

Operating Systems: Memory Management

Alistair Rendell

(minor updates by Peter Strazdins)

Computer Systems: A Programmers Perspective

Randal E Bryant and David O'Hallaron

Structured Computer Organization

Andrew Tanenbaum

Memory Management: Objectives

- 1. Relocation
 - load to different memory location
 - dynamically move processes in memory
- 2. Protection
 - write/read(?) memory of other processes
- 3. Sharing
 - sometimes desirable
- 4. Logical Organisation
 - address ranges of 0-N for each process/source file(?)
- 5. Physical Organisation
 - multilevel storage (memory $\approx 70\text{ns}$, disk $\approx 1\text{ms}$)

Address Map

- A conceptually simple and elegant device to achieve all previous objectives
- Distinguishes program address from memory locations into which they are mapped
 - Address space (or namespace)
 - Memory space

$$f: N \rightarrow M$$

The programmer “sees”, and programs for, a **virtual memory** whose characteristics differ from those of the real memory

Relocation and Protection

- Base Register (B)
 - Constant offset to add to memory references

$$f(a) = B + a$$

- Limit Register (amax)
 - Maximum valid offset

if (a < 0) memory violation

a' = B + a

if (a' > amax) memory violation

a is valid address

- Mapping is linear and size \leq physical memory size
 - objectives 3 and 5 not met

Paging

- Provides virtual memory that is greater than physical memory
- Program address is divided into two parts:
 - high order address bits give page number
 - low order address gives page offset

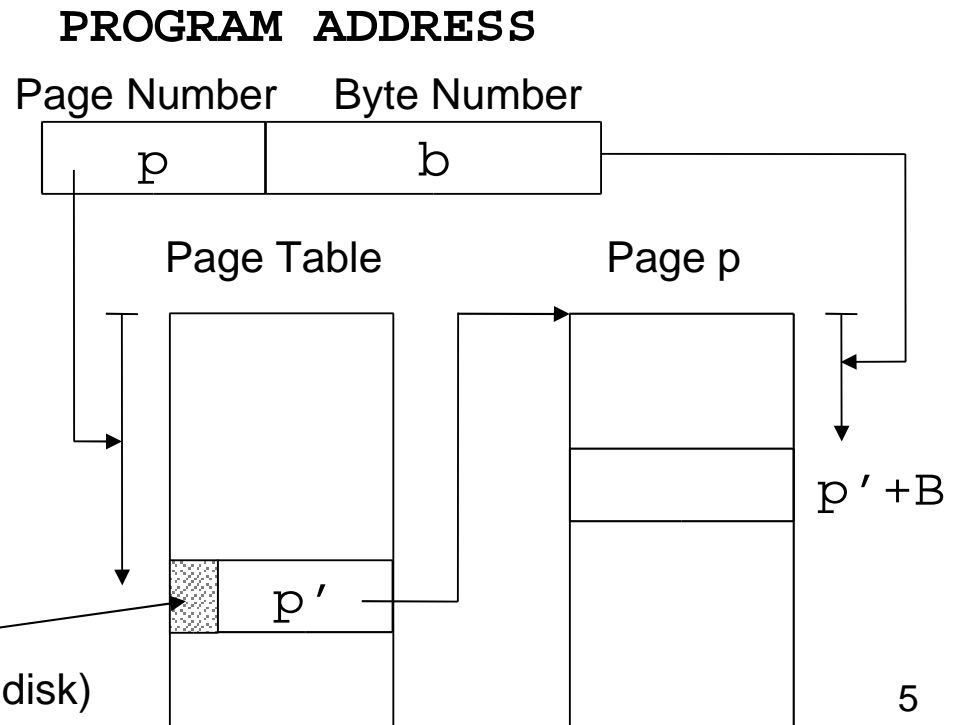
$$f(a) = f(p, b) = p' + b$$

p = integral part of (a/Z)

b = remainder of (a/Z)

(Z = page size)

(eg. located in memory or on disk)



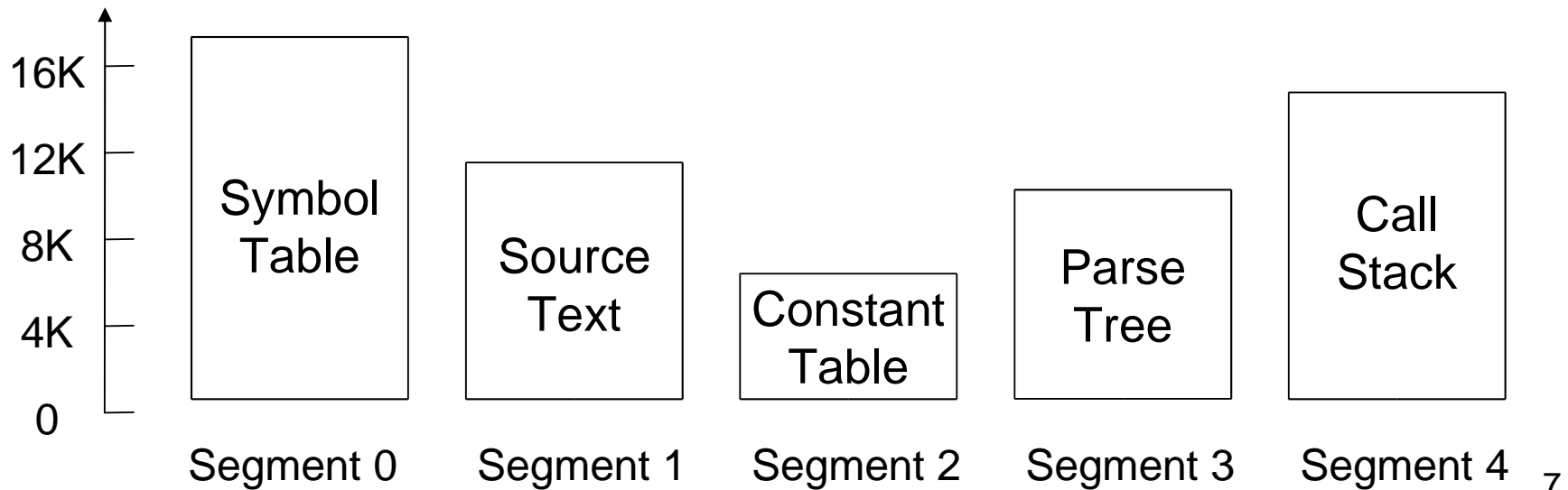
Paging Issues

- Page size
- Page replacement policy
 - working set
 - thrashing

All what Peter talked to you about!!

Segmentation

- Having two or more separate virtual address spaces desirable
- Eg. data structures/areas during compilation
 - The symbol table, containing names and attributes of variables
 - The source text
 - Integer and floating point constants
 - The parse tree containing syntactic analysis of the program
 - The stack used for procedure calls by compiler



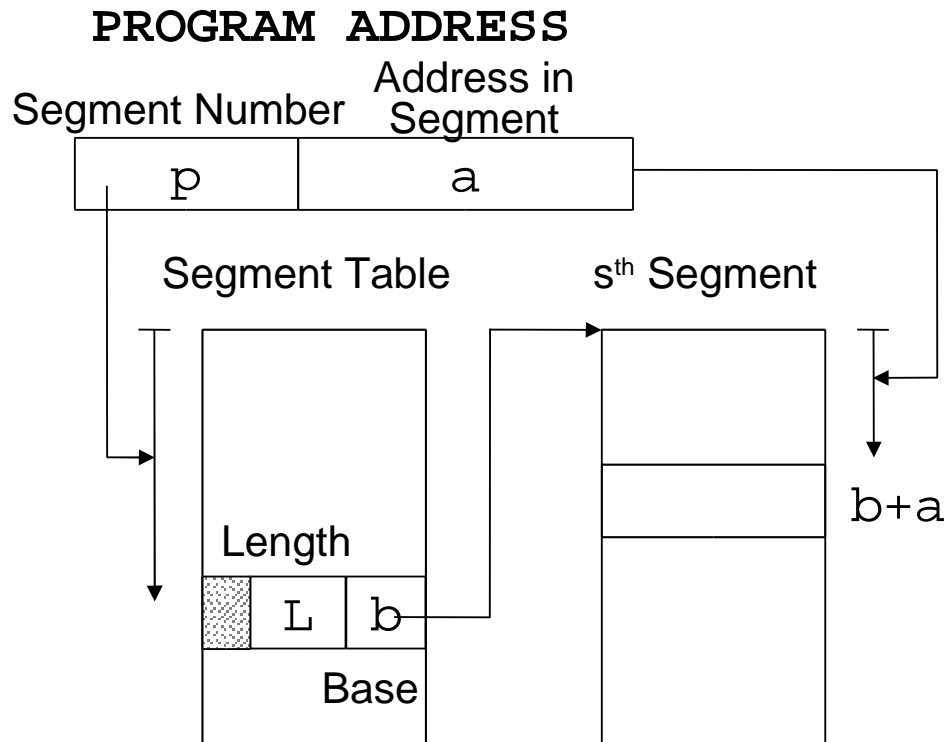
Segmentation

Extract program address (s, a)

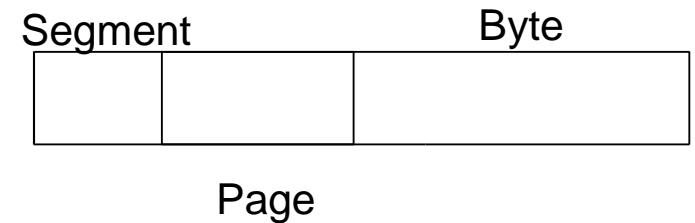
Use s to index segment table

if $(a < 0 \ || \ a > L)$ memory violation

$(b+a)$ is required memory location



Combining Segmentation and Paging



Paging and Segmentation

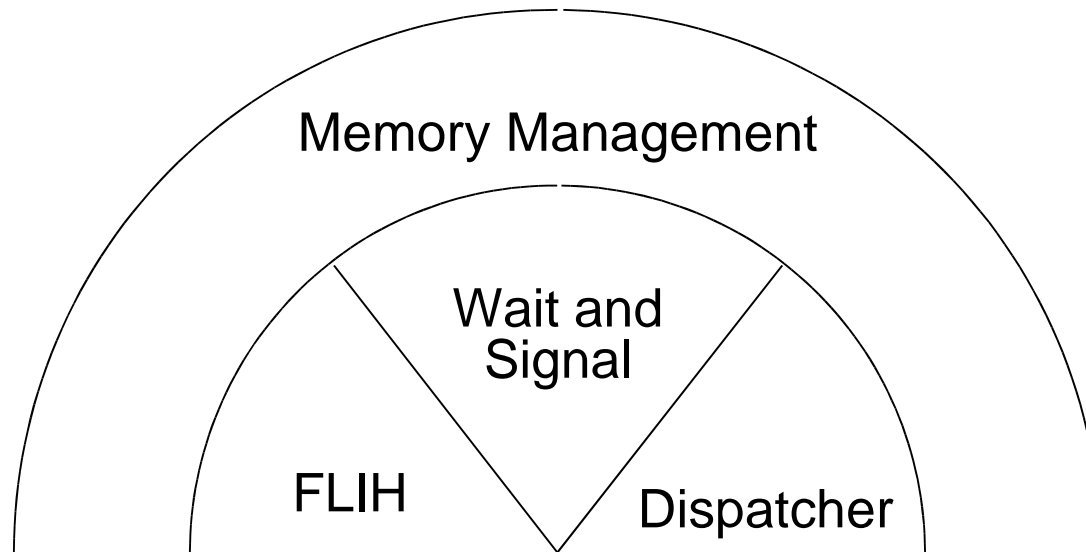
| Consideration | Paging | Segmentation |
|---------------------------------------|----------------------------|------------------------------------|
| Programmer aware | No | Yes |
| Number linear address spaces | 1 | Many |
| Virtual address space exceed physical | Yes | Yes |
| Variable sized tables easily handled | No | Yes |
| Why invented | To simulate large memories | To provide multiple address spaces |

See Tanenbaum Chapter 6

Real Machines

- UltraSparc II
 - Demand paging
 - Page size 8, 64, 512 and 4096KB
- Pentium II
 - Demand paging
 - Segmentation
 - Segmentation with paging

Operating System Layers



Linking and Loading!

Why

- Help you to build large programs
 - Linker errors
- Avoid dangerous program errors
 - Multiply defined global variables
- Scoping rules
 - Global/local, and static variables
- Systems concepts
 - Virtual memory, paging, memory mapping
- Explicit shared libraries
 - Dynamic linking

Compilation

```
/* mswap.c */  
void swap();
```

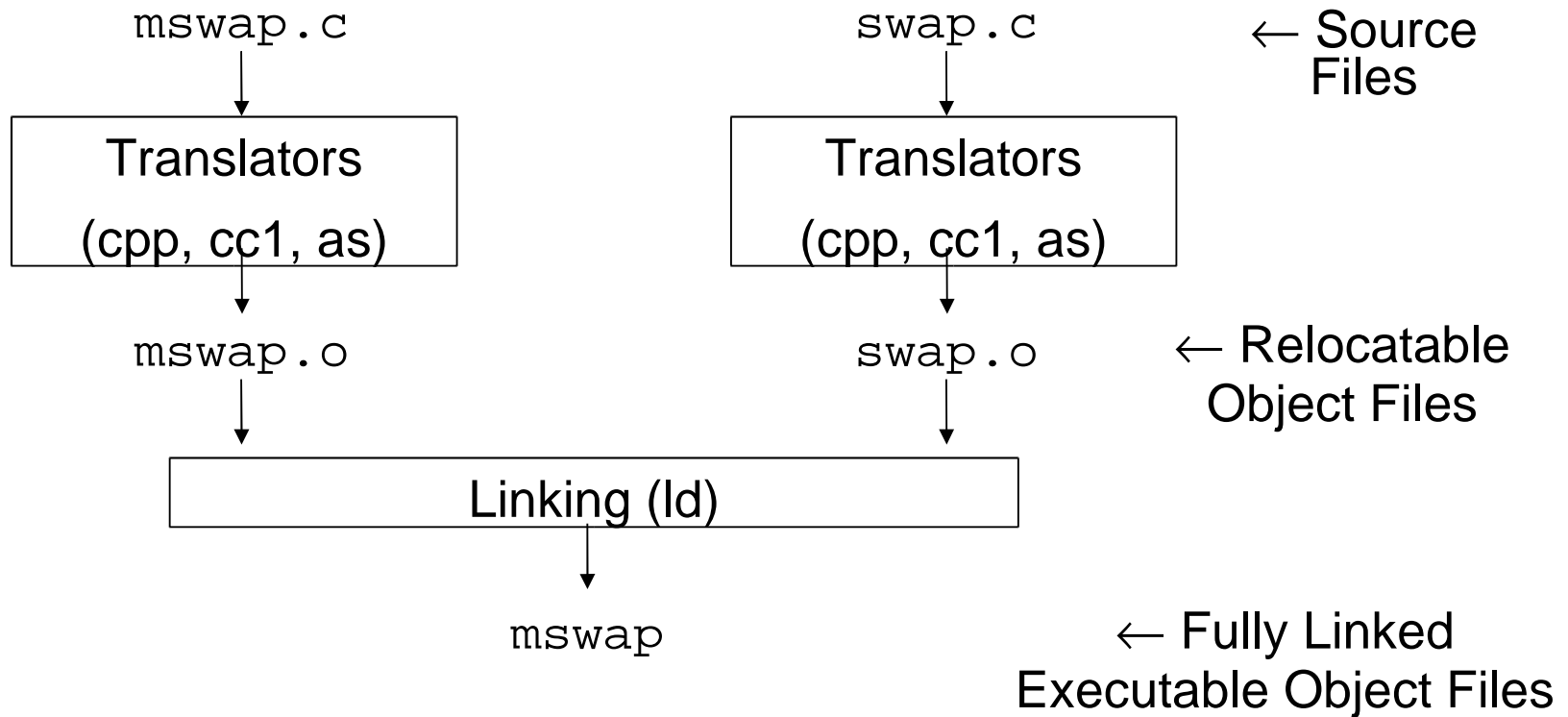
```
int buf[2] = {1,2};
```

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

```
/* swap.c */  
extern int buf[];  
  
int *bufp0 = &buf[0];  
int *bufp1;  
  
void swap()  
{  
    int temp;  
    bufp1 = &buf[1];  
    temp = *bufp0;  
    *bufp0=*bufp1;  
    *bufp1 = temp;  
}
```

```
me@partch> gcc -v -O2 -o mswap mswap.c swap.c
```

Static Linking



- Symbol resolution
 - Associate each symbol reference with exactly one definition
- Relocation
 - Add offsets to compiler generated addresses that start from zero

Object Files

- Relocatable object file
 - binary code and data in a form that can be combined with other relocatable object files at compile time to give executable object file
- Executable object file
 - binary code and data in a form that can be copied directly into memory and executed
- Shared object files
 - special type of relocatable object file that can be loaded into memory and linked dynamically, at load or run time

Modern Unix systems use Executable and Linkable Format (ELF) object files

ELF Relocatable Object

| | | |
|----------|--------------------------------|--|
| Sections | ELF Header | Machine code |
| | .text | Read only data (printf strings) |
| | .rodata | Initialized global data |
| | .data | Uninitialized global data (no space) |
| | .bss | Symbol table |
| | .symtab | Locations modified by linker (ext fun) |
| | .rel.text | Relocation info for global variables |
| | .rel.data | Debug symbol table (only -g) |
| | .debug | Source line numbers to machine code |
| | .line | String table from .symtab and .debug |
| | .strtab | Section header table |
| | Describes object file sections | Section header table |

Symbols and Symbol Tables

- Global symbols
 - defined by a relocatable object module and can be referenced by other modules
 - C global variables and functions defined without the **static** attribute
- Local symbols
 - defined and references exclusively by a given relocatable object module
 - C functions and global variables with **static** attribute

(Static attribute provides means to “hide” variables and functions)

GNU readelf Tool

```
u8914893@varese:~/.../02progs> readelf -s mswap.o
```

```
Symbol table '.symtab' contains 10 entries:
```

| Num: | Value | Size | Type | Bind | Vis | Ndx | Name |
|------|----------|------|---------|--------|---------|-----|----------------|
| 0: | 00000000 | 0 | NOTYPE | LOCAL | DEFAULT | UND | |
| 1: | 00000000 | 0 | FILE | LOCAL | DEFAULT | ABS | mswap.c |
| 2: | 00000000 | 0 | SECTION | LOCAL | DEFAULT | 3 | |
| 3: | 00000000 | 0 | NOTYPE | LOCAL | DEFAULT | 2 | gcc2_compiled. |
| 4: | 00000000 | 0 | SECTION | LOCAL | DEFAULT | 2 | |
| 5: | 00000000 | 0 | SECTION | LOCAL | DEFAULT | 4 | |
| 6: | 00000000 | 0 | NOTYPE | GLOBAL | DEFAULT | UND | printf |
| 7: | 00000000 | 8 | OBJECT | GLOBAL | DEFAULT | 3 | buf |
| 8: | 00000000 | 120 | FUNC | GLOBAL | DEFAULT | 2 | main |
| 9: | 00000000 | 0 | NOTYPE | GLOBAL | DEFAULT | UND | swap |

Ndx = which ELF section

3 ≡ .data, 2 ≡ .text

Symbol Resolution

- Linker resolves symbol references by associating each reference with exactly one symbol definition from the symbol tables of the relocatable object files

- Failure to resolve these gives cryptic messages like

```
/tmp/ccKE2HoZ.o(.text+0x2f): In function `main':  
: undefined reference to `swap'
```

- Compiler exports global symbols to assembler as either strong or weak
 - Strong: functions and initialized global variables
 - Weak: uninitialized global variables
- Rules
 - Multiple strong symbols not allowed
 - A strong symbol is taken in preference to a weak one
 - If multiple weak symbols, choose any

Aside

- Function overloading is achieved by creating unique symbol names to be used with different parameter lists
 - This encoding process is called *mangling*

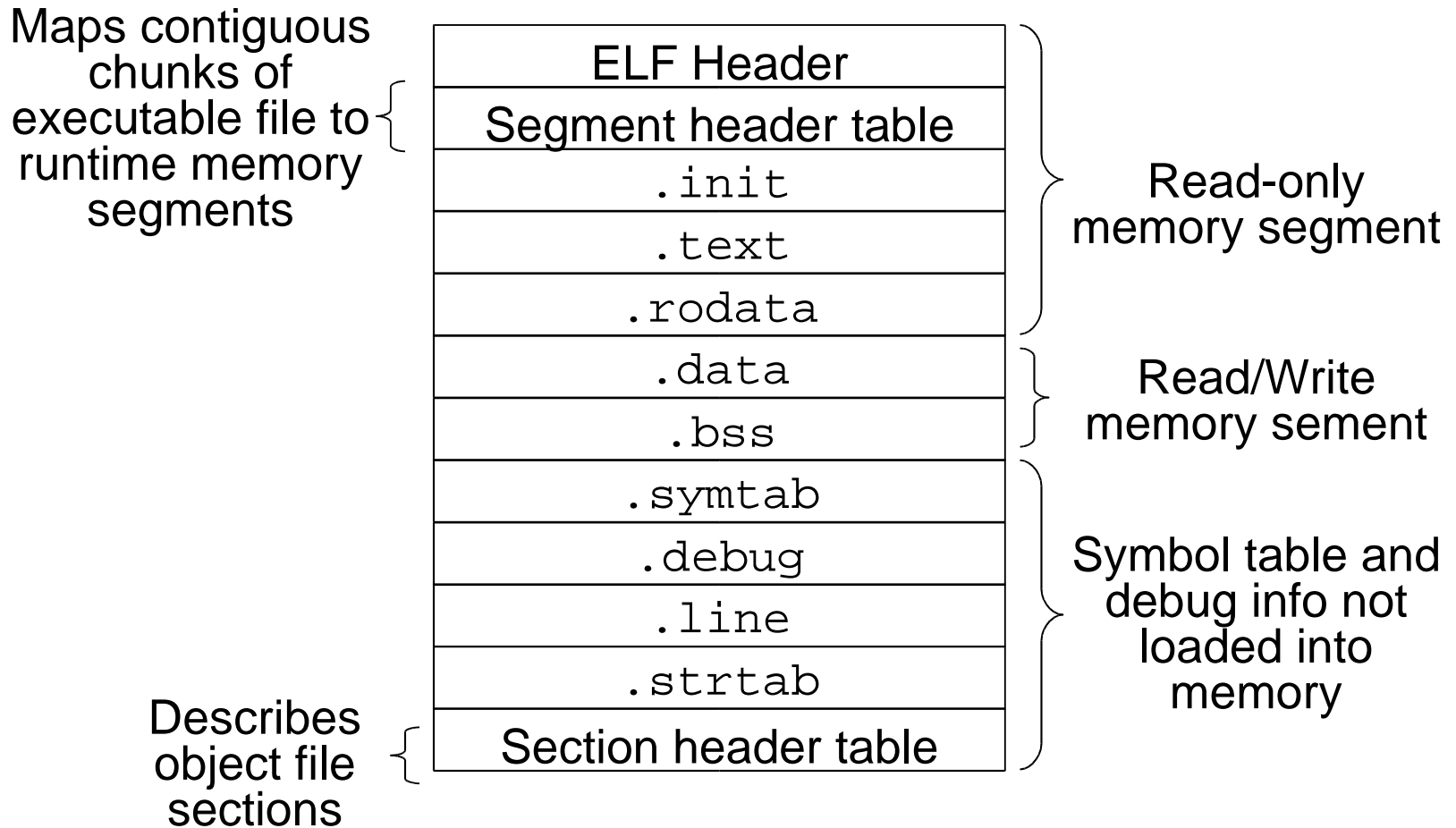
Potential Problems

```
#include <stdio.h>
void f(void);
int x = 12345;
int y = 67890;
int main()
{
    printf (" Before %d %d\n",x,y);
    f();
    printf (" After  %d %d\n",x,y);
    return 0;
}

double x;
void f()
{
    x = 0.0;
}
```

```
u8914893@varese:~/.../02progs> gcc mproblem.c problem.c
/usr/bin/ld: Warning: alignment 4 of symbol `x' in /
tmp/ccANhVvn.o is smaller than 8 in /tmp/ccENq65w.o
```

ELF Executable Objects



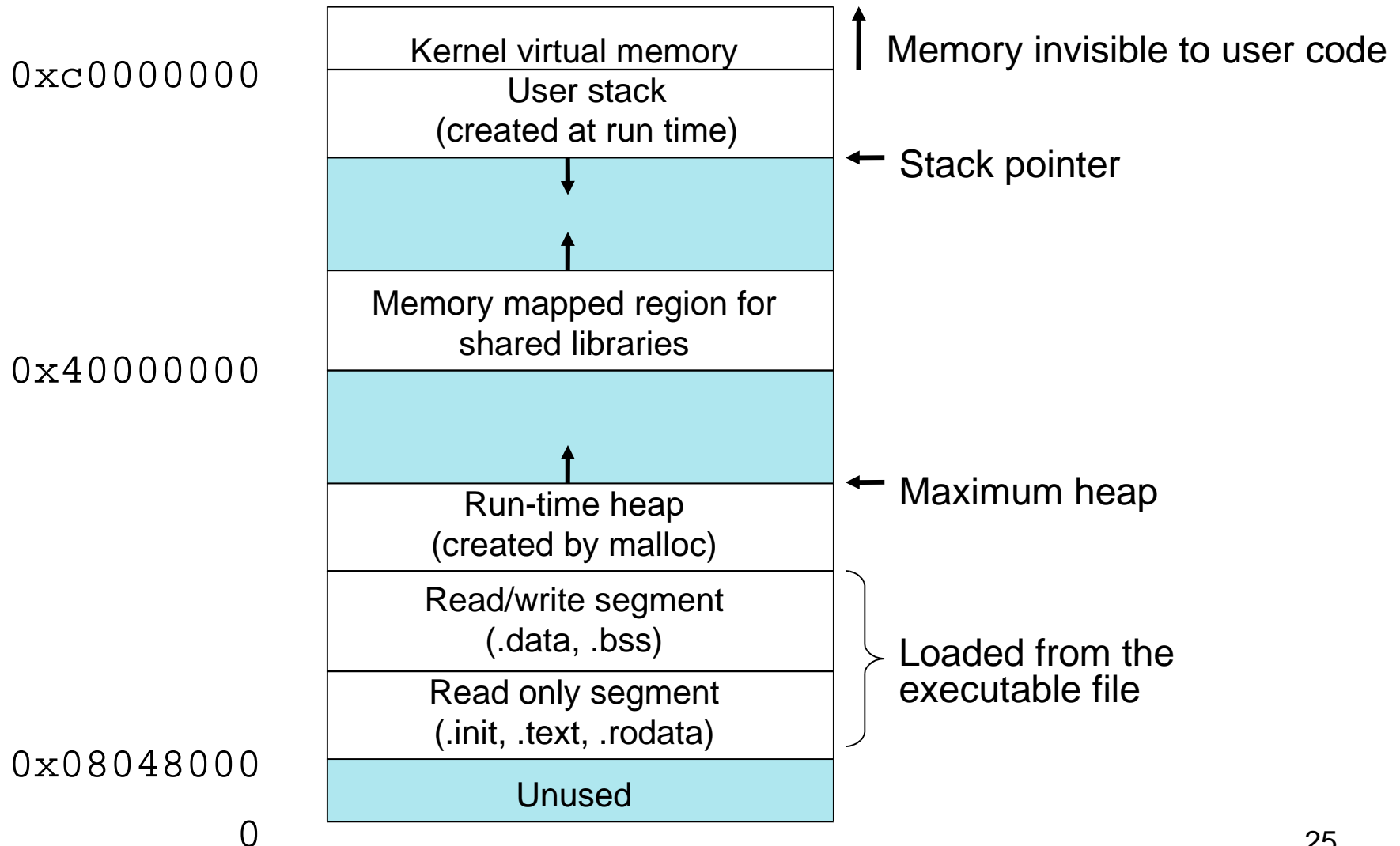
- Header includes program entry point
 - `init()` is a small function called by program's initialization code

The Loader

- Copies chunks of the executable into the code and data segments (next figure)
- Jumps to address of the `_start` symbol (for C programs defined in `crt1.o`)

```
0x080480c0 <_start>          /* entry point in .text */
    call __libc_init_first /* startup code in .text */
    call _init              /* startup code in .init */
    call atexit            /* routines to call on exit */
    call main              /* application main routine */
    call _exit            /* returns control to OS */
```

Loading Executable Object Files (Linux)



Manipulating Object Files

| | |
|----------------------|---|
| <code>ar</code> | creates static libraries |
| <code>strings</code> | lists all printable strings in an object file |
| <code>strip</code> | deletes symbol table information from object file |
| <code>nm</code> | lists the symbols defined in the symbol table of an object file |
| <code>size</code> | lists the names and sizes of the sections in an object file |
| <code>readelf</code> | displays complete structure of an object file |
| <code>objdump</code> | the mother of all binary tools!! |
| <code>ldd</code> | lists shared libraries than an executable needs at run time |

