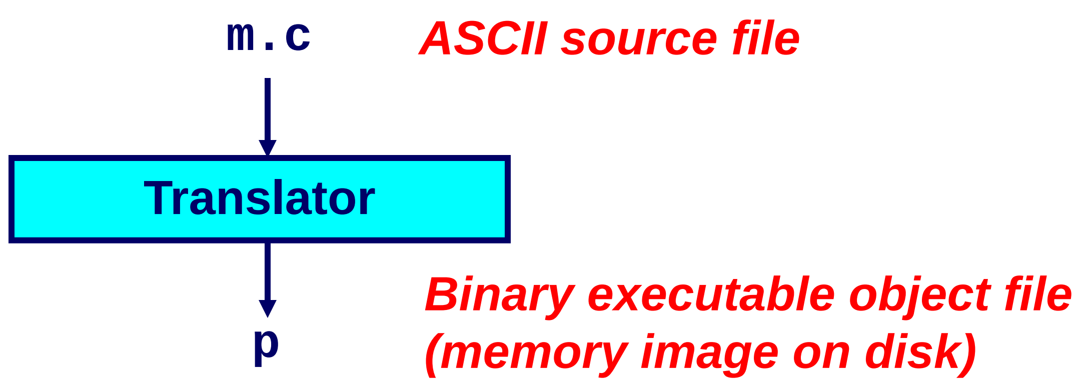
A Simplistic Program Translation Scheme



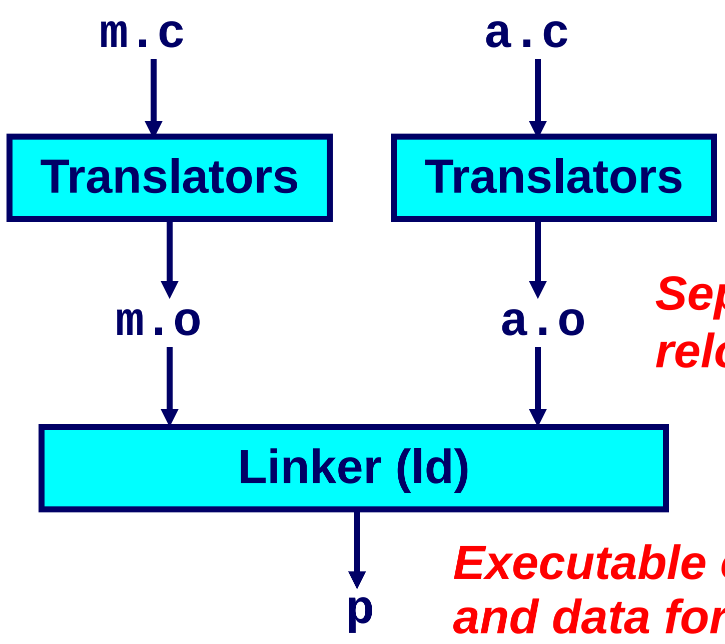
Problems:

- **Efficiency: small change requires complete recompilation**
- **Modularity: hard to share common functions (e.g. `printf`)**

Solution:

- ***Static linker (or linker)***

A Better Scheme Using a Linker



Separately compiled relocatable object files

Executable object file (contains code and data for all functions defined in `m.c` and `a.c`)

Translating the Example Program

Compiler driver coordinates all steps in the translation and linking process.

- Typically included with each compilation system (e.g., gcc)
- Invokes preprocessor (cpp), compiler (cc1), assembler (as), and linker (ld).
- Passes command line arguments to appropriate phases

Example: create executable p from `m.c` and `a.c`:

```
bass> gcc -O2 -v -o p m.c a.c
cpp [args] m.c /tmp/cca07630.i
cc1 /tmp/cca07630.i m.c -O2 [args] -o /tmp/cca07630.s
as [args] -o /tmp/cca076301.o /tmp/cca07630.s
<similar process for a.c>
ld -o p [system obj files] /tmp/cca076301.o /tmp/cca076302.o
bass>
```

What Does a Linker Do?

Merges object files

- Merges multiple relocatable (.o) object files into a single executable object file that can be loaded and executed by the loader.

Resolves external references

- As part of the merging process, resolves external references.
  - *External reference*: reference to a symbol defined in another object file.

Relocates symbols

- Relocates symbols from their relative locations in the `.o` files to new absolute positions in the executable.
- Updates all references to these symbols to reflect their new positions.
  - References can be in either code or data
    - » code: `a();            /* reference to symbol a */`
    - » data: `int *xp=&x;  /* reference to symbol x */`

Why Linkers?

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

Efficiency

- Time:
  - Change one source file, compile, and then relink.
  - No need to recompile other source files.
- Space:
  - Libraries of 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.

Executable and Linkable Format (ELF)

Standard binary format for object files

Derives from AT&T System V Unix

- Later adopted by BSD Unix variants and Linux

One unified format for

- Relocatable object files (`.o`)
- Executable program files (`a.out`)
- Shared object library files (.so)
- Core dump files

Generic name: ELF binaries

ELF Object File Format

Elf header

- Magic number, type (.o, exec, .so,core), machine, byte ordering, etc.

Program header table

- Page size, virtual addresses memory segments (sections), segment sizes.

`.text` section

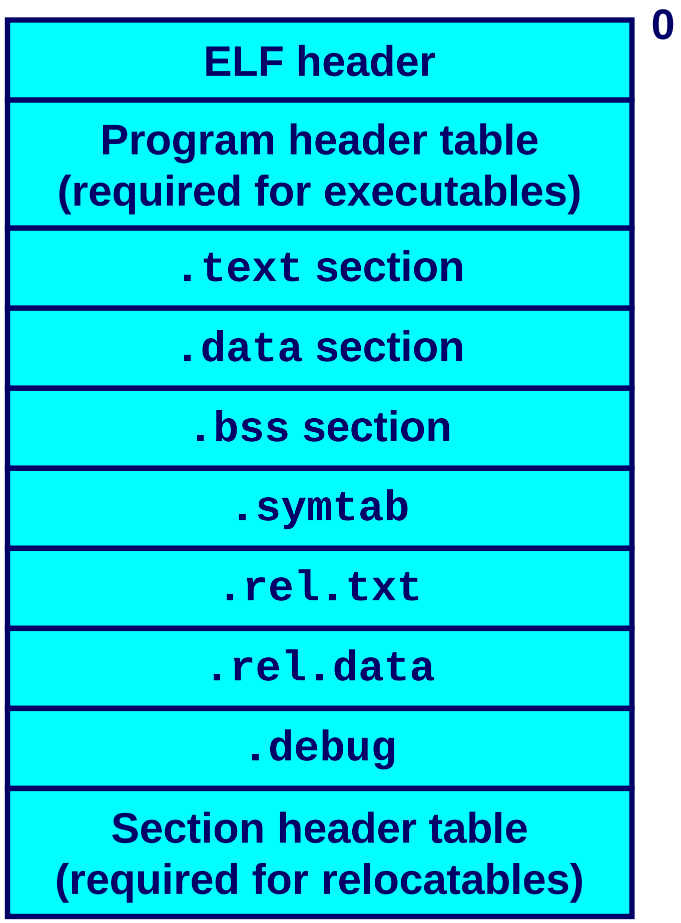
- Code

`.data` section

- Initialized (static) data

`.bss` section

- Uninitialized (static) data
- "Block Started by Symbol"
- "Better Save Space"
- Has section header but occupies no space


0
ELF header
Program header table (required for executables)
.text section
.data section
.bss section
.symtab
.rel.txt
.rel.data
.debug
Section header table (required for relocatables)

ELF Object File Format (cont)

`.symtab` section

- Symbol table
- Procedure and static variable names
- Section names and locations

`.rel.text` section

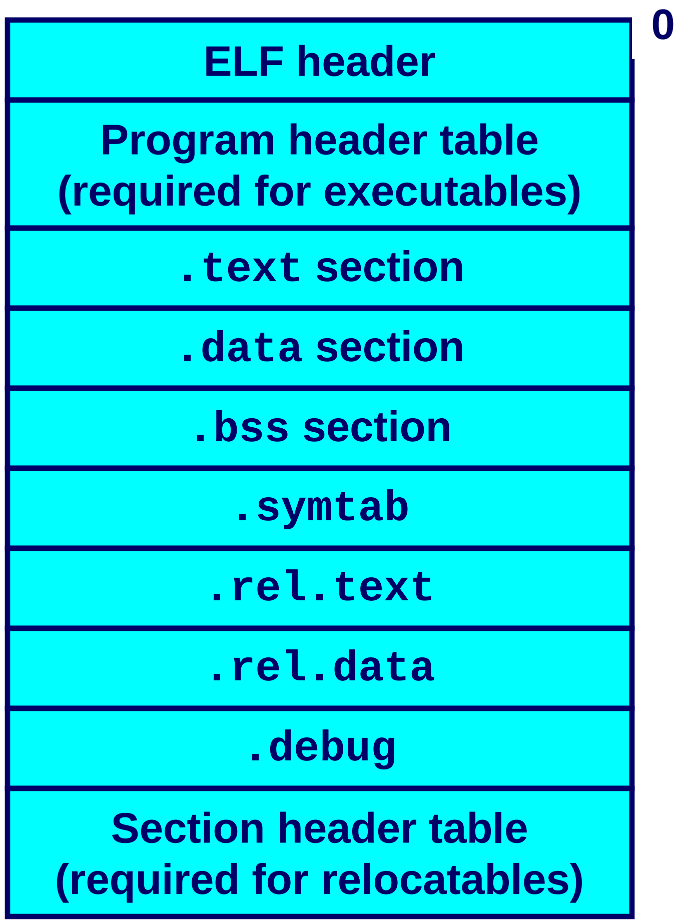
- Relocation info for `.text` section
- Addresses of instructions that will need to be modified in the executable
- Instructions for modifying.

`.rel.data` section

- Relocation info for `.data` section
- Addresses of pointer data that will need to be modified in the merged executable

`.debug` section

- Info for symbolic debugging (`gcc -g`)


0
ELF header
Program header table (required for executables)
.text section
.data section
.bss section
.symtab
.rel.text
.rel.data
.debug
Section header table (required for relocatables)

Example C Program

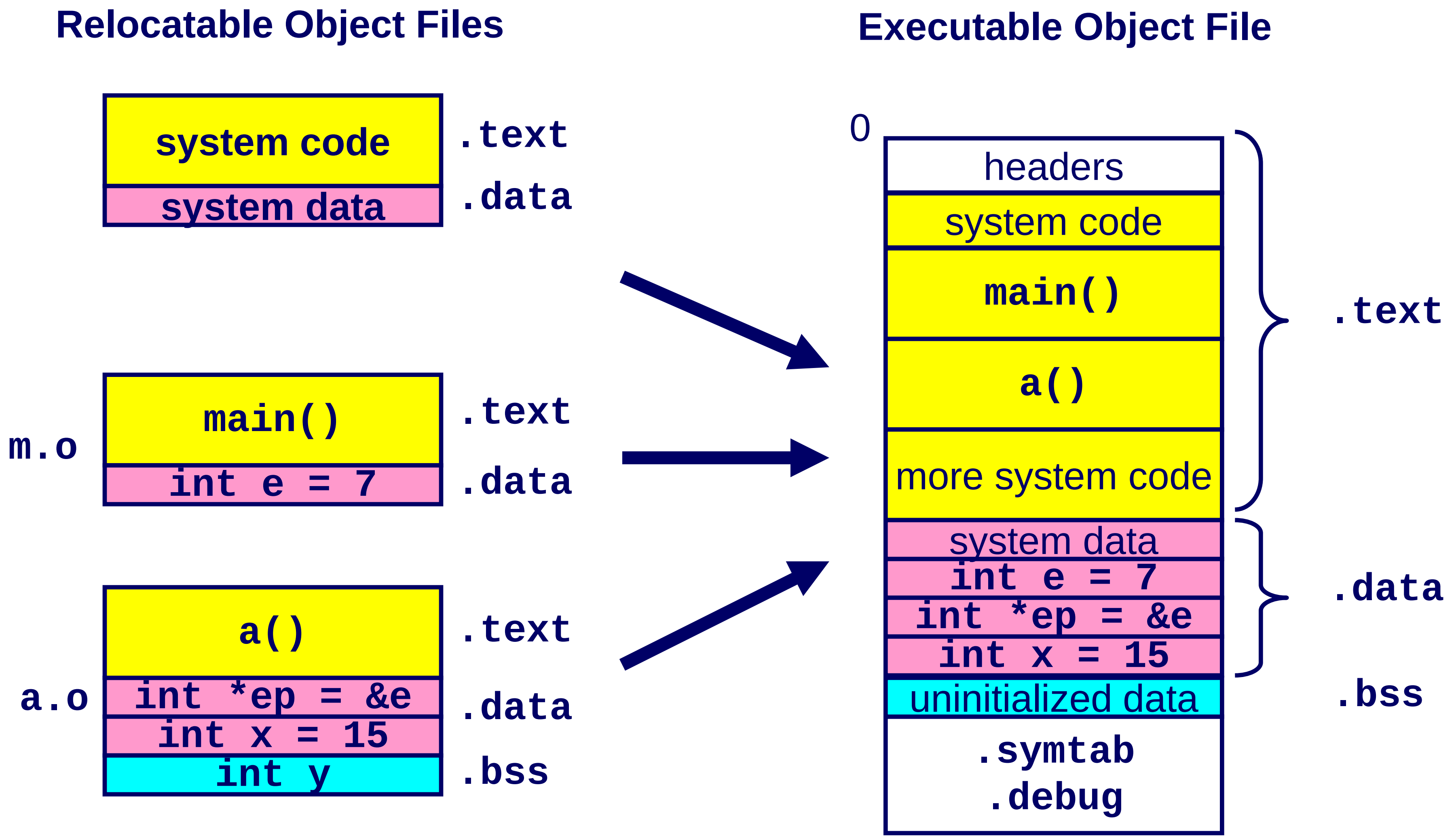
m.c

```
int e=7;

int main() {
  int r = a();
  exit(0);
}
```

a.c

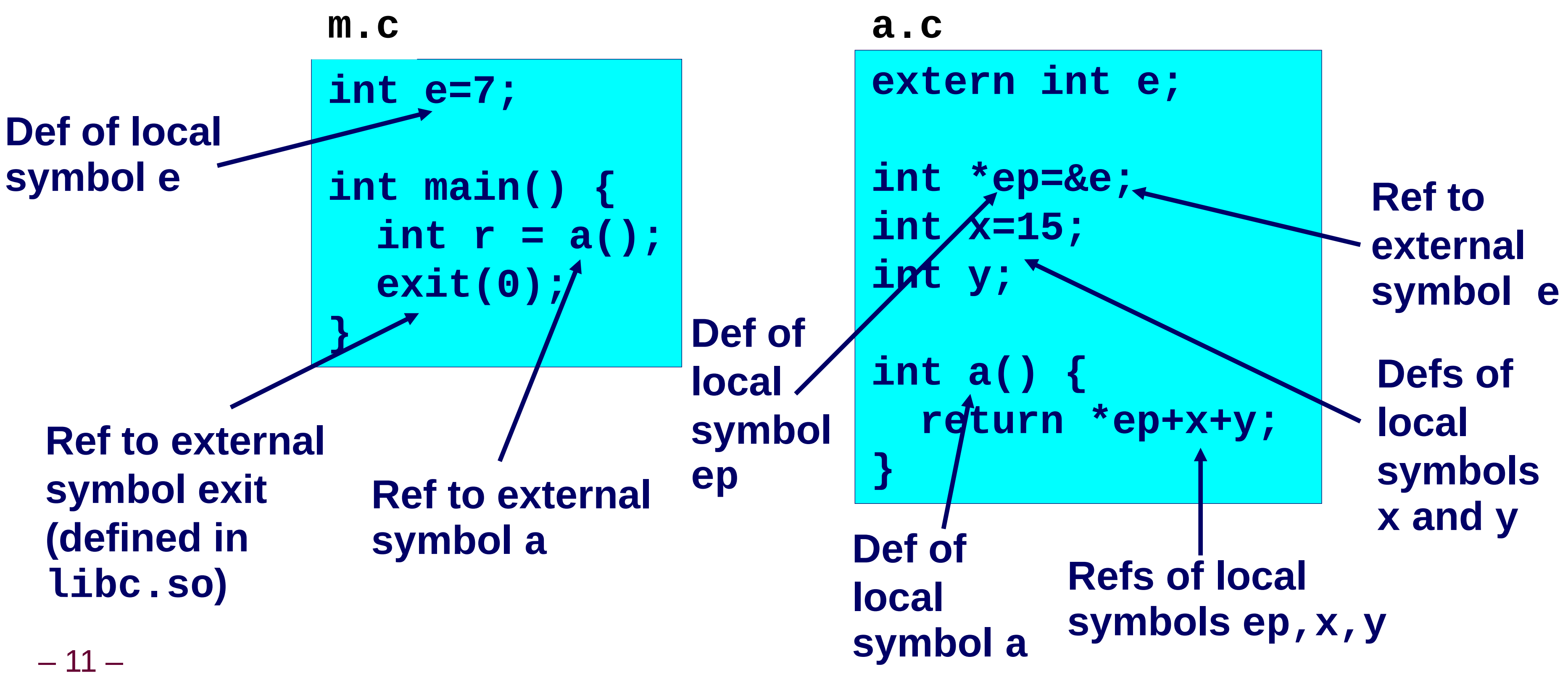
```
extern int e;

int *ep=&e;
int x=15;
int y;

int a() {
  return *ep+x+y;
}
```

Merging Relocatable Object Files into an Executable Object File



Relocating Symbols and Resolving External References

- *Symbols* are lexical entities that name functions and variables.
- Each symbol has a *value* (typically a memory address).
- Code consists of symbol *definitions* and *references*.
- References can be either *local* or *external*.



m.c

```
int e=7;

int main() {
  int r = a();
  exit(0);
}
```

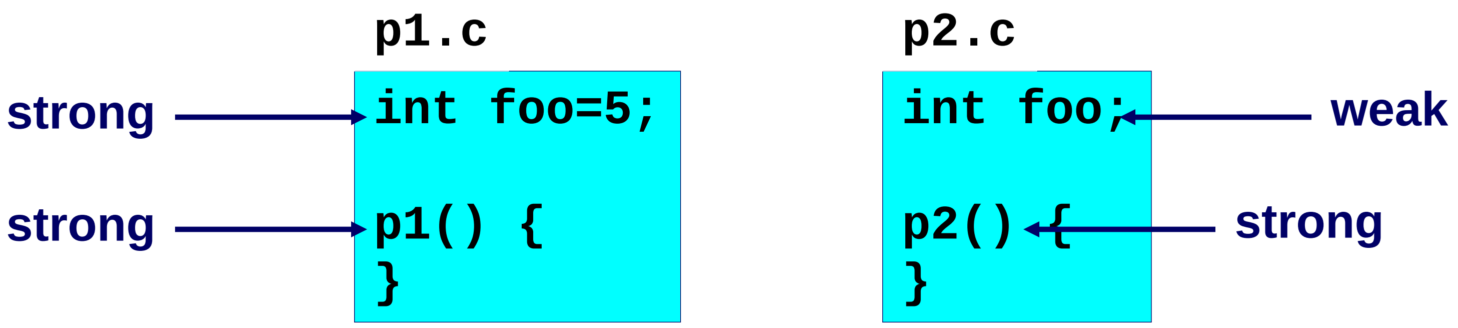
a.c

```
extern int e;

int *ep=&e;
int x=15;
int y;

int a() {
  return *ep+x+y;
}
```

Strong and Weak Symbols

Program symbols are either strong or weak

- *strong*: procedures and initialized globals
- *weak*: uninitialized globals



p1.c

```
int foo=5;

p1() {
}
```

p2.c

```
int foo;

p2() {
}
```

Linker’s Symbol Rules

Rule 1. A strong symbol can only appear once.

Rule 2. A weak symbol can be overridden by a strong symbol of the same name.

- references to the weak symbol resolve to the strong symbol.

Rule 3. If there are multiple weak symbols, the linker can pick an arbitrary one.

Linker Puzzles

```
int x;
p1() {}
```

```
p1() {}
```

Link time error: two strong symbols (p1)

```
int x;
p1() {}
```

```
int x;
p2() {}
```

References to x will refer to the same uninitialized int. Is this what you really want?

```
int x;
int y;
p1() {}
```

```
double x;
p2() {}
```

Writes to x in p2 might overwrite y!
Evil!

```
int x=7;
int y=5;
p1() {}
```

```
double x;
p2() {}
```

Writes to x in p2 will overwrite y!
Nasty!

```
int x=7;
p1() {}
```

```
int x;
p2() {}
```

References to x will refer to the same initialized variable.

Nightmare scenario: two identical weak structs, compiled by different compilers with different alignment rules.

Packaging Commonly Used Functions

How to package functions commonly used by programmers?

- Math, I/O, memory management, string manipulation, etc.

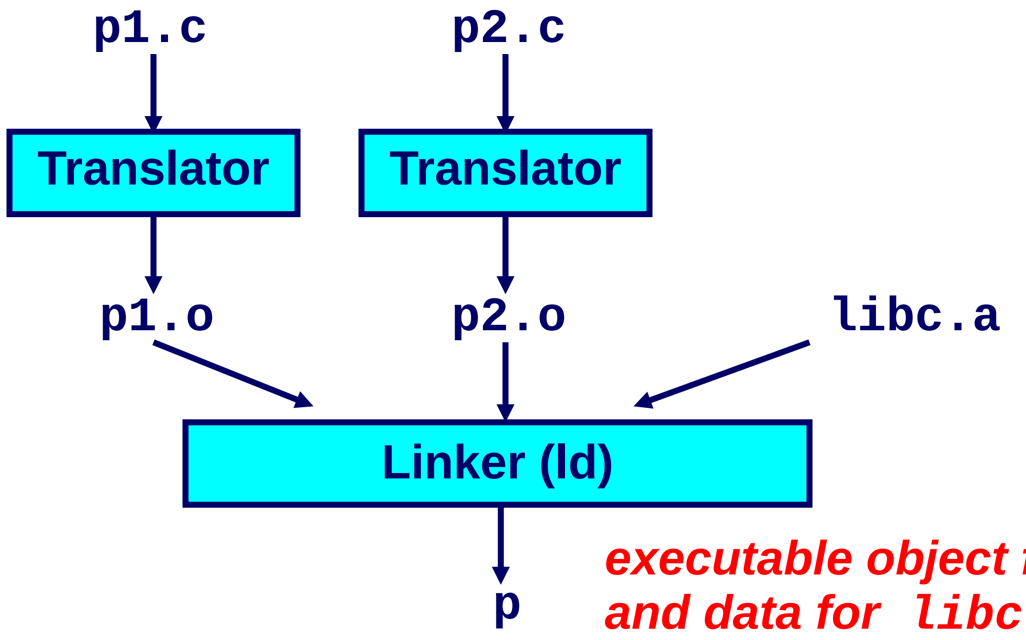
Awkward, given the linker framework so far:

- Option 1: Put all functions in 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

Solution: *static libraries* (.a archive files)

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

Static Libraries (archives)



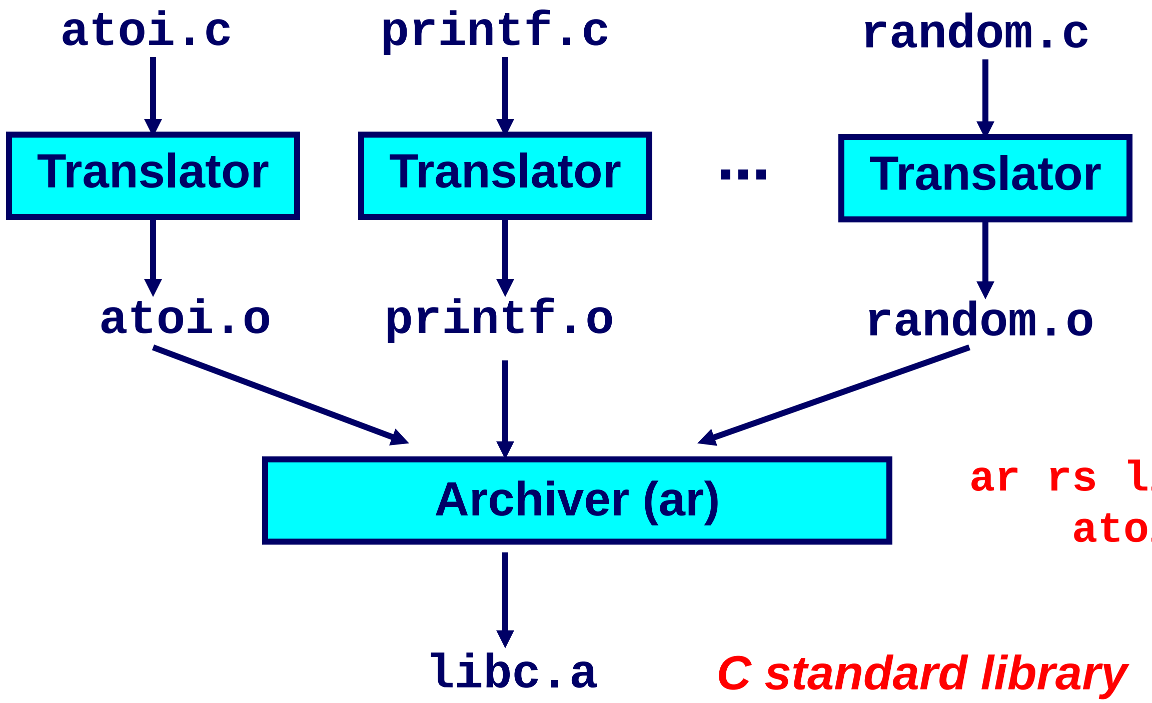
static library (archive) of relocatable object files concatenated into one file.

executable object file (only contains code and data for `libc` functions that are called from `p1.c` and `p2.c`)

Further improves modularity and efficiency by packaging commonly used functions [e.g., C standard library (`libc`), math library (`libm`)]

Linker selectively includes only the `.o` files in the archive that are actually needed by the program.

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)

- 2.4 MB archive of 1393 object files on mercury.cs.uml.edu.
- I/O, memory allocation, signal handling, string handling, data and time, random numbers, integer math

```
$ ar –t /usr/lib/i686-redhat-linux5E/lib/libc.a | wc –l
```

results: 1393

`libm.a` (the C math library)

- 450 KB 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
fstab.o
…
```

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

Using Static Libraries

Linker's algorithm for resolving external references:

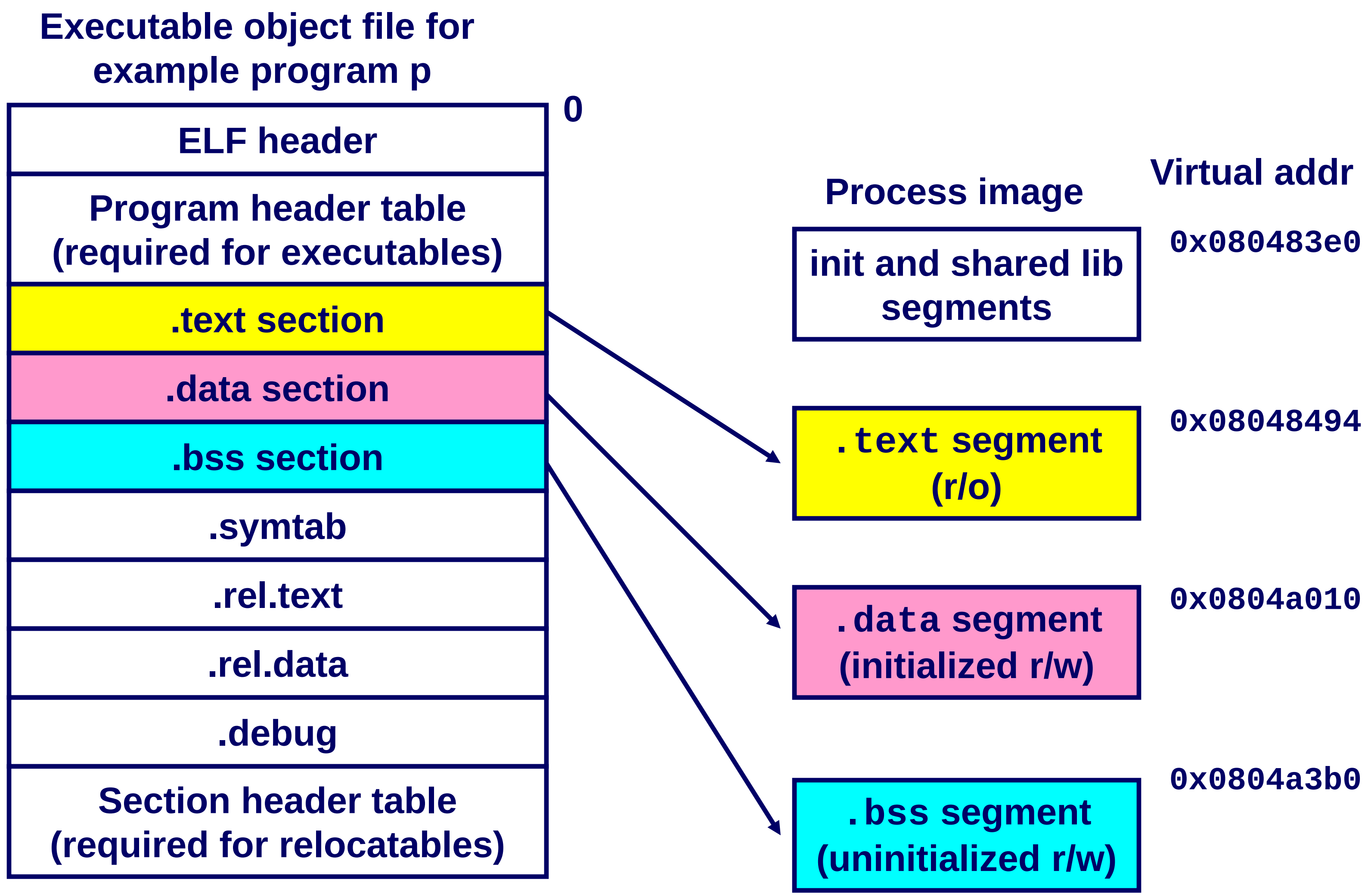
- Scan .o files and .a files in the command line order.
- During the scan, keep a list of the current unresolved references.
- As each new .o or .a file obj is encountered, try to resolve each unresolved reference in the list against the symbols in obj.
- If any entries in the unresolved list at end of scan, then error.

Problem:

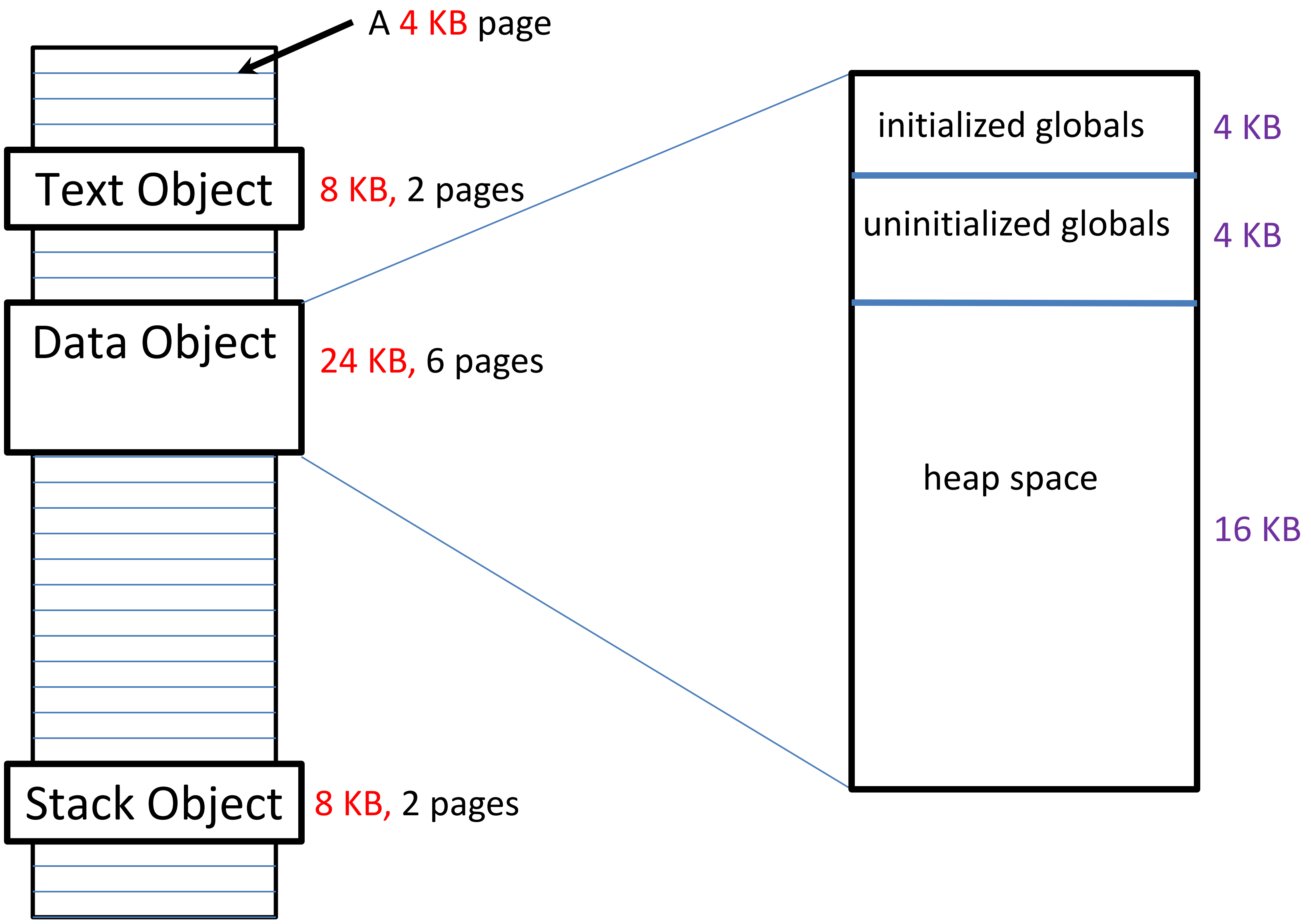
- Command line order matters!
- Moral: put libraries at the end of the command line.

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

Loading Executable Binaries


Executable object file for example program p
0
ELF header
Program header table (required for executables)
.text section
.data section
.bss section
.symtab
.rel.text
.rel.data
.debug
Section header table (required for relocatables)
Process image
Virtual addr
init and shared lib segments
0x080483e0
.text segment (r/o)
0x08048494
.data segment (initialized r/w)
0x0804a010
.bss segment (uninitialized r/w)
0x0804a3b0

Variably sized byte regions overlaid on a fixed page size object



Shared Libraries

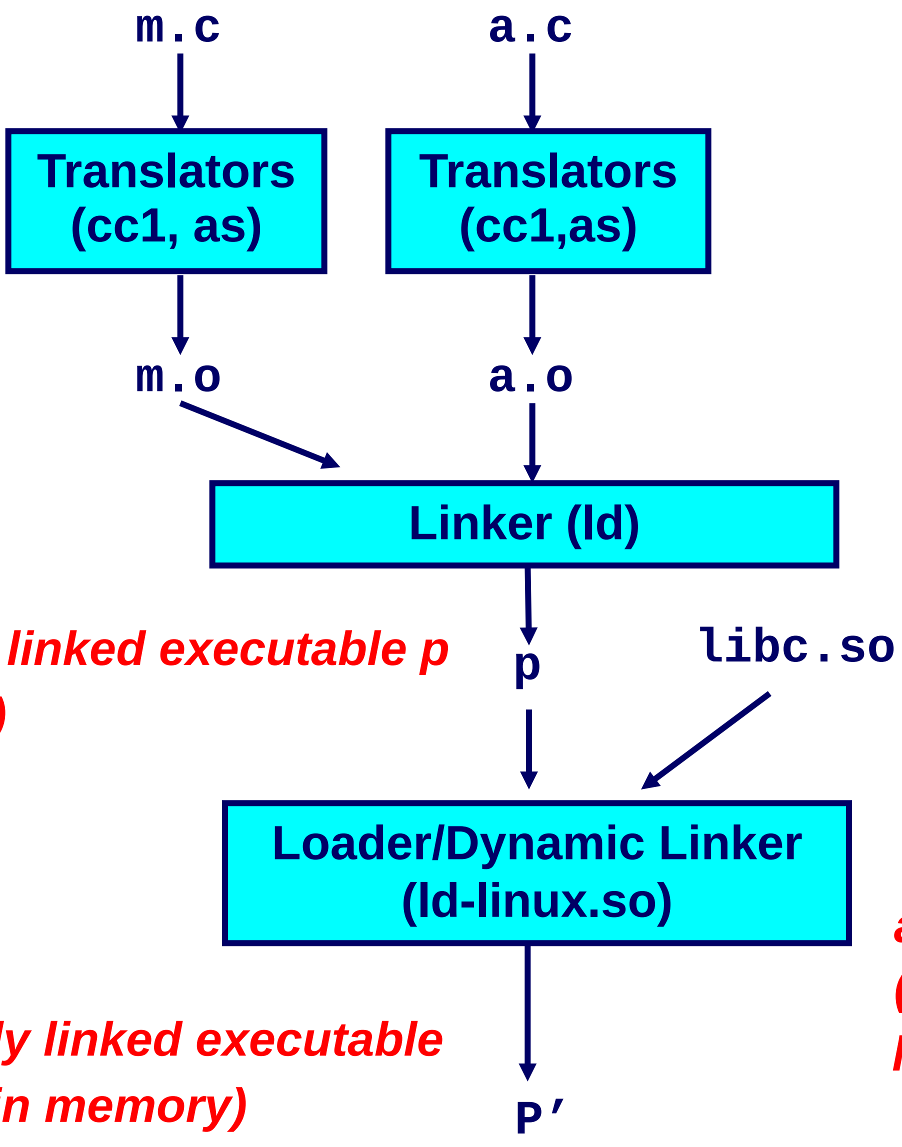
Static libraries have the following disadvantages:

- Potential for duplicating lots of common code in the executable files on a filesystem.
  - e.g., every C program needs the standard C library
- Potential for duplicating lots of code in the virtual memory space of many processes.
- Minor bug fixes of system libraries require each application to explicitly relink

Solution:

- *Shared libraries* (dynamic link libraries, DLLs) whose members are dynamically loaded into memory and linked into an application at run-time.
  - Dynamic linking can occur when executable is first loaded and run.
    - » Common case for Linux, handled automatically by `ld-linux.so`.
  - Dynamic linking can also occur after program has begun.
    - » In Linux, this is done explicitly by user with `dlopen()`.
    - » Basis for High-Performance Web Servers.
  - Shared library routines can be shared by multiple processes.

Dynamically Linked Shared Libraries



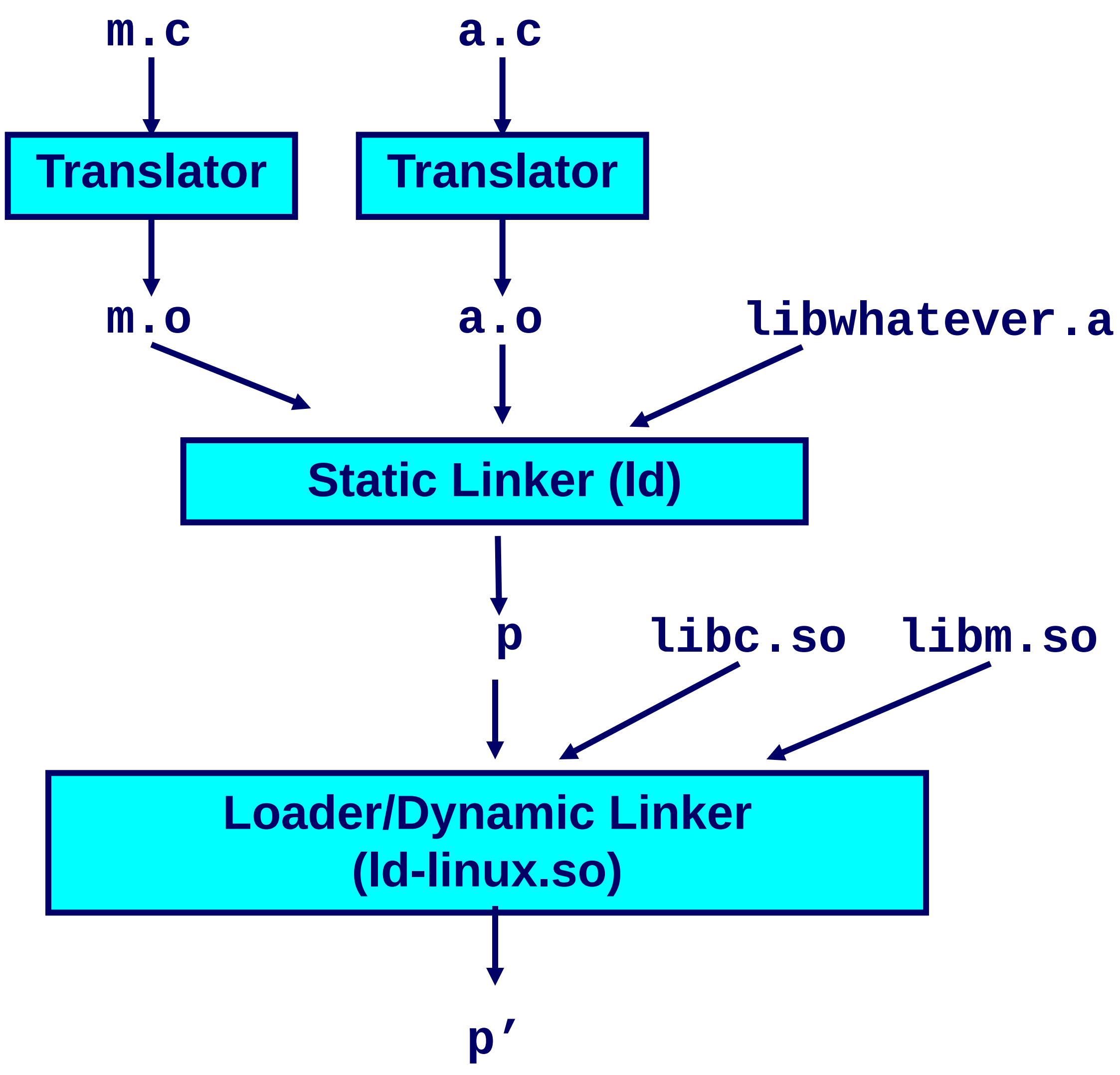
Partially linked executable p (on disk)

Shared library of dynamically relocatable object files

`libc.so` functions called by `m.c` and `a.c` are loaded, linked, and (potentially) shared among processes.

Fully linked executable p' (in memory)

The Complete Picture


m.c
a.c
Translator
Translator
m.o
a.o
libwhatever.a
Static Linker (ld)
p
libc.so
libm.so
Loader/Dynamic Linker (ld-linux.so)
p’