Operating Systems

Tutorial 9

Fabian Klopfer

16. Januar 2021

Intro

- Pingo Polls
- inofficial list of exam-relevant exercises
- Preview sheet 8
- Linking

Inofficial exam-relevant exercises list

Union of tasks & questions considered relevant by Max & Fabian.

Official list/sample exam to follow soon.

Comprehension questions:

- 1.1.1, 1.1.3-4, 2.1.2-4, 2.1.6, 2.4.3-5, 2.4.7, 3.1.1, 3.1.3, 4.1.1-4, 4.1.7, 5.1.1-2, 5.1.4-9, 6.1.2, 6.1.4-8, 7.1.2, 7.1.5, 7.1.7

Exercises:

- 1.4, 2.2, 2.3, 3.3, 3.4, 4.3, 4.4, 4.5, 5.2, 5.3, 5.4, 5.5, 6.2, 6.3, 6.4, 6.5, 7.2, 7.3, 8.2, 8.3, 8.4

Exercise sheet 8

Exercise 1 (comprehension questions) ($8 \cdot 1 = 8$ points)

1. Does active waiting work for cooperative scheduling?
2. Does the solution with active waiting and `turn` variable in Figure 2.23 in Andrew Tanenbaum & Herbert Bos: *Modern Operating Systems*, Pearson, 2015, and slide set 10, also work on computing systems with two CPUs and shared memory?
3. Can the priority inversion problem occur with threads in user space?
4. When computers are developed, they are usually simulated first. The simulation executes instructions one after the other, even if the simulated system has multiple CPUs. Can race conditions occur under these conditions?
5. At an intersection, four cars are each waiting for the car to their right to enter the intersection. Is this a deadlock?
6. The resource trajectories in Figure 6.8 in Andrew Tanenbaum & Herbert Bos: *Modern Operating Systems*, Pearson, 2015, and in slide set 11, are all horizontal or vertical. Under what circumstances can resource traces be diagonal?
7. How can the scheme of resource trajectories (Figure 6.8 and slide set 11) be extended to any number of processes?
8. In a system with two processes and three instances of a resource, can there be a deadlock if each process requires two instances of the resource? Why?

- Cooperative scheduling $\Rightarrow$ process needs to yield! Does active waiting yield?
- Shared memory means both processes can access the variable
- What scheduling type do user level threads use (usually)?
- Can race conditions appear on a single threaded processor? Think about scheduling.
- Who is waiting for whom/what? Draw the diagram.
- Shaded areas mark intersections. What do diagonals stand for in terms of execution?
- Here two processes are the x and y axis respectively
- Can a circular wait occur?

Exercise 2 (deadlock criteria) $(3 + 2 = 5$ points$)$

1. The four criteria for deadlocks according to

   E.G. Coffman, M. Elphick, A. Shoshani: *System Deadlocks*, ACM Computing Surveys 3(2): 67–78, 1971. DOI 10.1145/356586.356588 (slide set 11)

   are necessary for deadlocks, but not sufficient.

   Give an example of a case that meets the four criteria but does not cause a deadlock. To do this, specify what processes and resources exist, and in what order they are requested and allocated.

   Note: You do not have to limit yourself to resources with only one instance.

2. Specify a condition under which the four criteria are necessary and sufficient for deadlocks.

- It has to do with if a resource is unqiue or can have multiple instances (e.g. a printer spooler)
- as above

Exercise 3 (Ahmdal's rule) $(2+1+1+2+1+1=8$ points)

Programs can be decomposed into sequential and parallel parts. *Ahmdal's rule* (also Ahmdal's law), based on

> Gene Amdahl: *Validity of the Single Processor Approach to Achieving Large-Scale Computing Capabilities.* In AFIPS Conference Proceedings 30: 483–485, 1967. DOI 10.1145/1465482.1465560,

allows a quantitative estimate of the impact on execution time of speeding up the sequential parts and of using additional CPUs for the parallel parts. This can help, for example, to decide whether it is more worthwhile to speed up the sequential part or the parallel part.

Let T denote the total execution time of the program on a single CPU (i.e., completely sequential execution), B the time to execute the sequential parts, $T-B$ the time to execute the parallel parts on one CPU (sequential), and N the number of CPUs. Then the total execution time on N CPUs is given by

$$T(N) = B + (T-B)/N.$$

The total speed-up is given in terms of T as $T/T(N)$. If we set $T=1$, we get an acceleration by the factor

$$S = \frac{1}{B + (1-B)/N},$$

where B does not have units (for example, for $B = 10\,\mathrm{s}$ and $T = 50\,\mathrm{s}$, one would divide by T and get $T = 1$ and $B = 0.2$).

1. A program's sequential part is $1\,\%$. By what factor is the program accelerated when executed on 61 CPUs?
2. To which value does the speed-up S converge for $N \to \infty$?
3. Amdahl's rule does not take into account the cost of communication between CPUs. Extend the expression for the speed-up S to include communication costs C.
4. Two implementations of an algorithm have the same sequential fraction of $B = 0.001$, but differ in their communication costs of $C_1 = 0.0001N$ and $C_2 = 0.001\ln(N)$. Determine the number of CPUs N with maximum speed-up S for both implementations.
5. Plot the speed-up S as a function of the number of CPUs N for both algorithms and $1 \leq N \leq 1500$.
6. How do the two implementations differ qualitatively in the change in their speed-up when they use more CPUs than is optimal for them?

- Read carefully
- take the limit
- To what do the communication costs contribute to?
- Similar to 6.5.3 in terms of the steps
- depends on the communication costs

Exercise 4 (race conditions) $(1 + 2 + 4 = 9$ points)

Consider the adjacent program snippet.

```
const int n = 5000;
int total_count;

void total(void)
{
    int count = 0;
    for (; count < n; ++count)
        if (count % 2)
            --total_count;
        else
            ++total_count;
}
```

1. What is the value of `total_count` (depending on `n`) after a call to `total`?
2. The function `total` is executed in several threads simultaneously. Describe the race condition in the program snippet. Which variables are affected? What is the effect of the race condition?
3. Fix the race condition by using a semaphore. To do this, you can use a semaphore `int semaphore;` and the functions `void up(*int)` and `void down(*int)`.
4. Write a `C` program to test your solution. Use POSIX semaphores as well as POSIX threads to execute `total` concurrently. Compare your results for `total_count` for sequential execution without semaphore, parallel execution without semaphore, and parallel execution with semaphore.

   Note: Semaphores are an optional part of the POSIX standard and are not supported by all implementations. For example, `macOS` does not support unnamed semaphores, and for named semaphores does not support `sem_getvalue`. If you are working on an operating system without full POSIX semaphore support, online development environments such as repl.it may be a useful alternative.

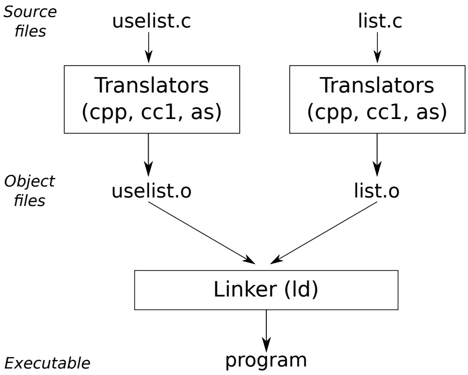
- Do this properly! Esp. 2) is a popular exam question.

Linking

12.1 Introduction

- We have already seen how separate compilation works (*cf.* page 179).
- The compiler driver gcc(1) employs a bunch of different tools for this task:

  - preprocessor `cpp`(1) — removes comments, applies macros.
  - compiler `cc1` — compiles into assembler code.
  - assembler `as`(1) — translates into binary object file.
  - linker `ld`(1) — **links together the compiled object files**.



- We'll have a closer look at linking now...

Object files

Object files contain chunks of data, (almost) ready to be copied to memory for execution.

- program code, *i.e.*, CPU instructions compiled from your program, and
- constant data (*e.g.*, string literals),

There are three kinds of object files:

- **Executable** object files can be executed directly, *cf.* page 301.
  - Generated by the linker, not by the compiler!
- **Relocatable** object files can be **linked** with other relocatable object files, to form an executable.
  - Symbols may change their position (*cf.* page 309), hence the name.
- **Shared** object files are relocatable object files that can be loaded into memory at runtime, and be shared amongst processes (*cf.* page **??**).

The functions, global variables, and `static` variables defined in an object file, can be referred to by name: The **symbols**.

Linker Symbols

Relocatable object files come with a **symbol table**, that lists all the symbols an object file exposes.

- **Global** symbols are defined in the object file, and may be referenced from other object files.
- **External** symbols are referenced by the object file, but not defined. *I.e.*, the definition must be provided in another object file.
- **Local** symbols are defined and referenced only from within the object file.

Note *Local symbols* have nothing to do with function-local variables in a C-program. Unless `static`, they are never visible in the symbol table.

Example

```c
extern int buf[];
int *bufp0 = &buf[0];
int *bufp1;

void swap(void)
{
    int temp;
    static int count = 42;

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

    count++;
}
```

```
$ pk-cc -c swap.c
$ readelf -s swap.o                  # cf. readelf(1)
Symbol table '.symtab' contains 18 entries:
   Num: Size Type    Bind   Ndx Name
# ...
    5:     4 OBJECT  LOCAL    3 count.1597
# ...
   14:     8 OBJECT  GLOBAL   3 bufp0
   15:     0 NOTYPE  GLOBAL UND buf
   16:     8 OBJECT  GLOBAL COM bufp1
   17:    74 FUNC    GLOBAL   1 swap
```

(some lines and columns have been removed)

- The local symbol count (has its name extended to avoid name clashes) uses 4 bytes, and will be stored in section 3 (Section? *cf.* page 305)
- Object `bufp0` uses 8B in section 3, function `swap` uses 74B in section 1.
- `buf` is `UND`efined, *i.e.*, referenced by this module, but we have no idea where it will be in the compiled program.
- `COM`mon objects, like `bufp1` are uninitialized, and not yet allocated.

12.2 Symbol resolution

- For each **local symbol**, the compiler guarantees exactly one definition. The name is modified to be unique (*e.g.* `count` above).
- If *the compiler* **finds no definition**, it expects it to come from another module, and leaves it to the linker, (*e.g.* `buf` above).
- When **the linker** resolves *global* symbols, several conditions can occur:
  - **No definition** is found in the symbol table of any input object file.
  - **Multiple definitions** are found in different object files, and one must be chosen.

Example No `main` function, and `buf` undefined.

```
$ pk-cc swap.o #without -c, try to build an executable
.../lib/crt1.o: In function `_start':
(.text+0x20): undefined reference to `main'
swap.o: In function `swap':
.../swap.c:12: undefined reference to `buf'
swap.o:(.data+0x0): undefined reference to `buf'
collect2: error: ld returned 1 exit status
```

- The linker tries to link with `crt1.o`, wich refers to the `main` function.

What else?

- After resolving symbols, the linker knows which definition belongs to each symbol.

Recall

- Machine code does not use variable names any more.
- The compiler produced code that accesses variables and functions only by their **memory addresses**.

$\Rightarrow$ How does this go together with separate compilation and symbol resolution?

12.3 The program in memory

How does a program start?

- When a program is run, it is **copied into memory** by the **loader**.
  - Copy **text segment**, *i.e.*, the actual machine code,
  - copy **initialized data**,
  - **initialize** uninitialized data,
  - *etc.*
- We want to minimize the amount of data to be copied!
  - Only load parts that are **actually required**,
  - and only load them **when** they are needed.
- Wa want to save memory!
  - Do not load the same code into memory multiple times.
  - **Share** already loaded code between processes.
- Avoid expensive **transformations**
  - Store program on disk in a **format** that allows fast setup of the process image.

Virtual memory

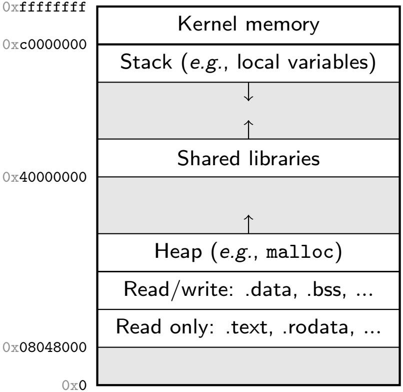
- VM is a mapping from the process' **virtual address space** into the machine's physical address space (organized in **pages**).
- The VM system may flag pages as, *e.g.*, **read only**, **executable**, or **private**, *cf.* `mmap`(2).
- A physical page may reside on disk, until **loaded on demand**.
  - So we compile the memory layout into the executable file,
  - the loader just **maps the file** into the process' virtual address space, and
  - the VM system gets the pages into memory when actually referenced.
- Multiple running instances of a program share their text (machine code) through a *read only* mapping to **the same physical** address space.

Note To achieve all this, the structure of the program file depends on the process' memory layout!

Process memory layout

When running, a process has the following **virtual memory layout**.

(This is for 32bit Linux)



- Kernel memory (1GiB) is not accessible by the process.
  - **Shared** among all processes.
- The stack maintains local variables and function calls.
- Shared libraries may even be added at runtime.
- The heap contains allocated memory.
- `.data` and `.bss` store global variables.
- `.text` and `.rodata` are marked `ro`, so can be shared with other processes.

12.4 Object file layout

- There are various formats to store binary programs.
- Linux uses ELF, the **Exectable and Linking Format**.
- COFF and `a.out` are others, the latter coined the name used by `gcc` for default binaries (in ELF on Linux!).
- All formats have the concept of **sections** in common.
- A section is the unit of organization in a binary.

Some section names (but there are many more)

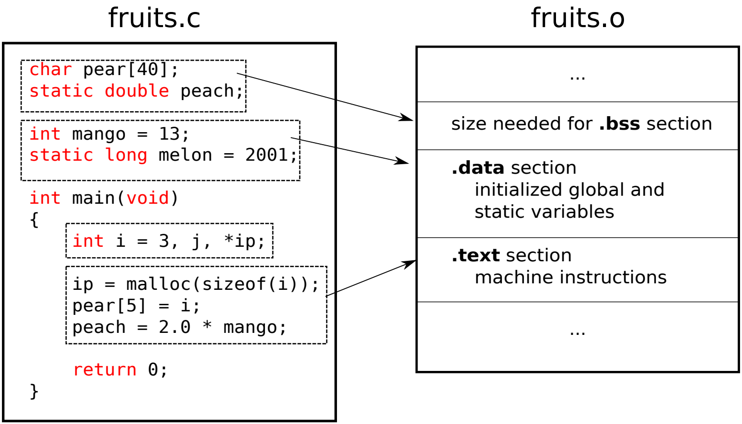
`.text` The program code, *i.e.*, processor instructions.

`.rodata` Read-only data, *e.g.*, string literals.

`.data` *Initialized global* variables.

`.bss` *Uninitialized global* variables.

`.symtab` The symbol table, displayed with `readelf -s`.


fruits.c
char pear[40];
static double peach;
int mango = 13;
static long melon = 2001;
int main(void)
{
int i = 3, j, *ip;
ip = malloc(sizeof(i));
pear[5] = i;
peach = 2.0 * mango;
return 0;
}
fruits.o
...
size needed for .bss section
.data section
initialized global and
static variables
.text section
machine instructions
...

A typical *relocatable* object file

This is what the compiler produces out of the **individual C files**.

	ELF header
Sections	`.text`
	`.rodata`
	`.data`
	`.bss`
	`.symtab`
	`.rel.text`
	`.rel.data`
	`.debug`
	`.line`
	`.strtab`
	Section header table

- The **ELF header** describes word size, endian, object file type, machine type, offset and format of the section header table, and other information.
- The **section header table** describes the locations of the various sections.
- Try

```
$ pk-cc -c -m32 swap.c
$ readelf -S swap.o
```

A typical *executable* object file

That is what we want to have in the **final binary program**.

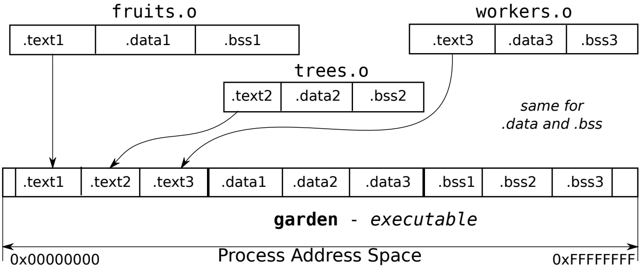
	ELF header
	Segment header table
Code segment mapped read-only	`.init`
	`.text`
	`.rodata`
Data segment mapped read/write	`.data`
	`.bss`
Symbol table and debug info are not loaded into memory	`.symtab`
	`.debug`
	`.line`
	`.strtab`
	Section header table

- The **segment header table** describes the mapping of contiguous file sections into memory.
- Try this

```
$ pk-cc -m32 -static swap.c \
> main.c
$ readelf -l a.out
```

12.5 Relocation

- So after resolving the symbols, the linker needs to put all the code from the individual object files' sections into the final program's sections:



- This process is called **Relocation**.

Relocation involves **two tasks**:

- Relocating **sections** and symbol **definitions**.
  - Merge sections of the same type.
  - Assign **run-time memory addresses** to the new aggregate sections, the input sections, and each symbol defined in the input.
  - ⇒ After this step, every global symbol has a known run-time memory address.
- Relocating symbol **references** everywhere in the code.
  - Modification of each reference in `.text` and `.data`, so that they point to correct location.

Relocation entries

- The assembler does not know where data and code will be stored ultimetly,
- nor does it know addresses of the external objects.

⇒ In such situations a *relocation entry* is generated by the assembler.

Relocation entries

```
typedef struct {
  int offset;       /* Offset of the reference to relocate. */
  int symbol: 24,   /* Symbol the reference should point to. */
      type:8;       /* Relocation type. */
} Elf32_Rel;
```

- ELF defines 11 different relocation types
- We will look at `R_386_32` only.
  This is used to relocate 32bit absolute addresses.

Example: Relocation at work

- Function `f` simply assigns `0xbeef` to the global variable `x`.
- The final memory location of `x` is not yet known.
- Relocation will fix the runtime address of `x`.

foo6.c

```
int x = 0xdead;

void f(void)
{
    x = 0xbeef;
}
```

- First we **compile** `foo.c` into a *relocatable object file*:

```
$ pk-cc -m32 -c foo6.c
```

- Have a look at the `.data` section:

```
$ objdump -d -j.data foo6.o
Disassembly of section .data:
00000000 <x>:
   0:   ad de 00 00                                      ....
```

- Variable `x` appears at address `0x0` in the `.data` section.
- The value `0xdead` is stored there.

- Have a look at the `.text` section:

```
1 $ objdump -d -j.text foo6.o
2 Disassembly of section .text:
3 00000000 <f>:
4    0:   55                      push   %ebp
5    1:   89 e5                   mov    %esp,%ebp
6    3:   c7 05 00 00 00 00 ef    movl   $0xbeef,0x0
7    a:   be 00 00
8    d:   5d                      pop    %ebp
9    e:   c3                      ret
```

  - In line 6, the value `0xbeef` is copied to address `0x0`.
  - This address `0x0` appears at **offset 5** in the `.text` section.

- These are the relocation entries:

```
$ objdump -r -j.text foo6.o
RELOCATION RECORDS FOR [.text]:
OFFSET   TYPE              VALUE
00000005 R_386_32          x
```

  - So on relocation of symbol `x`, the absolute 32bit address at **offset 5** in the `.text` section must be updated.

- Then we **link** `foo.o` with something that uses `f`.

```
$ pk-cc -m32 foo6.o bar6.c                 /* main in bar6.c simply calls f in foo6.c */
```

- Have a look at the `.data` section **after relocation**:

```
$ objdump -d -j.data a.out
Disassembly of section .data:
08049698 <x>:
 8049698:       ad de 00 00                                        ....
```

  - Variable `x` has been moved to address `0x8049698` in the `.data` section.
  - The value `0xdead` is stored there.

- Have a look at the `.text` section **after relocation**:

```
$ objdump -d -j.text a.out | grep -C3 beef
080483cd <f>:
 80483cd:       55                      push   %ebp
 80483ce:       89 e5                   mov    %esp,%ebp
 80483d0:       c7 05 98 96 04 08 ef    movl   $0xbeef,0x8049698
 80483d7:       be 00 00
 80483da:       5d                      pop    %ebp
 80483db:       c3                      ret
```

  - The reference to variable `x` has been **updated to address** `0x8049698`.

Example: Relocation in the `.data` section

- Sometimes, updating references in the `.text` section is not enough.
- Here we have a global variable `xp` initialized with an address!
- The compiler **cannot even fix a value** for `xp`!

foo7.c

```c
int x = 0xdead;
int *xp = &x;

void f(void)
{
    *xp = 0xbeef;
}
```

- Again, we compile and link our object files:

```
$ pk-cc -m32 -c foo7.c
$ pk-cc -m32 foo7.o bar6.c
```

- Relocation in the `.data` section.

```
$ objdump -d -j.data foo7.o
Disassembly of section .data:
00000000 <x>:
   0:   ad de 00 00                                          ....
00000004 <xp>:
   4:   00 00 00 00    # This value has to be updated on relocation of x!
$ objdump -r -j.data foo7.o    # Note: in .data this time!
RELOCATION RECORDS FOR [.data]:
OFFSET   TYPE              VALUE
00000004 R_386_32          x
$ objdump -d -j.data a.out
Disassembly of section .data:
08049698 <x>:
 8049698:       ad de 00 00                                         ....
0804969c <xp>:
 804969c:       98 96 04 08                                         ....
```

- Relocation in the .text section:

```
$ objdump -d -j.text foo7.o
Disassembly of section .text:
00000000 <f>:
   0:   55                      push   %ebp
   1:   89 e5                   mov    %esp,%ebp
   3:   a1 00 00 00 00          mov    0x0,%eax
   8:   c7 00 ef be 00 00       movl   $0xbeef,(%eax)
   e:   5d                      pop    %ebp
   f:   c3                      ret
$ objdump -r -j.text foo7.o
RELOCATION RECORDS FOR [.text]:
OFFSET   TYPE              VALUE
00000004 R_386_32          xp
$ objdump -d -j.text a.out | grep -C3 beef
 80483cd:       55                      push   %ebp
 80483ce:       89 e5                   mov    %esp,%ebp
 80483d0:       a1 9c 96 04 08          mov    0x804969c,%eax
 80483d5:       c7 00 ef be 00 00       movl   $0xbeef,(%eax)
 80483db:       5d                      pop    %ebp
 80483dc:       c3                      ret
```

- Recall: 0x804969c is the address of the variable xp!

12.6 Static libraries

- A static library is a **collection of relocatable object files**.
  - Since the term “object file” is not correct in this context, the members of a library are referred to as **object modules** instead.
- Linking with a library means to link with all the **required** object files.

Why use libraries at all?

- Why not put all library functions into one relocatable object file?
  - An object module is added to a program in its **entirety, or not at all**.
  - The potentially **large object file** would be added to every binary using one of the functions. ⇒ Waste of space.
- Why not copy only **required functions** from an object file?
  - Sections like `.text` and `.data` are merely binary blocks to the linker.
- Why not **explicitly link** all the required object files with the binary?
  - Tedious with object modules at the granulatiry of individual functions!

```
$ gcc -o main main.c printf.o atoi.o read.o write.o ...
```

Making a static library

- A static library is an **archive** of relocatable object modules

`ar rcs` archive [member...] Create archive, containing the members.

`nm` archive List symbols from object files or archives.

- The `ar`(1) command provides various means to modify an archive.
  - `r` Add the members to archive, **replace** existing with the same name.
  - `c` **Create** the archive if it does not exist.
  - `s` Write an object-file **index** into the archive.
  - ... many others
- `nm`(1) displays the symbols defined in a module, or a library.

- By the way: See `info binutils` for an overview of tools for manipulating ELF binaries.

Example

addvec.c

```c
#include "addvec.h"

void addvec(int n, int *x, int *y,
    int *z)
{
    for (int i = 0; i < n; i++)
        z[i] = x[i] + y[i];
}
```

dotproduct.c

```c
#include "dotproduct.h"

int dotproduct(int n, int *x, int *y)
{
    int r = 0;
    for (int i = 0; i < n; i++)
        r += x[i] * y[i];
    return r;
}
```

- The header files just contain the respective function prototype.

```
$ pk-cc -c addvec.c                    # usually, all this is arranged for in a Makefile
$ pk-cc -c dotproduct.c
$ ar rcs libvector.a addvec.o dotproduct.o
$ nm libvector.a   # a shiny new library
addvec.o:
0000000000000000 T addvec
dotproduct.o:
0000000000000000 T dotproduct
```

main.c

```c
#include "dotproduct.h"
#include <stdio.h>

int main(void)
{

	int x[3] = { 1, 0, 0 }, y[3] = { 0, 1, 1 }, z[3] = { 1, 1, 0 };

	printf("<x,y> = %d\n", dotproduct(3, x, y));
	printf("<x,z> = %d\n", dotproduct(3, x, z));

	return 0;
}
```

- The `-static` flag tells gcc to build a **statically linked** binary.

```
$ pk-cc -c main.c
$ pk-cc -static -omain main.o libvector.a
$ ./main
<x,y> = 0
<x,z> = 1
```

Linker flags for static libraries

- Typically, libraries do not reside in the directory where they are used.
  - With `-lname` the library `libname.a` is searched for in the library search path.
  - With `-Ldir`, a directory is added to the **library search path**.
  - The library search path is searched in the order of the `-L` options.

```
$ gcc -static -omain main.o -L. -lvector      # link with the libvector.a library
```

- The **order** in which libraries are given on the **command line** is significant, and counter-intuitive:
  - The library providing a symbol must appear **after** the object using it.

```
$ gcc -static -omain libvector.a main.o
main.o: In function ‘main':
/home/sk/uni/teach/inf3_14w/pk/lect/src/lib/main.c:10: undefined reference
to ‘dotproduct'
/home/sk/uni/teach/inf3_14w/pk/lect/src/lib/main.c:12: undefined reference
to ‘dotproduct'
collect2: error: ld returned 1 exit status
$ gcc -static -omain -L. -lvector main.o
# the same error message
```

Symbol resolution with static libraries

Input : Files passed to the linker on the command line
Output: A statically linked binary
Data : The set of object modules O **to be linked** to the binary, the set of referenced but yet **unresolved symbols** U, and the set of already **defined symbols** D

$O \leftarrow \emptyset$; $U \leftarrow \emptyset$; $D \leftarrow \emptyset$
foreach input file f given on the command line **do**
  if f is an object file **then**
    $O \leftarrow O \cup \{f\}$; $D \leftarrow D \cup \mathrm{global}\, f$; $U \leftarrow (U \setminus D) \cup \mathrm{external}\, f$
  else if f is an archive **then**
    repeat
      foreach object module m which is a member of f **do**
        if $U \cap \mathrm{global}\, m \neq \emptyset$ **then**
          $O \leftarrow O \cup \{m\}$; $D \leftarrow D \cup \mathrm{global}\, m$; $U \leftarrow (U \setminus D) \cup \mathrm{external}\, m$
    until U and D do not change anymore
if $U \neq \emptyset$ **then**
  Fail with error message: Undefined references to all symbols in U
else
  Relocate object modules in O and build executable.

($\mathrm{global}\, m =$ symbols defined in m; $\mathrm{external}\, m =$ symbols not defined in, but referenced by m)

- Review the previous example: Why does this fail?

```
$ gcc -static -omain libvector.a main.o     # Wrong!
```

  - *U* is empty when `libvector.a` is visited,
  - so no object modules are added to *O*.
  - When `main.o` is checked, `dotproduct` is added to *U*, but `libvector.a` is not visited again.

- Sometimes, it is necessary to specify a library multiple times on the command line. Example (pseudocode!):

```
main.o
    main() { foo(); }
```

```
libfoobar.a
    foo.o
        foo() { ding(); }
    bar.o
        bar() { dong(); }
```

```
libdingdong.a
    ding.o
        ding() { bar(); }
    dong.o
        dong() { foo(); }
```

```
$ gcc -static -omain main.o -L. -lfoobar -ldingdong -lfoobar -ldingdong
```

12.7 Shared libraries

- Shared libraries are linked to the program **not until runtime**.
- **Different programs** can use the same shared library.
- The tool `ldd`(1) lists the **dynamic dependencies** of a thus linked binary.

Example

- The binary created in the previous section is quite big:

```
$ gcc -static -o main main.o -L. -lvector
$ ls -l main
-rwx------ 1 sk users 826k Feb  3 18:37 main
$ ldd main
        not a dynamic executable
```

- This is, because the `-static` flag enforces static linking, including way more than only `libvector`.

- Without `-static`, **dynamic linking** is used where possible:

```
$ gcc -o main main.o -L. -lvector
$ ls -l main
-rwx------ 1 sk users 8.7k Feb  3 18:44 main
$ ldd main
        linux-vdso.so.1 (0x00007fffc013a000)
        libc.so.6 => /usr/lib/libc.so.6 (0x00007ff560b8c000)
        /lib64/ld-linux-x86-64.so.2 (0x00007ff560f36000)
```

  - `linux-vdso.so.1` (Virtual Dynamic Shared Objects) is a part of the kernel, providing **fast system calls**. It is not a shared library in the usual sense.
  - `libc.so.6` is the **standard C library** on Linux systems.
  - `ld-linux-x86-64.so.2` contains the ELF **dynamic linker and loader**.

  Most Programs (unless compiled `-static`) will depend on these.

- Obviously, `libvector.a` is **not shared**, but still statically linked.
  $\Rightarrow$ How can we change this?

Making a shared library

- The individual object modules have to be compiled with `-fPIC`.
  - This generates **position-independent code**, which allows relocation later on, *cf.* page 332.
- Instead of using `ar(1)`, the object files are linked together into a **single shared object file** with `.so` suffix.

Example To build a shared `libvector` instead of a static one:

```
$ pk-cc -c -fPIC addvec.c
$ pk-cc -c -fPIC dotproduct.c
$ gcc -shared -o libvector.so addvec.o dotproduct.o
$ nm libvector.so     # my first shared library
0000000000000665 T addvec
# ...
00000000000006d5 T dotproduct
# ...
```

Using a shared library

- The shared library is used just like a static library:

```
$ pk-cc -c main.c
$ gcc -o main main.o -L. -lvector
```

- The generated binary now **depends** on `libvector.so`:

```
$ ldd main
        linux-vdso.so.1 (0x00007fffe8eaa000)
        libvector.so => not found
        libc.so.6 => /usr/lib/libc.so.6 (0x00007f2cbc343000)
        /lib64/ld-linux-x86-64.so.2 (0x00007f2cbc6ed000)
$ ./main
./main: error while loading shared libraries: libvector.so: cannot open
shared object file: No such file or directory
```

- The dynamic linker `ld-linux`(8) only searches a **default path**:
  - Falling back to `/lib`, and `/usr/lib`, but see the manual!
  - The search path can be extended (prefixed) with `$LD_LIBRARY_PATH`.

```
$ LD_LIBRARY_PATH=. ./main
<x,y> = 0
<x,z> = 1
```

Choosing a shared library at runtime

- Applications can **decide on which shared libraries to load** at runtime.

```
#include <dlfcn.h>

void *dlopen(const char *filename, int flag);
void *dlsym(void *handle, const char *symbol);
int dlclose(void *handle);
char *dlerror(void);
```

- `dlopen`(3) **loads** a shared library, and returns a handle to it.
  - See the manual for how the library is **searched**.
  - The flag indicates when/how to **resolve symbols**:

    `RTLD_NOW` Before `dlopen` returns, or

    `RTLD_LAZY` when the called function is needed.

    … further flags are available
- `dlsym`(3) returns a **pointer to the symbol** named.
- `dlclose`(3) **unloads** a shared library if it is not used anymore.
- `dlopen`(3) and `dlsym`(3) return `NULL` on failure. `dlerror`(3) returns a string describing the most recent error.
- Programs using this interface **must be linked** with `-ldl`.

Example

```
typedef int (*dotproduct_t)(int, int *, int *);

int main(int argc, char *argv[])
{
    dotproduct_t dotproduct;
    void *handle;

    if (argc < 2) errx(1, "use: sick <lib>");

    handle = dlopen(argv[1], RTLD_NOW);
    if (handle == NULL) errx(1, "dlopen: %s", dlerror());

    /* dotproduct = (dotproduct_t)dlsym(handle, "dotproduct"); */
    *(void **)(&dotproduct) = dlsym(handle, "dotproduct");
    if (dotproduct == NULL) errx(1, "dlsym: %s", dlerror());

    int x[3] = { 1, 0, 0 }, y[3] = { 0, 1, 1 }, z[3] = { 1, 1, 0 };
    printf("<x,y> = %d\n", dotproduct(3, x, y));
    printf("<x,z> = %d\n", dotproduct(3, x, z));

    dlclose(handle);
    return 0;
}
```

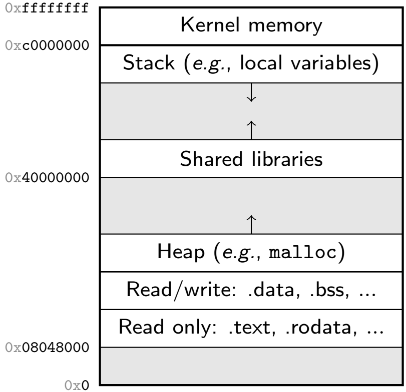
```
$ ./sick ./libvector.so
<x,y> = 0
<x,z> = 1
$ ./sick ./libfake.so
<x,y> = 42
<x,z> = 42
```

- Some projects use this mechanism to provide a **plugin interface**.
- If the program is compiled with `-rdynamic`, then a loaded library can use the program's global symbols.

12.8 Position-independent code

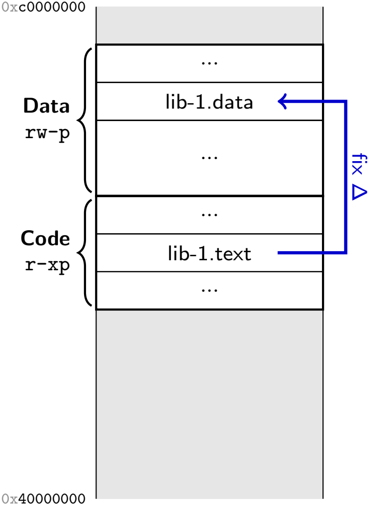
- **Different programs** should use a shared library simultaneously.
- Mapping is likely to happen to **different virtual memory regions**.
- Simple **relocation breaks sharing**, since it modifies `.text`.

How can we solve this?



- Recall process memory layout:
  - Shared libraries' `.text` and `.data` is **not merged** with the main program's.
  - Instead, each shared library is loaded **somewhere above** `0x40000000`.
- **Position-independent code** is compiled in a way that allows it to be executed at any address, without prior relocation.

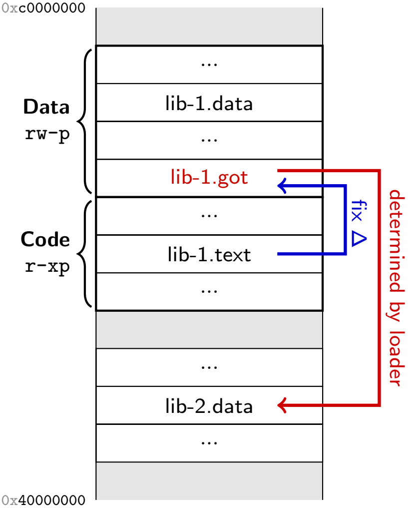
Memory layout of a shared library



- The data segment of a shared library is mapped **directly after** the code segment.
  - For each access to a local symbol, the **distance from instruction to variable** is fix!
  - This Δ is known at **compile time**.
  - Variable access is implemented by **offset from the program counter**, instead of absolute addresses.

- How can we access variables in **other modules**?

Memory layout of a shared library



- PIC adds **one level of indirection** to access external symbols.
  - A **global offset table** (GOT) is added at the **start of the data segment** of every module.
  - At compile time, references to variables are replaced by **indirect references** via the GOT.
  - The **dynamic loader fills the GOT** with the correct addresses (relocation) at runtime.
- Thus, relocation **happens in the private data segment**!
  - The code segment can be shared.

Final Remarks

- Relocation for **function calls** also uses the GOT.
  - A more sophisticated algorithm, called **lazy binding**, reduces the overhead after the first function call.
- Shared libraries come with a **runtime overhead** for accessing any external symbols.
- Using shared libraries requires **expensive setup of all GOTs** when loading a program.
- Shared libraries **increase code sharing** more than static libraries.
  - Static library code **cannot be shared between different programs**, only between different instances of the same program.

- An in-depth discussion about shared libraries can be found here:
  - Ulrich Drepper. *How To Write Shared Libraries.* December 2011, `http://www.akkadia.org/drepper/dsohowto.pdf`.

References I

[1] W. Stallings, *Operating systems: internals and design principles*. Upper Saddle River, NJ: Pearson/Prentice Hall, 2009.
[2] A. Silberschatz, P. B. Galvin und G. Gagne, *Operating system principles*. John Wiley & Sons, 2006.
[3] A. S. Tanenbaum und H. Bos, *Modern operating systems*. Pearson, 2015.
[4] B. W. Kernighan und D. M. Ritchie, *The C programming language*. 2006.
[5] D. Trugman. (6. Jan. 2021), „ELF Loaders, Libraries and Executables on Linux | by Daniel Trugman | Medium“, Adresse: `https://medium.com/@dtrugman/elf-loaders-libraries-and-executables-on-linux-e5cfce318f94` (besucht am 06. 01. 2021).
[6] (6. Jan. 2021). „ld.so(8): dynamic linker/loader - Linux man page“, Adresse: `https://linux.die.net/man/8/ld.so` (besucht am 06. 01. 2021).
[7] (6. Jan. 2021). „ld(1): GNU linker - Linux man page“, Adresse: `https://linux.die.net/man/1/ld` (besucht am 06. 01. 2021).
[8] (25. Nov. 2019). „How programs get run [LWN.net]“, Adresse: `https://lwn.net/Articles/630727/` (besucht am 06. 01. 2021).
[9] (8. Sep. 2018). „How programs get run: ELF binaries [LWN.net]“, Adresse: `https://lwn.net/Articles/631631/` (besucht am 06. 01. 2021).
[10] (30. Nov. 2020). „A Whirlwind Tutorial on Creating Really Teensy ELF Executables for Linux“, Adresse: `http://www.muppetlabs.com/~breadbox/software/tiny/teensy.html` (besucht am 06. 01. 2021).
[11] (6. Jan. 2021). „linux/binfmt_elf.c at fcadab740480e0e0e9fa9bd272acd409884d431a ů torvalds/linux ů GitHub“, Adresse: `https://github.com/torvalds/linux/blob/fcadab740480e0e0e9fa9bd272acd409884d431a/fs/binfmt_elf.c` (besucht am 06. 01. 2021).
[12] (21. Dez. 2020). „elf(5) - Linux manual page“, Adresse: `https://man7.org/linux/man-pages/man5/elf.5.html` (besucht am 06. 01. 2021).
[13] (6. Jan. 2021). „Linux Foundation Referenced Specifications“, Adresse: `https://refspecs.linuxfoundation.org/` (besucht am 06. 01. 2021).