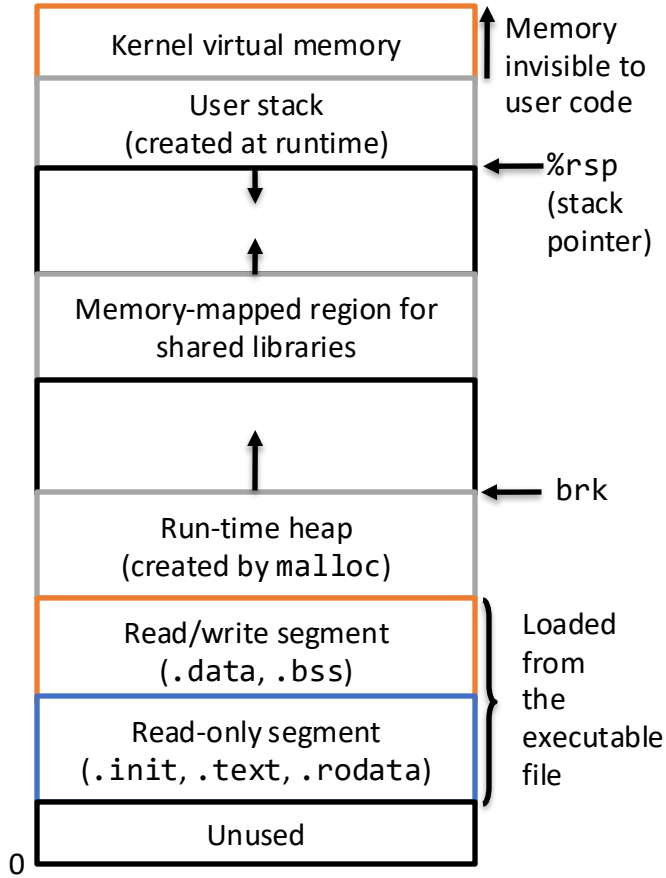
```
unix> gcc -L. libtest.o -lmine
unix> gcc -L. -lmine libtest.o
libtest.o: In function `main':
libtest.o(.text+0x4): undefined reference to `libfun'
```

Dynamic Linking at Load-Time



Loading Executable Object Files



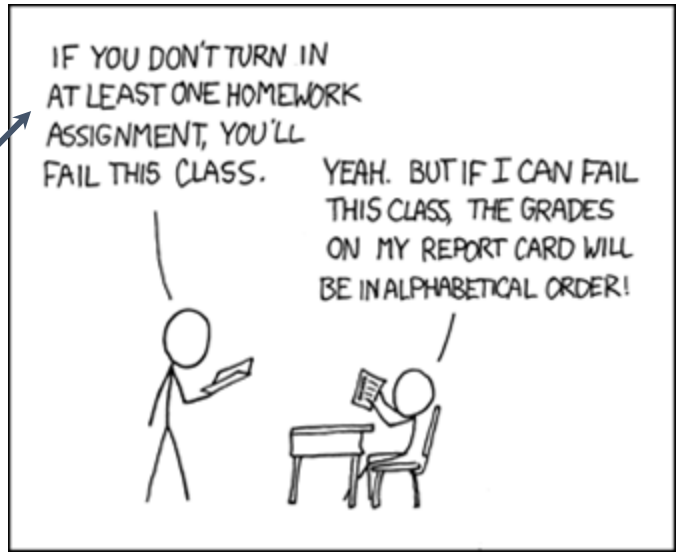


Library Linking Demo

Assignment 6

- Pen-and-Paper exercises
- If you wish to receive feedback - Submit as PDF via email or GitLab

Not the case for our class :)



HS19 – Question 10

For each symbol listed in the following C object file, say whether it is a **strong linker symbol,** a **weak linker symbol**, a **macro**, a symbol **local** to the compilation unit, on **none** of these.

(Assume –fcommon used)

```
#define string "hello, world"
static int count;
extern char *otherstring;
struct element {
    void            *data;
    struct element *next;
};
struct element *head = NULL;
struct element *tail;

void push( struct element *e )
{
    e->next = head;
    head    = e;
}
struct element *pull();
```

- `count:` local ← Declared as static
- `element:` none ← Simply a struct declaration
- `head:` strong ← Initialized global
- `otherstring:` none ← Not used, so no symbol generated
- `pull:` none ← Not used, so no symbol generated
- `push:` strong ← Function definitions are strong
- `string:` macro ← Self-explanatory
- `tail:` weak ← Uninitialized global

Systems@ETH Zürich

Questions?