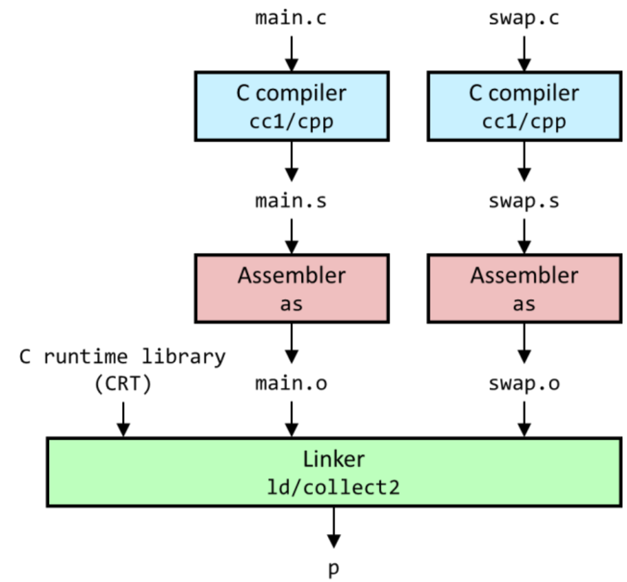
REVE1

Program Execution

```
 1:  48 89 e5                          mov    %rsp,%rbp
 4:  48 c7 05 00 00 00 00              movq   $0x0,0x0(%rip)
 b:  00 00 00 00
        7: R_X86_64_PC32  .bss-0x8
        b: R_X86_64_32S   buf+0x4
 f:  48 8b 05 00 00 00 00              mov    0x0(%rip),%rax
       12: R_X86_64_PC32 bufp0-0x4
16:  8b 00                             mov    (%rax),%eax
18:  89 45 fc                          mov    %eax,-0x4(%rbp)
1b:  48 8b 05 00 00 00 00              mov    0x0(%rip),%rax
       1e: R_X86_64_PC32 bufp0-0x4
22:  48 8b 15 00 00 00 00              mov    0x0(%rip),%rdx
       25: R_X86_64_PC32 .bss-0x4
29:  8b 12                             mov    (%rdx),%edx
```

Module Outline

- The Life Cycle of a Program
- Linking Overview
- Executable and Linkable Format
- Linking
  - Symbol Resolution
  - Symbol Relocation
- Module Summary



Life Cycle of a Program

HOCHSCHULE LUZERN

A Simple C Program

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

int main() {
  printf("[ %d, %d ]\n",
         buf[0],buf[1]);
  swap();
  printf("[ %d, %d ]\n",
         buf[0],buf[1]);
  return EXIT_SUCCESS;
}
```
main.c

```c
extern int buf[];

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

void swap() {
  int temp;

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

- Compile to executable program with GCC

```
$ gcc -O2 -Wall -g -o p main.c swap.c
main.c: In function ‘main’:
main.c:9:3: warning: implicit declaration of function ‘swap’
    9 |   swap();
      |   ^~~~
$ ./p
[ 1, 2 ]
[ 2, 1 ]
$
```

A Simple C Program

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

int main() {
  printf(…);
  swap();
  printf(…);
}                main.c
```

```c
extern int buf[];

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

void swap() {
  int temp;

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

- **GCC is, in fact, a *compiler driver***
- Executes a series of commands that transform the source code into an executable

```
$ gcc -v -O2 -Wall -g -o p main.c swap.c
Using built-in specs.
COLLECT_GCC=gcc
COLLECT_LTO_WRAPPER=/usr/libexec/gcc/x86_64-pc-linux-gnu/11.3.0/lto-wrapper
Target: x86_64-pc-linux-gnu
Configured with: /var/tmp/portage/sys-devel/gcc-11.3.0/work/gcc-11.3.0/configure
--host=x86_64-pc-linux-gnu --build=x86_64-pc-linux-gnu --prefix=/usr
--bindir=/usr/x86_64-pc-linux-gnu/gcc-bin/11.3.0 --includedir=…
Thread model: posix
…
```


HOCHSCHULE LUZERN

A Simple C Program

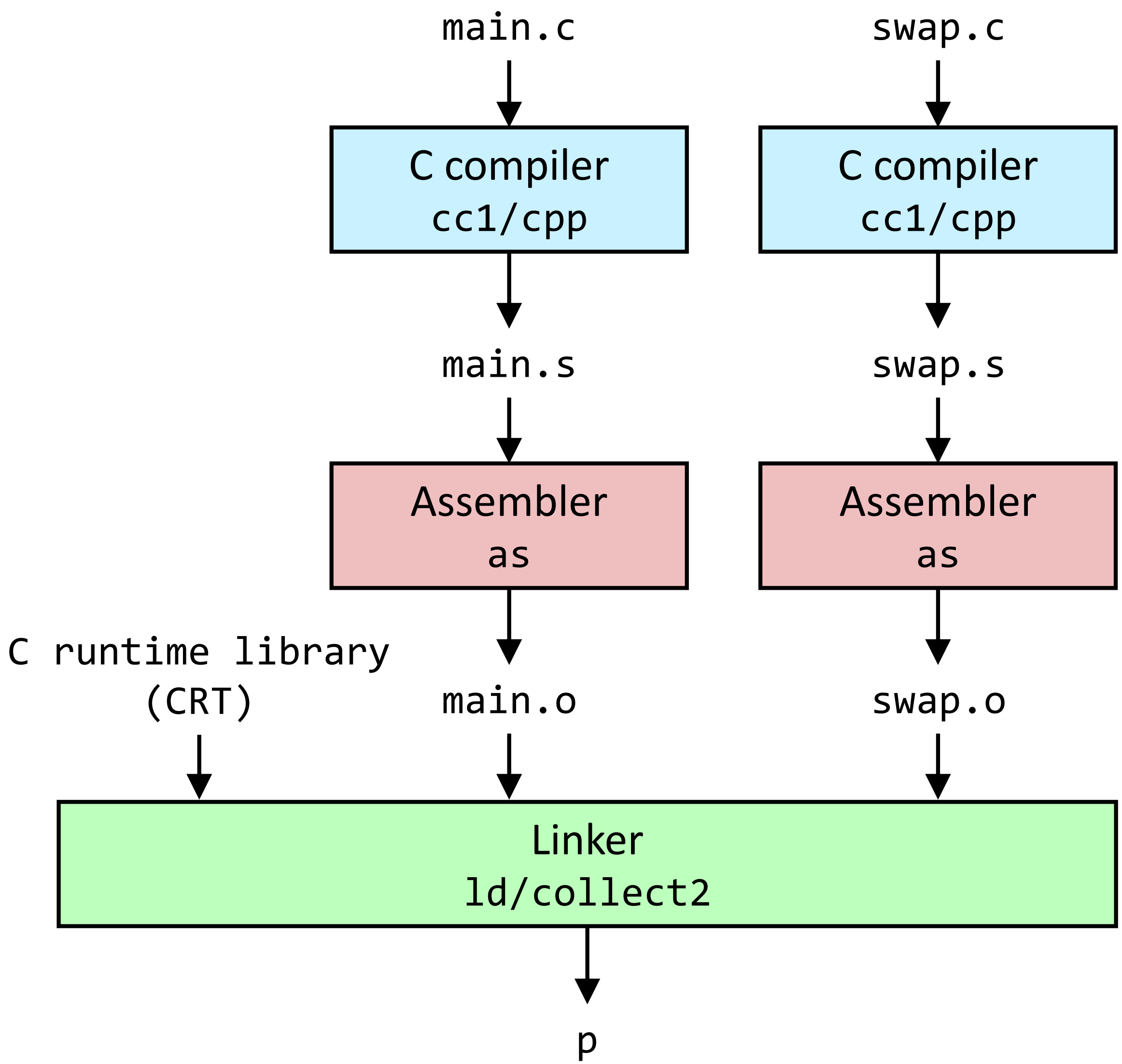
```c
int buf[2] = {1, 2};

int main() {
  printf(…);
  swap();
  printf(…);
  return 0;
}                main.c
```

```c
extern int buf[];

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

void swap() {
  int temp;

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

- **Making sense of GCC's compilation steps**

```
$ ( gcc -v -O2 -Wall -g -o p main.c swap.c 2>&1 ) | \
  grep "^ /usr/lib" | grep -v include | \
  grep -E "^ /usr[^ ]*
 /usr/libexec/gcc/x86_64-pc-linux-gnu/11.3.0/cc1 -quiet -v main.c -quiet -dumpdir p- -dumpbase
main.c -dumpbase-ext .c -mtune=generic -march=x86-64 -g -O2 -Wall -version -o /tmp/ccUMd0qR.s
 /usr/lib/gcc/x86_64-pc-linux-gnu/11.3.0/../../../../x86_64-pc-linux-gnu/bin/as -v --gdwarf-5 --64
-o /tmp/ccFGruA6.o /tmp/ccUMd0qR.s
 /usr/libexec/gcc/x86_64-pc-linux-gnu/11.3.0/cc1 -quiet -v swap.c -quiet -dumpdir p- -dumpbase
swap.c -dumpbase-ext .c -mtune=generic -march=x86-64 -g -O2 -Wall -version -o /tmp/ccUMd0qR.s
 /usr/lib/gcc/x86_64-pc-linux-gnu/11.3.0/../../../../x86_64-pc-linux-gnu/bin/as -v --gdwarf-5 --64
-o /tmp/ccyXZYrW.o /tmp/ccUMd0qR.s
 /usr/libexec/gcc/x86_64-pc-linux-gnu/11.3.0/collect2 -plugin /usr/libexec/gcc/x86_64-pc-linux-
gnu/11.3.0/liblto_plugin.so -plugin-opt=/usr/libexec/gcc/x86_64-pc-linux-gnu/11.3.0/lto-wrapper -
plugin-opt=-fresolution=/tmp/ccU0tDGc.res -plugin-opt=-pass-through=-lgcc -plugin-opt=-pass-
through=-lgcc_s -plugin-opt=-pass-through=-lc -plugin-opt=-pass-through=-lgcc -plugin-opt=-pass-
through=-lgcc_s --eh-frame-hdr -m elf_x86_64 -dynamic-linker /lib64/ld-linux-x86-64.so.2 -pie -o p
/usr/lib/gcc/x86_64-pc-linux-gnu/11.3.0/../../../../lib64/Scrt1.o /usr/lib/gcc/x86_64-pc-linux-
gnu/11.3.0/../../../../lib64/crti.o /usr/lib/gcc/x86_64-pc-linux-gnu/11.3.0/crtbeginS.o -
L/usr/lib/gcc/x86_64-pc-linux-gnu/11.3.0 -L/usr/lib/gcc/x86_64-pc-linux-
gnu/11.3.0/../../../../lib64 -L/lib/../lib64 -L/usr/lib/../lib64 -L/usr/lib/gcc/x86_64-pc-linux-
gnu/11.3.0/../../../../x86_64-pc-linux-gnu/lib -L/usr/lib/gcc/x86_64-pc-linux-gnu/11.3.0/../../..
/tmp/ccFGruA6.o /tmp/ccyXZYrW.o -lgcc --push-state --as-needed -lgcc_s --pop-state -lc -lgcc --
push-state --as-needed -lgcc_s --pop-state /usr/lib/gcc/x86_64-pc-linux-gnu/11.3.0/crtendS.o
/usr/lib/gcc/x86_64-pc-linux-gnu/11.3.0/../../../../lib64/crtn.o
```

A Simple C Program



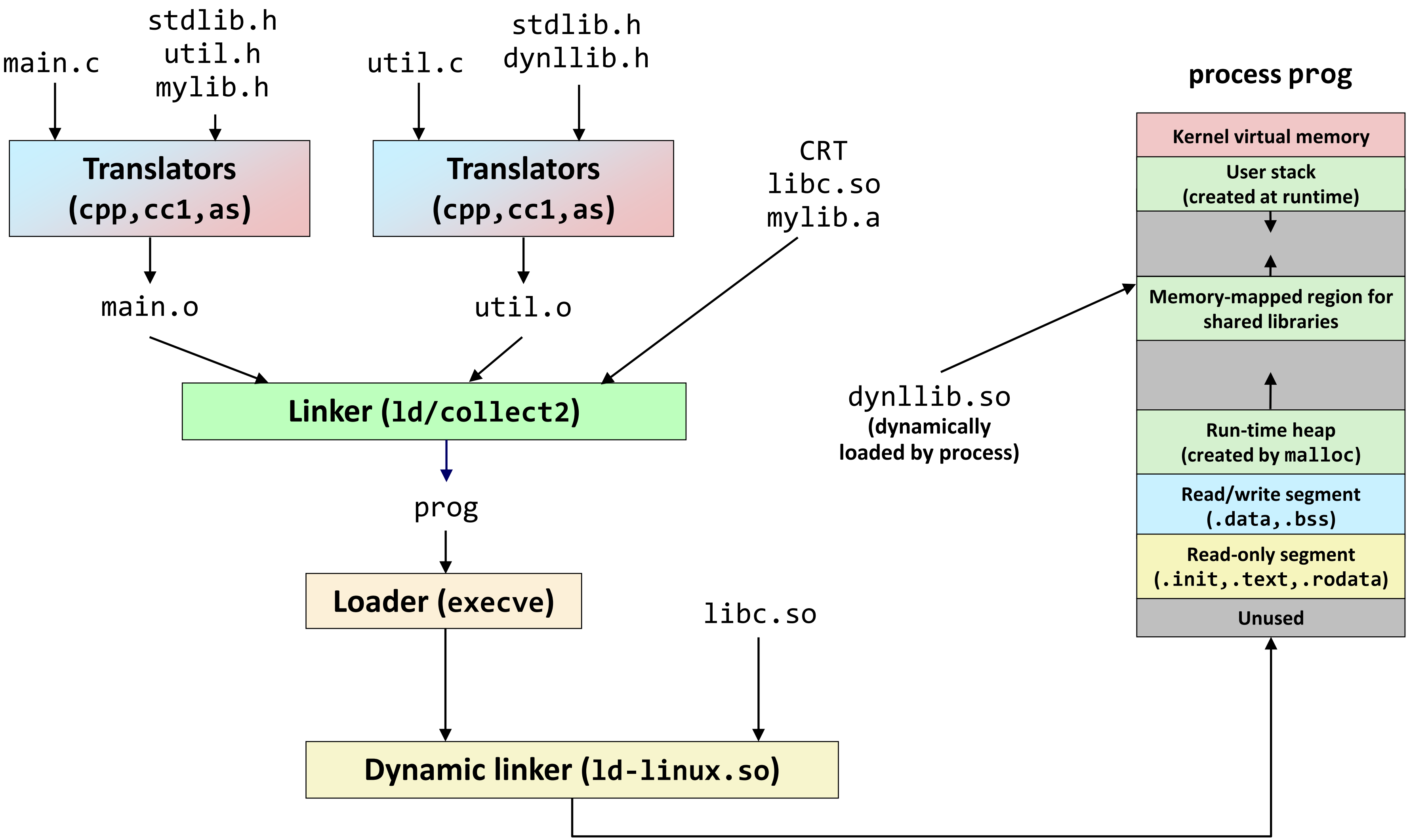
Source files

Separately compiled *assembly* files

Separately assembled *relocatable* object files

Fully linked *executable* object file

From Source Code to a Running Process



HOCHSCHULE LUZERN

Problems to Solve

- **Separate compilation of individual C files**
  - How does the compiler know about functions and variables defined in other files to make sure the type matches?
  - How can we generate addresses to call functions / access variables if we do not know where they are located in memory?
  - How does the system know which dynamic libraries to load when executing a binary?
  - How does the application know how to call functions from dynamic libraries?

```
$ gcc -c main.c
$ readelf -s main.o
…
 Num: Size Type    Bind   Ndx Name
   9:    8 OBJECT  GLOBAL   3 buf
  10:   83 FUNC    GLOBAL   1 main
  13:    0 NOTYPE  GLOBAL UND swap
```

Linking Overview

HOCHSCHULE
LUZERN

Why Linkers?

- **Reason 1: Modularity**
  - Program can be written as a collection of smaller source files, rather than one monolithic mass.
  - Can build libraries of common functions (more on this later)
    - e.g., Math library, standard C library

HOCHSCHULE LUZERN

Why Linkers? (cont)

- **Reason 2: Efficiency**

  - Time: Separate compilation
    - Change one source file, compile, and then relink.
    - No need to recompile other source files.

  - Space: Libraries
    - Common functions can be aggregated into a single file...
    - yet executable files and running memory images contain only code for the functions they actually use.

What Do Linkers Do?

- **Step 1. Symbol resolution**

  - Programs define and reference symbols (variables and functions):
    - void swap() {…} /* define symbol swap */
    - swap(); /* reference symbol swap */
    - int *bufp0 = &buf[0]; /* define symbol bufp0, reference buf */

  - Symbol definitions are stored (by the compiler) in a symbol table.
    - Symbol table is an array of structs
    - Each entry includes name, size, and location of symbol

  - Linker associates each symbol reference with exactly one symbol definition.

HOCHSCHULE LUZERN

What Do Linkers Do? (cont)

- **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.

HOCHSCHULE LUZERN

ELF header
Segment header table (required for executables)
`.text` section
`.rodata` section
`.data` section
`.bss` section
`.symtab` section
`.rel.txt` section
`.rel.data` section
`.debug/.line` section
Section header table

Executable and Linkable Format (ELF)

Three Kinds of Object Files (Modules)

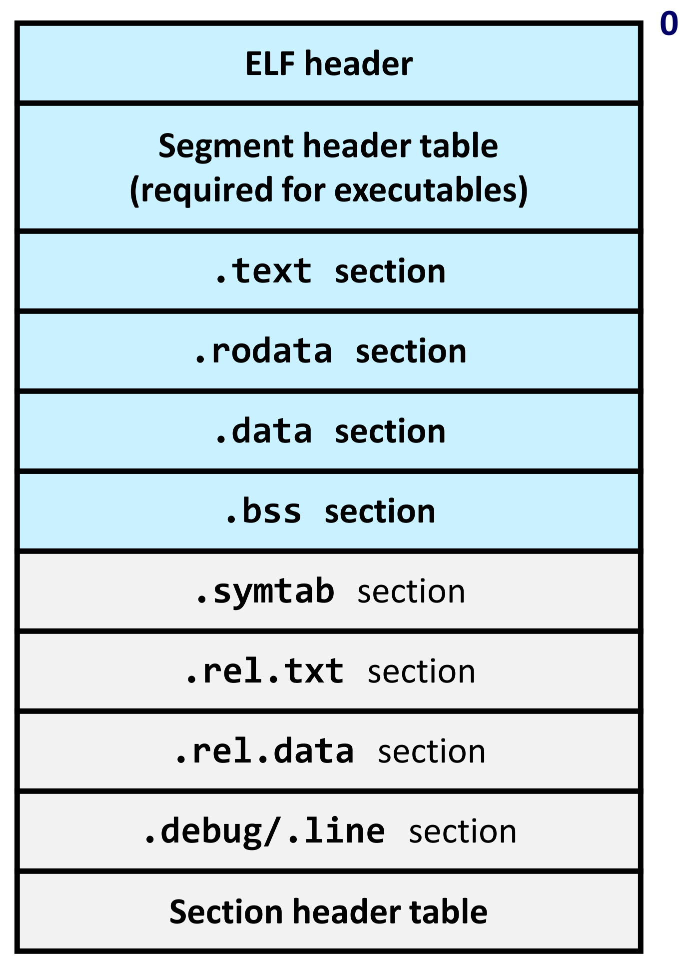
- **Relocatable object file (.o file)**
  - Contains code and data in a form that can be combined with other relocatable object files to form executable object file.
    - Each .o file is produced from exactly one source (.c) file
- **Executable object file (a.out file)**
  - Contains code and data in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

Executable and Linkable Format (ELF)

- Standard binary format for object files
- Originally proposed by AT&T System V Unix
  - Later adopted by BSD Unix variants and Linux
- One unified format for
  - Relocatable object files (.o),
  - Executable object files (a.out)
  - Shared object files (.so)
- Generic name: ELF binaries

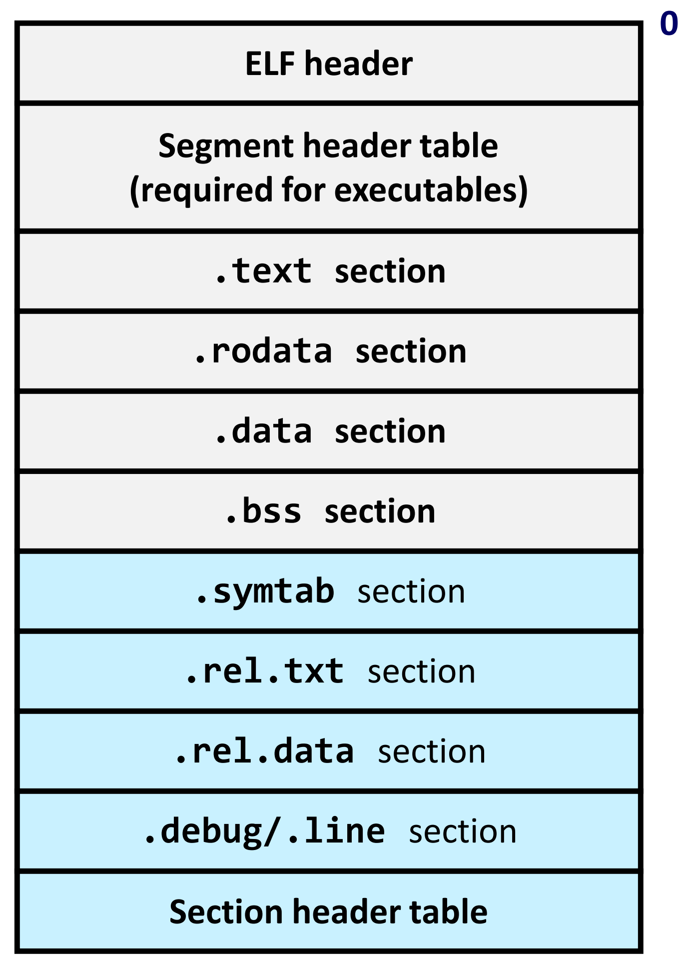
ELF Object File Format

- **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: printf strings, jump tables, ...
- **.data section**
  - Initialized global variables
- **.bss section**
  - Uninitialized global variables
  - “Block Started by Symbol”
  - Has section header but occupies no space



ELF Object File Format (cont.)

- **.symtab section**
  - Symbol table
  - Procedure and static variable names
  - Section names and locations
- **.rel.text section**
  - Relocation info for .text section
  - Addresses of instructions that will need to be modified in the executable
  - Instructions for modifying.
- **.rel.data section**
  - Relocation info for .data section
  - Addresses of pointer data that will need to be modified in the merged executable
- **.debug/.line section**
  - Info for symbolic debugging (gcc -g)
- **Section header table**
  - Offsets and sizes of each section



HOCHSCHULE LUZERN

Analyzing ELF Files with Readelf

```
$ gcc -O2 -c linking.c                                          (gcc 11.3.0)
$ readelf -a linking.o
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
  ABI Version:                       0
  Type:                              REL (Relocatable file)
  Machine:                           Advanced Micro Devices X86-64
…

Section Headers:
  [Nr] Name              Type             Address           Offset
       Size              EntSize          Flags  Link  Info  Align
  [ 0]                   NULL             0000000000000000  00000000
       0000000000000000  0000000000000000           0     0     0
  [ 1] .text             PROGBITS         0000000000000000  00000040
       0000000000000000  0000000000000000  AX       0     0     1
…

Relocation section '.rela.text.startup' at offset 0x230 contains 9 entries:
  Offset          Info           Type           Sym. Value    Sym. Name + Addend
000000000002  000200000002 R_X86_64_PC32     0000000000000000 .bss - 8
…
00000000004e  000800000002 R_X86_64_PC32     0000000000000000 chksum - 4
…

Symbol table '.symtab' contains 9 entries:
   Num:    Value          Size Type    Bind   Vis      Ndx Name
     …
     4: 0000000000000000     4 OBJECT  LOCAL  DEFAULT    3 i
     5: 0000000000000000    83 FUNC    GLOBAL DEFAULT    4 main
     6: 0000000000000000     0 NOTYPE  GLOBAL DEFAULT  UND foo
     7: 0000000000000000     4 OBJECT  GLOBAL DEFAULT    2 counter
     8: 0000000000000000     0 NOTYPE  GLOBAL DEFAULT  UND chksum
…
```

- -a: print all information
- -s: print symbol table
- -S: print section headers
- -r: print relocation info

HOCHSCHULE LUZERN

Analyzing ELF Files with Readelf

```
$ gcc -O2 -c linking.c                                           (gcc 11.3.0)
$ readelf -Ss linking.o
There are 14 section headers, starting at offset 0x3a0:

Section Headers:
  [Nr] Name              Type             Address           Offset
       Size              EntSize          Flags  Link  Info  Align
  …
  [ 1] .text             PROGBITS         0000000000000000  00000040
       0000000000000000  0000000000000000  AX       0     0     1
  [ 2] .data             PROGBITS         0000000000000000  00000040
       0000000000000004  0000000000000000  WA       0     0     4
  [ 3] .bss              NOBITS           0000000000000000  00000044
       0000000000000004  0000000000000000  WA       0     0     4
  [ 4] .text.startup     PROGBITS         0000000000000000  00000050
       0000000000000053  0000000000000000  AX       0     0     16
  …
  [11] .symtab           SYMTAB           0000000000000000  00000130
       00000000000000d8  0000000000000018          12     5     8
  [12] .strtab           STRTAB           0000000000000000  00000208
       0000000000000025  0000000000000000           0     0     1
  …

Symbol table '.symtab' contains 9 entries:
   Num:    Value          Size Type    Bind   Vis      Ndx Name
     0: 0000000000000000     0 NOTYPE  LOCAL  DEFAULT  UND
     1: 0000000000000000     0 FILE    LOCAL  DEFAULT  ABS linking.c
     2: 0000000000000000     0 SECTION LOCAL  DEFAULT    3 .bss
     3: 0000000000000000     0 SECTION LOCAL  DEFAULT    4 .text.startup
     4: 0000000000000000     4 OBJECT  LOCAL  DEFAULT    3 i
     5: 0000000000000000    83 FUNC    GLOBAL DEFAULT    4 main
     6: 0000000000000000     0 NOTYPE  GLOBAL DEFAULT  UND foo
     7: 0000000000000000     4 OBJECT  GLOBAL DEFAULT    2 counter
     8: 0000000000000000     0 NOTYPE  GLOBAL DEFAULT  UND chksum
```

- -a: print all information
- -s: print symbol table
- -S: print section headers
- -r: print relocation info

UND: undefined
(as in "we don't know yet")

HOCHSCHULE LUZERN

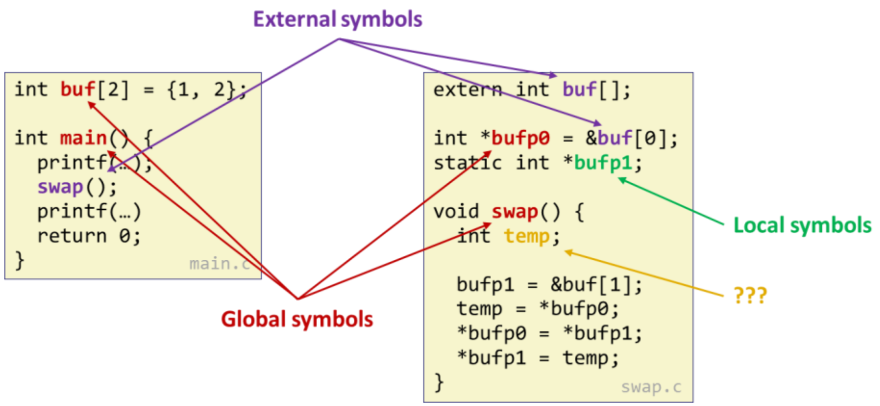
Analyzing ELF Files with Readelf

```
$ gcc -c linking.c                                              (gcc 10.3.0)
$ readelf -Ss linking.o
There are 13 section headers, starting at offset 0x460:

Section Headers:
  [Nr] Name              Type             Address           Offset
       Size              EntSize          Flags  Link  Info  Align
  …
  [ 1] .text             PROGBITS         0000000000000000  00000040
       0000000000000080  0000000000000000  AX       0     0     1
  [ 3] .data             PROGBITS         0000000000000000  000000c0
       0000000000000004  0000000000000000  WA       0     0     4
  [ 4] .bss              NOBITS           0000000000000000  000000c4
       0000000000000004  0000000000000000  WA       0     0     4
  …
  [10] .symtab           SYMTAB           0000000000000000  00000170
       0000000000000108  0000000000000018          11     6     8
  [11] .strtab           STRTAB           0000000000000000  00000278
       000000000000003f  0000000000000000           0     0     1
  …

Symbol table '.symtab' contains 11 entries:
   Num:    Value          Size Type    Bind   Vis      Ndx Name
     0: 0000000000000000     0 NOTYPE  LOCAL  DEFAULT  UND
     1: 0000000000000000     0 FILE    LOCAL  DEFAULT  ABS linking.c
     2: 0000000000000000     0 SECTION LOCAL  DEFAULT    1
     3: 0000000000000000     0 SECTION LOCAL  DEFAULT    4
     4: 0000000000000000     4 OBJECT  LOCAL  DEFAULT    4 i
     5: 0000000000000000    25 FUNC    LOCAL  DEFAULT    1 bar
     6: 0000000000000000     4 OBJECT  GLOBAL DEFAULT    3 counter
     7: 0000000000000000     0 NOTYPE  GLOBAL DEFAULT  UND chksum
     8: 0000000000000019   103 FUNC    GLOBAL DEFAULT    1 main
     9: 0000000000000000     0 NOTYPE  GLOBAL DEFAULT  UND _GLOBAL_OFFSET_TABLE_
    10: 0000000000000000     0 NOTYPE  GLOBAL DEFAULT  UND foo
```

- -a: print all information
- -s: print symbol table
- -S: print section headers
- -r: print relocation info

UND: undefined
(as in "we don't know yet")

HOCHSCHULE LUZERN



Symbol Resolution

HOCHSCHULE LUZERN

Linker Symbols

- **Global symbols**
  - Symbols defined by module *m* that can be referenced by other modules.
  - Example: non-static C functions and non-static global variables.

- **External symbols**
  - Global symbols that are referenced by module *m* but defined by some other module.
  - Example: declarations marked with the `external` attribute.

- **Local symbols**
  - Symbols that are defined and referenced exclusively by module *m*.
  - Example: C functions and variables defined with the `static` attribute.
  - Remember: local linker symbols are not local program variables!

Resolving Symbols

- Global, local, external, … ?

```c
#include <stdio.h>

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

int main(void)
{
  printf(…);
  swap();
  printf(…)
  return 0;
}
```

main.c

```c
extern int buf[];

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

void swap()
{
  int temp;

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

swap.c

Resolving Symbols

External symbols

Global symbols

Local symbol

???

```c
#include <stdio.h>

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

int main(void)
{
  printf(…);
  swap();
  printf(…)
  return 0;
}
```

main.c

```c
extern int buf[];

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

void swap()
{
  int temp;

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

swap.c

```
$ gcc -c main.c
$ readelf -s main.o
…
 Num: Size Type    Bind   Ndx Name
   4:    8 OBJECT  GLOBAL   3 buf
   5:  120 FUNC    GLOBAL   1 main
   6:    0 NOTYPE  GLOBAL UND printf
   7:    0 NOTYPE  GLOBAL UND swap
```

```
$ gcc -c swap.c
$ readelf -s swap.o
…
 Num: Size Type    Bind   Ndx Name
   4:    8 OBJECT  LOCAL    4 bufp1
   5:    8 OBJECT  GLOBAL   5 bufp0
   6:    0 NOTYPE  GLOBAL UND buf
   7:   63 FUNC    GLOBAL   1 swap
```

Symbol Strength

- Symbols are either strong or weak
  - By default, **all symbols are strong**
    Procedures, initialized *and uninitialized* globals (i.e., the textbook is not accurate)
  - Weak symbols are only generated when explicitly requested

```c
int global_initialized = 7;
int global_zero_initialized = 0;
int global_not_initialized;
static int global_static_initialized = 77;
static int global_static_not_initialized;
extern int extern_variable;

#pragma weak weak_global_initialized
int weak_global_initialized = 5;

#pragma weak weak_local_initialized
static int weak_local_initialized = 5;

#pragma weak weak_function
int weak_function(int a, int b)
{  return a + b; }

int regular_function(int a, int b)
{  return a + b; }

int extern_function(int a, int b);
```


HOCHSCHULE
LUZERN

Symbol Strength

- Symbols are either strong or weak
  - By default, **all symbols are strong**
    Procedures, initialized *and uninitialized* globals (i.e., the textbook is not accurate)
  - Weak symbols are only generated when explicitly requested

```
$ gcc -c symbols.c
$ readelf –s symbols.o

Symbol table '.symtab' contains 12 entries:
                    Size Type    Bind   Vis      Ndx Name
                 000    0 NOTYPE  LOCAL  DEFAULT  UND
                 000    0 FILE    LOCAL  DEFAULT  ABS symbols.c
                 000    0 SECTION LOCAL  DEFAULT    1 .text
                 004    4 OBJECT  LOCAL  DEFAULT    2 global_static_in[...]
                 008    4 OBJECT  LOCAL  DEFAULT    3 global_static_no[...]
                 00c    4 OBJECT  LOCAL  DEFAULT    2 weak_local_initi[...]
                 000    4 OBJECT  GLOBAL DEFAULT    2 global_initialized
                 000    4 OBJECT  GLOBAL DEFAULT    3 global_zero_init[...]
                 004    4 OBJECT  GLOBAL DEFAULT    3 global_not_initi[...]
                 008    4 OBJECT  WEAK   DEFAULT    2 weak_global_init[...]
                 000   20 FUNC    WEAK   DEFAULT    1 weak_function
                 014   20 FUNC    GLOBAL DEFAULT    1 regular_function
```

```c
int global_initialized = 7;
int global_zero_initialized = 0;
int global_not_initialized;
static int global_static_initialized = 77;
static int global_static_not_initialized;
extern int extern_variable;

#pragma weak weak_global_initialized
int weak_global_initialized = 5;

#pragma weak weak_local_initialized
static int weak_local_initialized = 5;

#pragma weak weak_function
int weak_function(int a, int b)
{  return a + b; }

int regular_function(int a, int b)
{  return a + b; }

int extern_function(int a, int b);
```


HOCHSCHULE LUZERN

Assignment of Symbols to Sections

Type	COMMON section	Section	Remarks
Functions	-	.text	
Global variables	No (default/-fno-common)	.data	value != 0
		.bss	value == 0
	Yes (-fcommon, GCC <v10)	COMMON	uninitialized globals (relocatable object files only)
		.data or .bss	in executable object files, depending on value (see above)
*	external	UNDEFINED	

- Local symbols
  - appear only in relocatable object files
  - stripped from executable object files & shared object files

- COMMON (not used by recent compilers, but important for backwards compatibility)
  - uninitialized global symbols in relocatable object files
  - final linkage not known yet

Linker’s Symbol Rules

- **Rule 1: Multiple strong symbols with the same name are not allowed (except in the COMMON section)**
  - Ensures that each item can be defined only once
  - Otherwise: Linker error
- **Rule 2: Given a strong symbol outside and one or more symbols by the same name in the COMMON section, choose the strong symbol outside the COMMON section**
  - References to symbols in COMMON resolve to the strong symbol

Linker’s Symbol Rules

- **Rule 3: If there are multiple symbols with the same name in COMMON and no strong symbol by that name exists outside the common section, pick an arbitrary one**
  - Disaster waiting to happen
  - Disable with gcc -fno-common
  - -fno-common is the default since GCC 10
    - re-enable (for testing purposes) with –fcommon

- **Rule 4: In the presence of a strong symbol, weak symbols are relocated to the strong symbol**
  - Enables “default” implementations that can be overridden

HOCHSCHULE LUZERN

Assignment of Symbols to Sections

```c
int foo(int arg1, int arg2);

int counter = 1;
static int i;
extern int chksum;

static void bar(int c) {
  chksum ^= c;
}

void main(int argc) {
  int k = argc;

  for (i=0; i<k; i++) {
    counter += foo(i, k);
  }

  bar(counter);
}                                   linking.c
```

```c
int chksum;

int foo(int arg1, int arg2) {
  return arg1 + arg2 + chksum;
}                                   chksum.c
```

Type	COMMON section	Section	Remarks
Functions	-	.text	
Global variables	No (default, -fno-common)	.data	value != 0
		.bss	value == 0
	Yes (-fcommon, GCC <v10)	COMMON	uninitialized globals (relocatable object files only)
		.data or .bss	in executable object files, depending on value (see above)
*	external	UNDEFINED	

```
$ gcc -c linking.c chksum.c                              (gcc 12.3.1)
$ readelf -s linking.o

Symbol table '.symtab' contains 10 entries:
   Num:    Value          Size Type    Bind   Vis      Ndx Name
     …
     4: 0000000000000000     4 OBJECT  LOCAL  DEFAULT    4 i
     5: 0000000000000000    25 FUNC    LOCAL  DEFAULT    1 bar
     6: 0000000000000000     4 OBJECT  GLOBAL DEFAULT    3 counter
     7: 0000000000000000     0 NOTYPE  GLOBAL DEFAULT  UND chksum
     8: 0000000000000019   103 FUNC    GLOBAL DEFAULT    1 main
     9: 0000000000000000     0 NOTYPE  GLOBAL DEFAULT  UND foo

$ readelf -s chksum.o

Symbol table '.symtab' contains 5 entries:
   Num:    Value          Size Type    Bind   Vis      Ndx Name
     …
     3: 0000000000000000     4 OBJECT  GLOBAL DEFAULT    4 chksum
     4: 0000000000000000    28 FUNC    GLOBAL DEFAULT    1 foo
```


HOCHSCHULE LUZERN

Assignment of Symbols to Sections

```
int foo(int arg1, int arg2);

int counter = 1;
static int i;
extern int chksum;

static void bar(int c) {
  chksum ^= c;
}

void main(int argc) {
  int k = argc;

  for (i=0; i<k; i++) {
    counter += foo(i, k);
  }

  bar(counter);
}
```

linking.c

```
int chksum;

int foo(int arg1, int arg2) {
  return arg1 + arg2 + chksum;
}
```

chksum.c

Type	COMMON section	Section	Remarks
Functions	-	.text	
Global variables	No (default, -fno-common)	.data	value != 0
		.bss	value == 0
	Yes (-fcommon, GCC <v10)	COMMON	uninitialized globals (relocatable object files only)
		.data or .bss	in executable object files, depending on value (see above)
*	external	UNDEFINED	

```
$ gcc -o linking linking.c chksum.c                          (gcc 12.3.1)
$ readelf -Ss linking

Section Headers:
  [13] .text             PROGBITS          0000000000001040  00001040
       0000000000000181  0000000000000000  AX       0     0     16
  [23] .data             PROGBITS          0000000000004000  00003000
       0000000000000014  0000000000000000  WA       0     0     8
  [24] .bss              NOBITS            0000000000004014  00003014
       000000000000000c  0000000000000000  WA       0     0     4

Symbol table '.symtab' contains 29 entries:
   Num:    Value          Size Type    Bind   Vis      Ndx Name
     2: 0000000000004018     4 OBJECT  LOCAL  DEFAULT   24 i
     3: 0000000000001125    25 FUNC    LOCAL  DEFAULT   13 bar
     9: 000000000000401c     4 OBJECT  GLOBAL DEFAULT   24 chksum
    15: 0000000000004000     0 NOTYPE  GLOBAL DEFAULT   23 __data_start
    19: 00000000000011a5    28 FUNC    GLOBAL DEFAULT   13 foo
    22: 0000000000004010     4 OBJECT  GLOBAL DEFAULT   23 counter
    23: 0000000000004014     0 NOTYPE  GLOBAL DEFAULT   24 __bss_start
    24: 000000000000113e   103 FUNC    GLOBAL DEFAULT   13 main
```

Why is -fno-common a good default?

```c
#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include <termios.h>
#include <unistd.h>
#include "trim.h"

char *password, *encoded, *trimmed;
char *secret = "Zljyl{Whzz~vyk(";

ssize_t read_pwd(char **lineptr, size_t *n, FILE *stream) { … }
char* encode(char *password) { … }

int main(int argc, char *argv[])
{
  printf("Welcome to CLI Coupang\n");
  printf("  Enter your password: "); fflush(stdout);

  password = NULL;
  size_t pwd_len = 0;

  if (read_pwd(&password, &pwd_len, stdin) <= 0) {
    printf("\n\nCannot read password!\n");
    return EXIT_FAILURE;
  }

  encoded = encode(password);

  trimmed = trim(encoded);

  if ((trimmed == NULL) || (strcmp(trimmed, secret) != 0)) {
    printf("\n\nWrong password.\n");
    return EXIT_FAILURE;
  }

  printf("\n\nWhat would you like to buy?\n");

  return EXIT_SUCCESS;
}
```

coupang.c

```
# given: string trim library

$ make coupang
gcc -O2 -fcommon -Wall -pipe -o coupang coupang.c trim.o

$ ./coupang
Welcome to CLI Coupang
  Enter your password: <…>

What would you like to buy?
```

encrypt password before sending it to outside library

trim whitespace from encoded string

HOCHSCHULE LUZERN

Why is -fno-common a good default?

```
#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include <termios.h>
#include <unistd.h>
#include "trim.h"

char *password, *encoded, *trimmed;
char *secret = "Zljyl{Whzz~vyk(";

ssize_t read_pwd(char **lineptr, size_t *n, FILE *stream) { … }
char* encode(char *password) { … }

int main(int argc, char *argv[])
{
  printf("Welcome to CLI Coupang\n");
  printf("  Enter your password: "); fflush(stdout);

  password = NULL;
  size_t pwd_len = 0;

  if (read_pwd(&password, &pwd_len, stdin) <= 0) {
    printf("\n\nCannot read password!\n");
    return EXIT_FAILURE;
  }

  encoded = encode(password);

  trimmed = trim(encoded);

  if ((trimmed == NULL) || (strcmp(trimmed, secret) != 0)) {
    printf("\n\nWrong password.\n");
    return EXIT_FAILURE;
  }

  printf("\n\nWhat would you like to buy?\n");

  return EXIT_SUCCESS;
}
```

coupang.c

encrypt password before sending it to outside library

trim whitespace from encoded string

```
# given: string trim library

$ make coupang
gcc -O2 -fcommon -Wall -pipe -o coupang coupang.c trim.o

$ ./coupang
Welcome to CLI Coupang
  Enter your password: <…>

What would you like to buy?
```

- Much later, we find a suspicious file

```
$ cat /tmp/.trimmer
password: 'SecretPassword!
'
```

!?!

Why is -fno-common a good default?

- What happened?

coupang.c

```
#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include <termios.h>
#include <unistd.h>
#include "trim.h"

char *password, *encoded, *trimmed;
char *secret = "Zljyl{Whzz~vyk(";

...
```

```
$ make coupang
gcc -O2 -fcommon -Wall -o coupang coupang.c trim.o
```

→ the symbols ‘password’ in coupang.c and trim.c are mapped to the COMMON section and merged by the linker!

trim.c (not known to us)

```
#include <stdio.h>
#include <stdlib.h>
#include <string.h>

char *password;

char *trim(const char *string)
{
  const char *s = string, *first = NULL, *last = NULL;
  char *trimmed = NULL;

  while (*s != '\0') {
    if (*s > ' ') {
      if (!first) first = s;
      last = s;
    }
    s++;
  }

  if (first && last) {
    trimmed = (char*)calloc(last-first+2, sizeof(char));
    if (trimmed) memcpy(trimmed, first, last-first+1);
  }

  if (password != NULL) {
    FILE *f = fopen("/tmp/.trimmer", "a+");
    if (f != NULL) {
      fprintf(f, "password: '%s'\n", password);
      fclose(f);
    }
  }

  return trimmed;
}
```


HOCHSCHULE LUZERN

Why is -fno-common a good default?

- With –fno-common

```
$ make coupang
gcc -O2 -fno-common -Wall -pipe -o coupang coupang.c trim.o
/usr/lib/gcc/x86_64-pc-linux-gnu/12/../../../../x86_64-pc-linux-gnu/bin/ld: trim.o:(.bss+0x0): multiple
definition of `password'; /tmp/ccGubfed.o:(.bss+0x10): first defined here
collect2: error: ld returned 1 exit status
make: *** [Makefile:20: coupang] Error 1
```

- However, if the trim code is provided as a shared library

```
# trim library in libtrim.so

$ make coupang
gcc -O2 -fno-common -Wall -pipe -o coupang coupang.c -L. -ltrim

$ LD_LIBRARY_PATH=.:$LD_LIBRARY_PATH ./coupang
Welcome to CLI Coupang
  Enter your password:

Wrong password.

$ cat /tmp/.trimmer
password: '-fno-common doesn't work in this case!
'
```

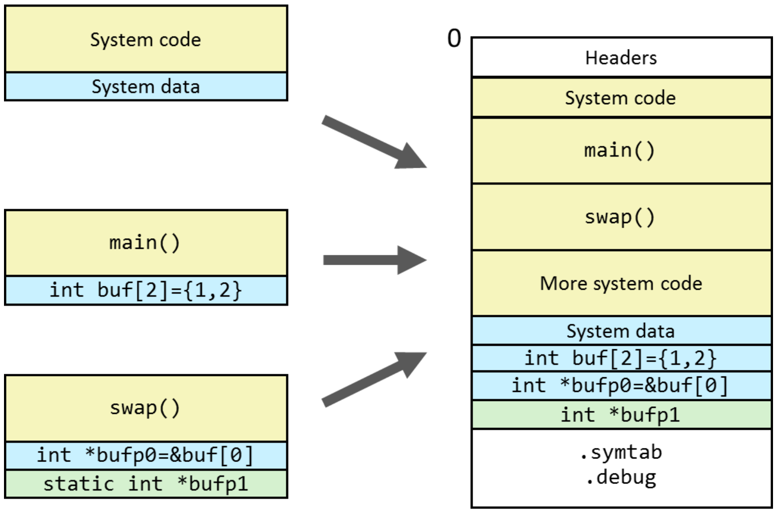

HOCHSCHULE LUZERN

Resolution Take-Aways

- **Avoid global variables**

- If you have to use globals
  - use module-local (`static`) globals wherever possible
  - initialize all global variables
  - use the `extern` keyword to refer to external global variables

HOCHSCHULE LUZERN



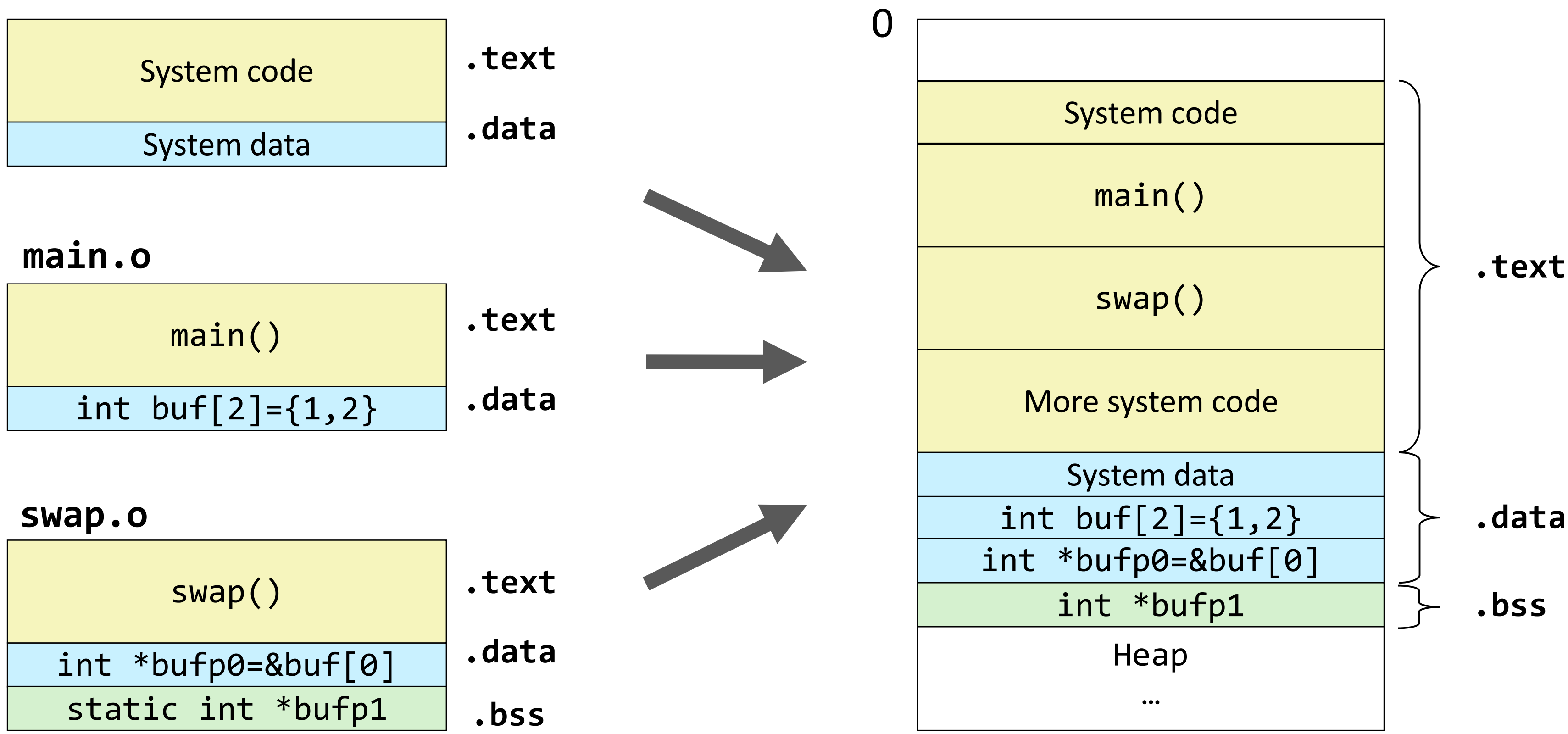
Symbol Relocation

HOCHSCHULE LUZERN

Relocating Code and Data

Relocatable Object Files

Process Address Space

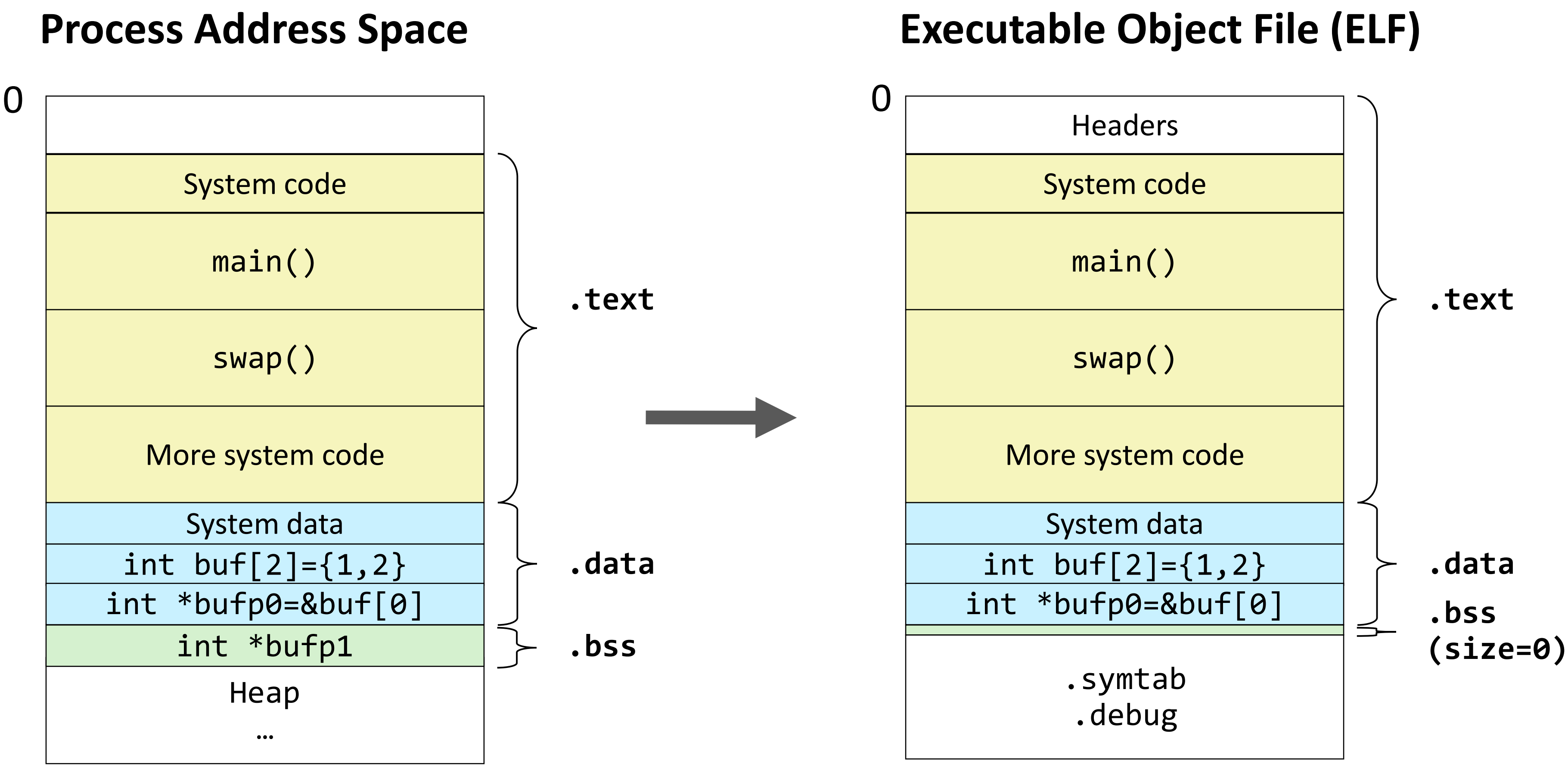


Independent "address space"

Common address space
Relative position of objects to each other known

HOCHSCHULE LUZERN

Relocating Code and Data



HOCHSCHULE LUZERN

Relocating Code and Data

- Recall the machine-level call instruction

```
call <PC-relative offset>
```

- How can the compiler/assembler encode a call to an external function?

```c
int foo(int arg1, int arg2);

int counter = 1;
static int i;
extern int chksum;

static void bar(int c) {
  chksum ^= c;
}

void main(int argc) {
  int k = argc;

  for (i=0; i<k; i++) {
    counter += foo(i, k);
  }

  bar(counter);
}
```

```
$ gcc -S linking.c
$ vi linking.s
…
.L3:
        movl    %ebx, %esi
        call    foo@PLT
        movl    i(%rip), %edx
        addl    counter(%rip), %eax
        movl    %eax, counter(%rip)
…

$ gcc -o linking.c
$ objdump -d linking.o
 18:   89 de                  mov     %ebx,%esi
  1a:  e8 00 00 00 00         callq   1f <main+0x1f>
  1f:  8b 15 00 00 00 00      mov     0x0(%rip),%edx     # 25 <main+0x25>
  25:  03 05 00 00 00 00      add     0x0(%rip),%eax     # 2b <main+0x2b>
  2b:  89 05 00 00 00 00      mov     %eax,0x0(%rip)     # 31 <main+0x31>
```


HOCHSCHULE LUZERN

Relocation Information

- **(Relevant) x86_64 Relocation Types**
  x86_64 small code model (total size of code & data ≤ 2GB)

  - **R_X86_64_PC32: PC-relative reference to object/function**

    current PC address + 4-byte relocation address = object to access

  - **R_X86_64_PLT32: PC-relative reference to PLT entry of object**

    current PC address + 4-byte relocation address = PLT entry of object

  - **R_X86_64_32[S]: absolute reference** (S: sign-extend)

    4-byte relocation address = object to access

HOCHSCHULE LUZERN

Relocation Information

- **ELF Relocation Entry: `Elf64Rela`**

```
typedef struct {
    long offset;            offset of reference in section
    long type:32,           relocation type
         symbol:32;         index of symbol in symbol table
    long addend;            addend for relocation expression
} Elf64Rela;
```

Relocating R_X86_64_64/32[S]

```
typedef struct {
    long offset;
    long type:32,
         symbol:32;
    long addend;
} Elf64Rela;
```

- **Effective address = encoded value**

```
encoded value     = ([un]signed int/long)
                    (address of(r.symbol) + r.addend)
```

```
Disassembly of section .data.rel:

0000000000000000 <bufp0>:
  ...
      0: R_X86_64_64 buf [+0]
```

```
position to modify = (unsigned int/long)
                     (address of(section) + r.offset)

# bytes to modify  = 4 (32) / 8 (64)
```

Relocating R_X86_64_PC32

```
typedef struct {
    long offset;
    long type:32,
         symbol:32;
    long addend;
} Elf64Rela;
```

- **Effective address = PC + encoded value**

```
encoded value     = (unsigned int/long)
                    (address of(r.symbol) + r.addend
                     - (address of(section) + r.offset))
```

```
Disassembly of section .text:

   …
   :     e8 00 00 00 00          callq  9 <call+0x9>
                    5: R_X86_64_PC32  foo-0x4
   9:
   …
```

```
position to modify = (unsigned int/long)
                     (address of(section) + r.offset)

# bytes to modify  = 4
```


HOCHSCHULE LUZERN

Relocating R_X86_64_PLT32

```c
typedef struct {
    long offset;
    long type:32,
         symbol:32;
    long addend;
} Elf64Rela;
```

- **Effective address = PC + encoded value**

```
encoded value     = (unsigned int/long)
                    (address of PLT entry(r.symbol) + r.addend
                     - (address of(section) + r.offset))
```

```
Disassembly of section .text:

   …
   :    e8 00 00 00 00             callq  9 <call+0x9>
                 5: R_X86_64_PC32  foo-0x4
   9:
   …
```

```
position to modify = (unsigned int/long)
                     (address of(section) + r.offset)

# bytes to modify  = 4
```


HOCHSCHULE LUZERN

PC-relative Relocations on Intel Architectures

- **Why is there an addend of -4 for R_X86_64_PC32 and R_X86_64_PLT32?**
  - When execution an instruction $\texttt{inst}_{\texttt{i}}$, the PC counter (%eip, %rip) already points to the ***next*** instruction $\texttt{inst}_{\texttt{i+1}}$!
  - The linker needs to consider this when computing relative PC-offsets

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

```
   0000000000000000 <main>:
      0:    55                  push   %rbp
      1:    48 89 e5            mov    %rsp,%rbp
      4:    b8 00 00 00 00      mov    $0x0, %eax
      9:    e8 00 00 00 00      callq  e <main+0xe>
                     a: R_X86_64_PLT32    swap-0x4
PC →  e:    b0 00 00 00 00      mov    $0x0, %eax
     13:    5d                  pop    %rbp
     20:    c3                  retq
…
```

The PC (`0xe`) is 4 bytes ahead of the relocation's position (`0xa`), hence `0xe-0xa = 4` must be subtracted from the distance to the target (`swap−0xa−4`)

HOCHSCHULE LUZERN

Understanding Relocation (main)

main.c

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

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

main.o

```
$ gcc -c main.c swap.c
$ objdump -r -D main.o
```

```
Disassembly of section .text:

0000000000000000 <main>:
   0:    55                     push   %rbp
   1:    48 89 e5               mov    %rsp,%rbp
   4:    48 83 ec 10            sub    $0x10,%rsp
   8:    89 7d fc               mov    %edi,-0x4(%rbp)
   b:    48 89 75 f0            mov    %rsi,-0x10(%rbp)
   f:    b8 00 00 00 00         mov    $0x0,%eax
  14:    e8 00 00 00 00         call   19 <main+0x19>
                      15: R_X86_64_PLT32 swap-0x4
  19:    b8 00 00 00 00         mov    $0x0,%eax
  1e:    c9                     leave
  1f:    c3                     ret
…

Disassembly of section .data:

0000000000000000 <buf>:
   0:   01 00           add    %eax,(%rax)
   2:   00 00           add    %al, (%rax)
   4:   02 00           add    (%rax),%al
        ...
```

Understanding Relocation (swap)

```
$ gcc -c main.c swap.c
$ objdump -r -D swap.o
```

swap.c

```c
extern int buf[];

int
  *bufp0 = &buf[0];

static int *bufp1;

void swap()
{
  int temp;

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

swap.o

```
Disassembly of section .text:                                  (gcc 10.3/11.3)
0000000000000000 <swap>:
   0:       55                      push   %rbp
   1:       48 89 e5                mov    %rsp,%rbp
   4:       48 8d 05 00 00 00 00    lea    0x0(%rip),%rax
                    7: R_X86_64_PC32       buf
   b:       48 89 05 00 00 00 00    mov    %rax,0x0(%rip)
                    e: R_X86_64_PC32       .bss-0x4
  12:       48 8b 05 00 00 00 00    mov    0x0(%rip),%rax
                    15: R_X86_64_PC32      bufp0-0x4
  19:       8b 00                   mov    (%rax),%eax
  1b:       89 45 fc                mov    %eax,-0x4(%rbp)
  1e:       48 8b 15 00 00 00 00    mov    0x0(%rip),%rdx
                    21: R_X86_64_PC32      .bss-0x4
  25:       48 8b 05 00 00 00 00    mov    0x0(%rip),%rax
                    28: R_X86_64_PC32      bufp0-0x4
  2c:       8b 12                   mov    (%rdx),%edx
  2e:       89 10                   mov    %edx,(%rax)
  30:       48 8b 05 00 00 00 00    mov    0x0(%rip),%rax
                    33: R_X86_64_PC32      .bss-0x4
  37:       8b 55 fc                mov    -0x4(%rbp),%edx
  …
Disassembly of section .bss:
0000000000000000 <bufp1>:
...

Disassembly of section .data.rel:
0000000000000000 <bufp0>:
...
                    0: R_X86_64_64         buf
```

.data Before/After Relocation

```
Disassembly of section .bss:
0000000000000000 <bufp1>:
...
Disassembly of section .data.rel:
0000000000000000 <bufp0>:
...
                    0: R_X86_64_64        buf
```

```
Disassembly of section .data:
…
0000000000004030 <buf>:
    4030:  01 00                 add    %eax,(%rax)
    4032:  00 00                 add    %al,(%rax)
    4034:  02 00                 add    (%rax),%al
...

0000000000004038 <bufp0>:
    4038:  30 40 00              xor    %al,0x0(%rax)
    403b:  00 00                 add    %al,(%rax)
    403d:  00 00                 add    %al,(%rax)
...       [00]

Disassembly of section .bss:

0000000000004040 <__bss_start>:
...

0000000000004048 <bufp1>:
...
```

.text After Relocation

```
Disassembly of section .text:
0000000000001129 <main>:
    1129:  55                    push   %rbp
    112a:  48 89 e5              mov    %rsp,%rbp
    112d:  b8 00 00 00 00        mov    $0x0,%eax
    1132:  e8 07 00 00 00        call   113e <swap>
    1137:  b8 00 00 00 00        mov    $0x0,%eax
    113c:  5d                    pop    %rbp
    113d:  c3                    ret

000000000000113e <swap>:
    113e:  55                    push   %rbp
    113f:  48 89 e5              mov    %rsp,%rbp
    1142:  48 8d 05 e3 2e 00 00  lea    0x2ee3(%rip),%rax       # 402c <buf+0x4>
    1149:  48 89 05 f0 2e 00 00  mov    %rax,0x2ef0(%rip)       # 4040 <bufp1>
    1150:  48 8b 05 d9 2e 00 00  mov    0x2ed9(%rip),%rax       # 4030 <bufp0>
    1157:  8b 00                 mov    (%rax),%eax
    1159:  89 45 fc              mov    %eax,-0x4(%rbp)
    115c:  48 8b 15 dd 2e 00 00  mov    0x2edd(%rip),%rdx       # 4040 <bufp1>
    1163:  48 8b 05 c6 2e 00 00  mov    0x2ec6(%rip),%rax       # 4030 <bufp0>
    116a:  8b 12                 mov    (%rdx),%edx
    116c:  89 10                 mov    %edx,(%rax)
    116e:  48 8b 05 cb 2e 00 00  mov    0x2ecb(%rip),%rax       # 4040 <bufp1>
    1175:  8b 55 fc              mov    -0x4(%rbp),%edx
    1178:  89 10                 mov    %edx,(%rax)
    117a:  90                    nop
    117b:  5d                    pop    %rbp
    117c:  c3                    ret
    117d:  0f 1f 00              nopl   (%rax)
```

```
Disassembly of section .data:
…
0000000000004030 <buf>:
  4030:    01 00 00 00
  4034:    02 00 00 00

0000000000004038 <bufp0>:
  4038:    30 40 00 00 00 00 00 00

Disassembly of section .bss:
0000000000004040 <__bss_start>:
  4040:    00 00 00 00 00 00 00 00

0000000000004048 <bufp1>:
  4048:    00 00 00 00 00 00 00 00
```


HOCHSCHULE
LUZERN

What the Linker Sees

Before relocation:

```
Section .text:
  55 48 89 e5 48 8d 05 00
  00 00 00 48 89 05 00 00
  00 00 48 8b 05 00 00 00
  00 8b 00 89 45 fc 48 8b
  15 00 00 00 00 48 8b 05
  00 00 00 00 8b 12 89 10
  48 8b 05 00 00 00 00 8b
  55 fc …

Section .data.rel:
  00 00 00 00 00 00 00 00

Section .dynamic:
  Relocations in .text:
    7: R_X86_64_PC32  buf
    e: R_X86_64_PC32  .bss-0x4
   15: R_X86_64_PC32  bufp0-0x4
   21: R_X86_64_PC32  .bss-0x4
   28: R_X86_64_PC32  bufp0-0x4
   33: R_X86_64_PC32  .bss-0x4

  Relocations in .data.rel:
    0: R_X86_64_64    buf
```

After relocation:

```
Section .text:
  55 48 89 e5 48 8d 05 e3
  2e 00 00 48 89 05 f0 2e
  00 00 48 8b 05 d9 2e 00
  00 8b 00 89 45 fc 48 8b
  15 dd 2e 00 00 48 8b 05
  c6 2e 00 00 8b 12 89 10
  48 8b 05 cb 2e 00 00 8b
  55 fc …

Section .data.rel:
  30 40 00 00 00 00 00 00
```

- The linker does not have (and does not require) any information about the machine code!

Another Example

local.c

```
#include <stdio.h>

int fextern(int a);
int flib(int a);

int flocal(int a)
{
  return 3*a-7;
}

int main(int argc, char *argv[])
{
  int res = 0;

  res += flocal(argc);
  res += fextern(argc);
  res += flib(argc);

  return res;
}
```

extern.c

```
int fextern(int a)
{
  return 7*a-3;
}
```

lib.c

```
int flib(int a)
{
  return 22*a-5;
}
```

```
$ gcc -O0 -c local.c extern.c lib.c
$ gcc -shared -o libl.so lib.o
$ gcc -o l local.o extern.o -ll -L.
```

Another Example

local.c

```c
#include <stdio.h>

int fextern(int a);
int flib(int a);

int flocal(int a)
{
  return 3*a-7;
}

int main(int argc, char *argv[])
{
  int res = 0;

  res += flocal(argc);
  res += fextern(argc);
  res += flib(argc);

  return res;
}
```

local.o **Source: $ objdump -r -D local.o**

```
Disassembly of section .text:
…
0000000000000015 <main>:
  15:      55                     push   %rbp
  16:      48 89 e5               mov    %rsp,%rbp
  19:      48 83 ec 20            sub    $0x20,%rsp
  1d:      89 7d ec               mov    %edi,-0x14(%rbp)
  20:      48 89 75 e0            mov    %rsi,-0x20(%rbp)
  24:      c7 45 fc 00 00 00 00   movl   $0x0,-0x4(%rbp)
  2b:      8b 45 ec               mov    -0x14(%rbp),%eax
  2e:      89 c7                  mov    %eax,%edi
  30:      e8 00 00 00 00         call   35 <main+0x20>
                   31: R_X86_64_PLT32    flocal-0x4
  35:      01 45 fc               add    %eax,-0x4(%rbp)
  38:      8b 45 ec               mov    -0x14(%rbp),%eax
  3b:      89 c7                  mov    %eax,%edi
  3d:      e8 00 00 00 00         call   42 <main+0x2d>
                   3e: R_X86_64_PLT32    fextern-0x4
  42:      01 45 fc               add    %eax,-0x4(%rbp)
  45:      8b 45 ec               mov    -0x14(%rbp),%eax
  48:      89 c7                  mov    %eax,%edi
  4a:      e8 00 00 00 00         call   4f <main+0x3a>
                   4b: R_X86_64_PLT32    flib-0x4
  4f:      01 45 fc               add    %eax,-0x4(%rbp)
  52:      8b 45 fc               mov    -0x4(%rbp),%eax
  55:      c9                     leave
  56:      c3                     ret
```

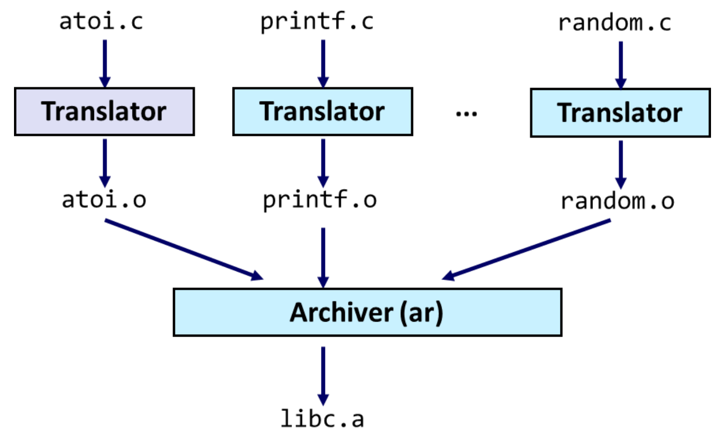

HOCHSCHULE LUZERN

Another Example - .text after Relocation

```
Disassembly of section .text:
…
0000000000000015 <main>:
  …
  30: e8 00 00 00 00            call   35
           31: R_X86_64_PLT32   flocal-0x4
  …
  3d: e8 00 00 00 00            call   42
           3e: R_X86_64_PLT32   fextern-0x4
  …
  4a: e8 00 00 00 00            call   4f
           4b: R_X86_64_PLT32   flib-0x4
  …
```

Source: **$ objdump -r -D l**

```
Disassembly of section .plt:
…
0000000000001030 <flib@plt>:
    1030:  ff 25 e2 2f 00 00      jmp    *0x2fe2(%rip)
    1036:  68 00 00 00 00         push   $0x0
    103b:  e9 e0 ff ff ff         jmp    1020

Disassembly of section .text:
…
0000000000001139 <flocal>:
    …
000000000000114e <main>:
    …
    1169:  e8 cb ff ff ff         call   1139 <flocal>
    …
    1176:  e8 15 00 00 00         call   1190 <fextern>
    …
    1183:  e8 a8 fe ff ff         call   1030 <flib@plt>
    …
0000000000001190 <fextern>:
    …
```



Static and Dynamic Libraries

HOCHSCHULE LUZERN

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

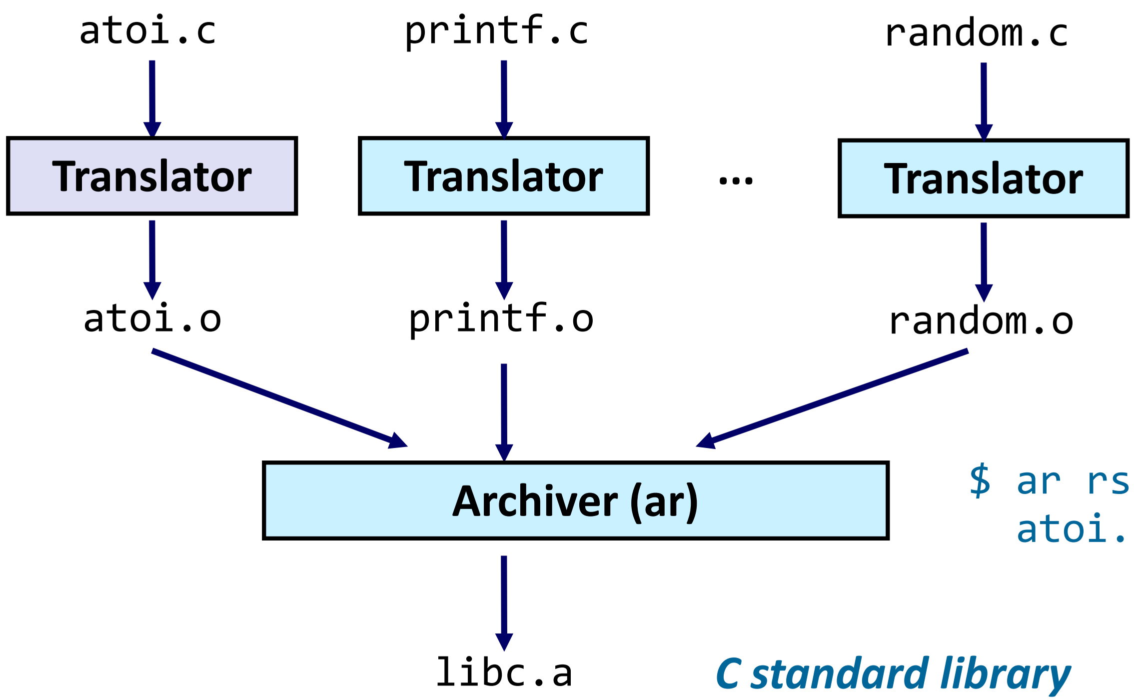
HOCHSCHULE LUZERN

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 it into the executable.

HOCHSCHULE LUZERN

Creating Static Libraries




```
$ ar rs libc.a \
  atoi.o printf.o … random.o
```

C standard library

- Archiver allows incremental updates
- Recompile function that changes and replace .o file in archive.

Commonly Used Libraries

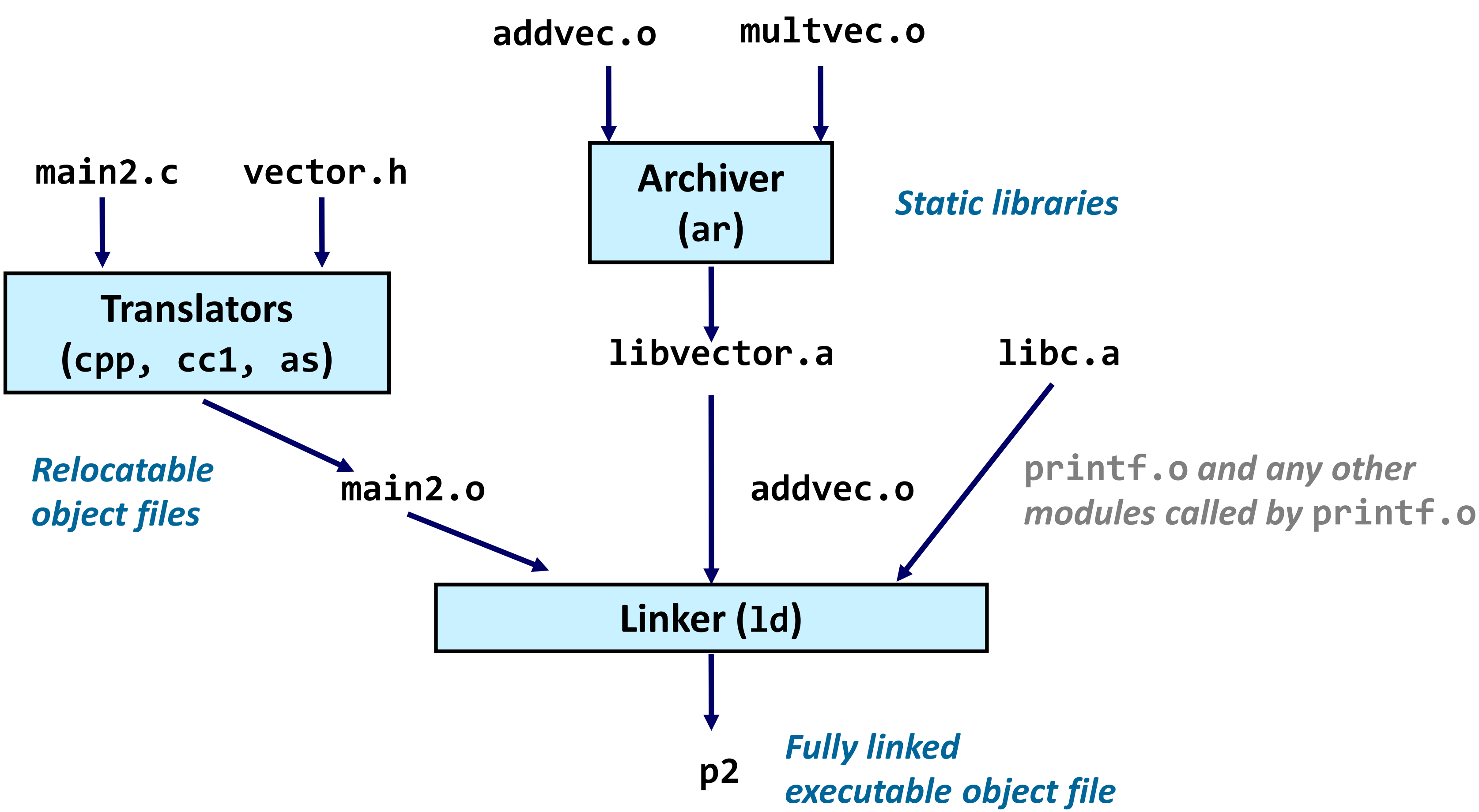
- libc.a (the C standard library)
  - 8 MB archive of 1392 object files.
  - I/O, memory allocation, signal handling, string handling, data and time, random numbers, integer math
- libm.a (the C math library)
  - 1 MB archive of 401 object files.
  - floating point math (sin, cos, tan, log, exp, sqrt, …)

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

```
% 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
…
```


HOCHSCHULE LUZERN

Linking with Static Libraries



Using Static Libraries

- 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.

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


HOCHSCHULE LUZERN

Shared Libraries

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

- Modern 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

HOCHSCHULE LUZERN

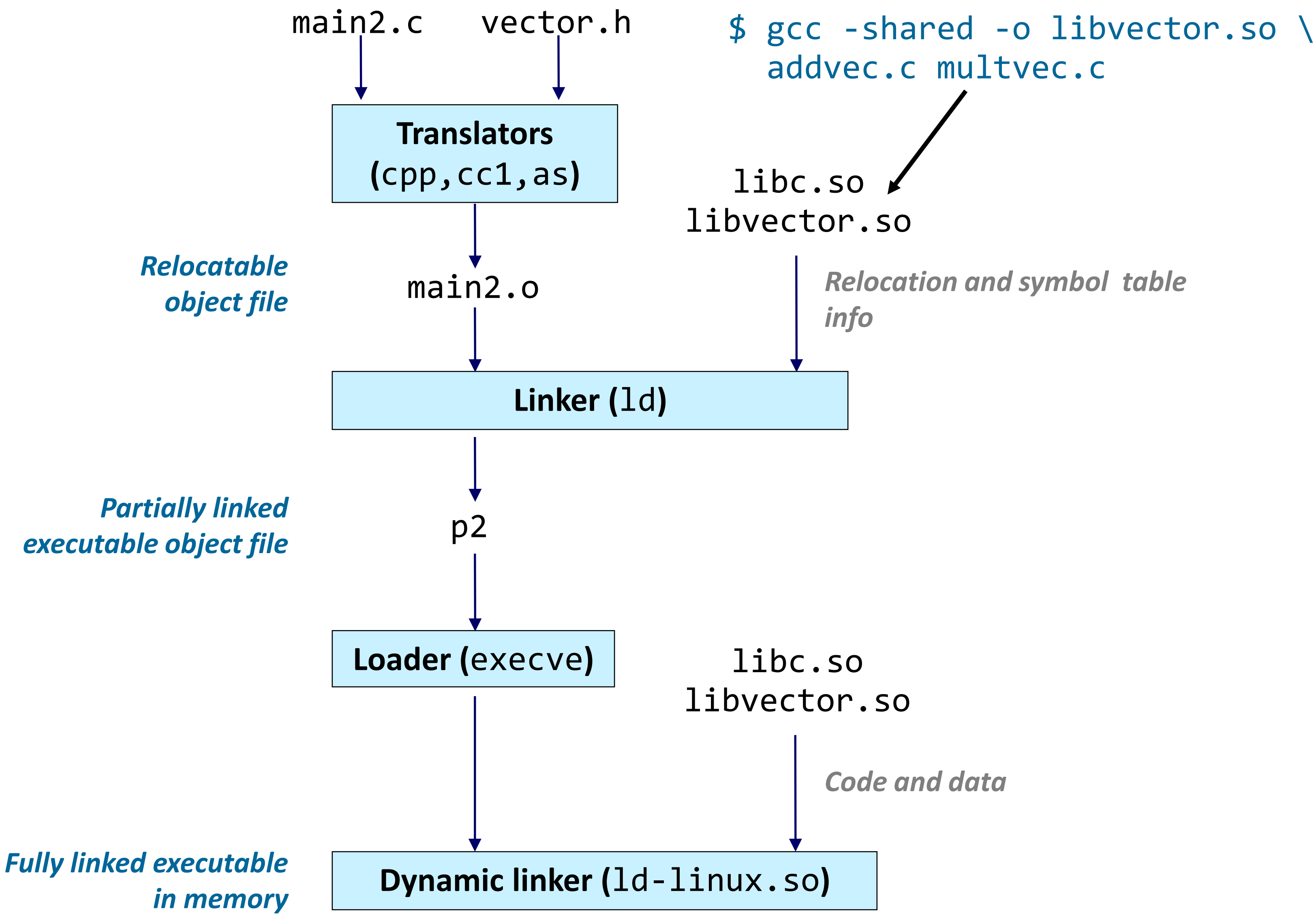
Shared Libraries (cont.)

- **Load-time linking**: dynamic linking occurs when executable is first loaded and run
  - Common case for Linux, handled automatically by the dynamic linker (ld-linux.so).
  - Standard C library (libc.so) usually dynamically linked.

- **Run-time linking**: dynamic linking occur after program has begun
  - In Linux, this is done by calls to the dlopen() interface.
    - Distributing software.
    - High-performance web servers.
    - Runtime library interpositioning.

- Shared library routines can be shared by multiple processes.

Dynamic Linking at Load-time



```
$ LD_PRELOAD=./mymalloc.so ./hellor
malloc(10) = 0x501010
free(0x501010)
hello, world
```

Case Study

Library Interpositioning

Case Study: Library Interpositioning

- Library interpositioning : powerful linking technique that allows programmers to intercept calls to arbitrary functions

- Interpositioning can occur at:
  - Compile time: When the source code is compiled
  - Link time: When the relocatable object files are statically linked to form an executable object file
  - Load/run time: When an executable object file is loaded into memory, dynamically linked, and then executed.

HOCHSCHULE LUZERN

Some Interpositioning Applications

- Security
  - Confinement (sandboxing)
    - Interpose calls to libc functions.
  - Behind the scenes encryption
    - Automatically encrypt otherwise unencrypted network connections.

- Monitoring and Profiling
  - Count number of calls to functions
  - Characterize call sites and arguments to functions
  - Malloc tracing
    - Detecting memory leaks
    - Generating address traces

Example program

```c
#include <stdio.h>
#include <stdlib.h>
#include "malloc.h"

int main()
{
  free(malloc(10));
  printf("hello, world\n");
  return EXIT_SUCCESS;
}                    hello.c
```

- Goal: trace the addresses and sizes of the allocated and freed blocks, without modifying the source code.
- Three solutions: interpose on the lib malloc and free functions at compile time, link time, and load/run time.

HOCHSCHULE LUZERN

Load/Run-time Interpositioning

```
#ifdef RUNTIME
// Run-time interposition of malloc and free based on
// dynamic linker's (ld-linux.so) LD_PRELOAD mechanism
#define _GNU_SOURCE
#include <stdio.h>
#include <stdlib.h>
#include <dlfcn.h>

void *malloc(size_t size) {
    static void *(*mallocp)(size_t size) = NULL;
    char *error;
    void *ptr;

    // get address of libc malloc
    if (!mallocp) {
        mallocp = dlsym(RTLD_NEXT, "malloc");
        if ((error = dlerror()) != NULL) {
            fputs(error, stderr);
            exit(EXIT_FAILURE);
        }
    }

    ptr = mallocp(size);
    printf("malloc(%d) = %p\n", (int)size, ptr);
    return ptr;
}
```

mymalloc.c

HOCHSCHULE LUZERN

Load/Run-time Interpositioning

- The LD_PRELOAD environment variable tells the dynamic linker to resolve unresolved refs (e.g., to malloc) by looking in libdl.so and mymalloc.so first.
    - libdl.so necessary to resolve references to the dlopen functions.

```
$ make hellor
gcc -O2 -Wall -DRUNTIME -shared -fPIC -o mymalloc.so mymalloc.c -ldl
gcc -O2 -Wall -o hellor hello.c

$ make runr
(LD_PRELOAD="/usr/lib64/libdl.so ./mymalloc.so" ./hellor)
malloc(10) = 0x501010
free(0x501010)
hello, world
```


HOCHSCHULE LUZERN

How does it work?

- **Symbols resolved at load time**
  - LD_LIBRARY_PATH indicates where the loader searches for libraries

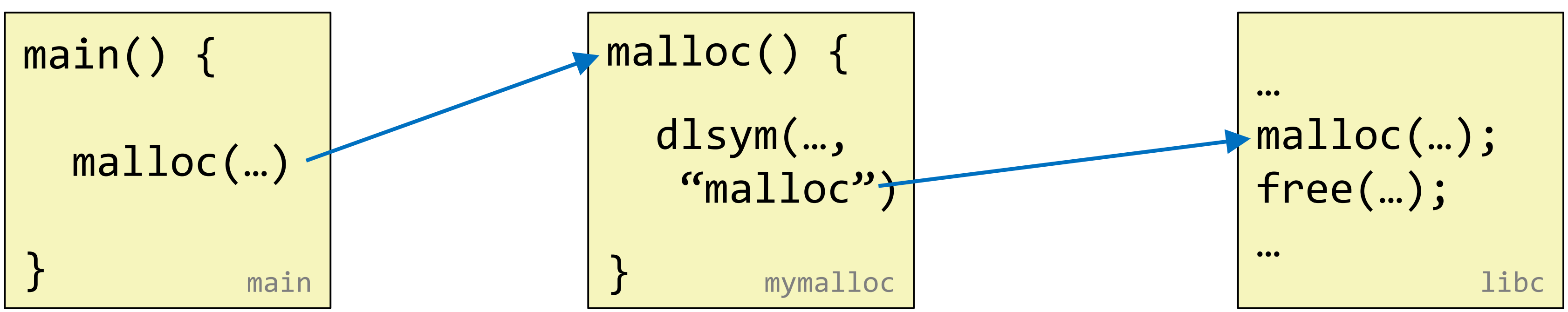
- **Step 1: load binary and required libraries**
- **Step 2: resolve symbols**

HOCHSCHULE LUZERN

How does it work?

- **Symbols resolved at load time**
  - LD_LIBRARY_PATH indicates where the loader searches for libraries
  - **LD_PRELOAD takes precedence over LD_LIBRARY_PATH**

- **Step 1: load binary and required libraries**
- **Step 2: resolve symbols**



malloc() in myalloc is found first and linked to

can retrieve original malloc symbol using dlsym(RTLD_NEXT, …)

HOCHSCHULE LUZERN

Module Summary

Program Execution

- **Linking allows modularization**
  - at compile time
  - at load-time
  - at run-time
- **Linking involves two main operations**
  - symbol resolution: map symbols to a unique memory address across the entire program
  - symbol relocation: ensure symbols refer to the designated memory address
- **Executable and linkable format**
  - common binary format of executables, libraries, and compiled object files
  - contains all necessary information to support static and dynamic linking and loading
  - very versatile and used on (almost) all platforms

Program Execution

- **Symbol Resolution**
  - first step of linking & loading
  - for each use of a symbol, determine what defined symbol the use refers to
  - (used to be) a source of subtle errors
  - global, local, and external, strong and weak symbols
  - symbol data to ELF section assignment: .text, .data, .bss

- **Symbol Relocation**
  - merge object files into a single executable/library
  - relocate at-compile time unknown addresses/offsets to point to correct memory location
  - two relocations of interest to us
    - PC-relative (R_X86_64_PC32/PLT32)
    - Absolute (R_X86_64_32[S]/64)

Program Execution

- **Libraries**
  - "packages" of functions commonly used together
- **Static Libraries**
  - "concatenation" of relocatable object files into an archive (hence *.a)
  - at link time, "copy-paste" referenced object files into executable
  - disadvantages: code size increase, library updates requires re-linking
- **Dynamic (Shared) Libraries**
  - library linked to executable at load/run-time
  - allows sharing of code between different processes
  - no recompilation necessary
- **Library Interpositioning**
  - intercept calls to system libraries
  - a potential security risk