CS33 DISCUSSION 7

LINKERS and CACHE LAB

Brandon Wu

11.21.14

Lab 3: Associative Cache

- Basic Cache Parameters: S, B, E, M
  - S = number of sets
  - E = associativity
  - B = block size (# of elements != # of bytes...in this example)
  - M = size of memory space

Lab 3: Associative Cache

- Extracting Tag index offset
  - m = # of addr bits = $\log_2 M$
  - s = # of index bits = $\log_2 S$
  - b = # of offset bits = $\log_2 B$
  - T = # tag bits = $m - s - b$
- Total cache size:
  - C = (# of sets) * (associativity) * (block size)
  - $C = S*B*E$
- $E = 1 \rightarrow$ Direct Mapped cache
- $S = 1 \rightarrow$ Fully Associative cache

Lab 3: Associative cache

- For this lab, assume our Memory is “4B” or “int addressable”
  - e.g. instead of an array of bytes, we have
  - int memory[M];

Initcache( )

- For each cache line, alloc a block
  - Set valid and dirty to 0
- Allocate your memory
- Set s, b and m based on Cache parameters
- Accessing cache example:

  cache[i][e]

  - Accesses i'th set, and e'th cache line in the set
  - i = [0, S-1], e = [0, E-1]

Readwritecache

void readwritecache(int readwrite, int a, int *value, int *hitmiss, int voice)

- Readwrite = 1 for read, readwrite = 0 for write
- If write, insert value into appropriate position of cache block (based on its offset)
- If read, read int from a cacheline based on its tag,index and block offset
- If evicting a block from cache and dirty = 1, write entire block back to memory

Submission Rules

- Please submit a .c file only (sorry, no more c++)
- Please test it on seasnet with this command:

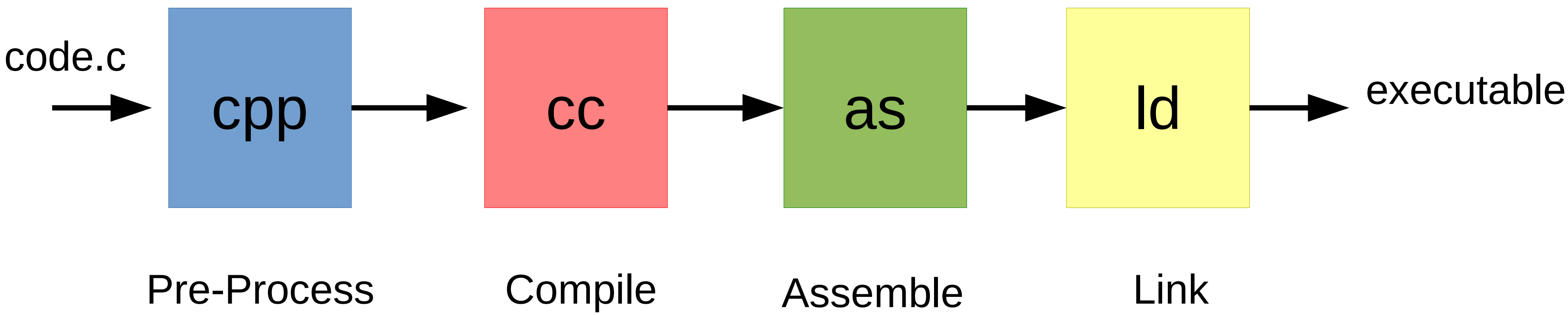
  `gcc lab3.c -lm`

- We will most likely allow `std=c99` flag as well
- Please do not remove the `printf` statements from `readwritecache`. We use these to grade!

Linkers (chapter 7)

Program Translation Process

- How to build an executable from plaintext file?



Pre-Processor

- Inserts “symbols” into our module according to “#” directives
  - #define MAX_INT 1<<31
  - #include “globals.h”
- Copy and paste symbols in .h files

Compiler

- Translate text program into universal assembly language
  - Does not resolve undefined symbols
- Exports symbols to assembler

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

int main( ) {
    int x = foo(3,4);
    return 0;
}
```

Assembler

- Builds **Relocatable object file**
- Instruction & data addresses are not “real”
  - Relative labels
- Builds symbol table

Linker

- Aggregates relocatable object files
- Integrates shared libraries, dynamic linked libraries
- Symbol resolution
- Relocates memory addresses of code blocks
- Produces an executable

BE CAREFUL ABOUT LINKING!

foo.c

```c
int bar(int a, int b);

int main( ) {
    int x = bar(0x8000,0x7fff);
    return 0;
}
```

bar.c

```c
short bar(short c, short d) {
    return (c > d) ? c : d;
}
```

foo.c

```c
int bar(int a, int b);

int main( ) {
    int x = bar(0x8000,0x7fff);
    return 0;
}
```

This actually compiles and links!

bar.c

```c
short bar(short c, short d) {
    return (c > d) ? c : d;
}
```

foo.c

```c
int bar(int a, int b);

int main( ) {
    int x = bar(0x8000,0x7fff);
    return 0;
}
```

This actually compiles and links!

But with some funny behavior...

bar.c

```c
short bar(short c, short d) {
    return (c > d) ? c : d;
}
```

Some Terminolgy

- **Reolocatable object file**: code and data, location and memory unresolved, code segments not associated with absolute memory address
- **Executable object file**: binary that can be copied into memory and run directly
- **Shared object file:** can be loaded into mem and linked dynamically at run time

Declaration vs Definition

- Q: What is a Declaration?

Declaration vs Definition

- Q: What is a Declaration?
  - A statement that a variable/function exists somewhere in the program
  - e.g. `int bar(int a, int b);`

Declaration vs Definition

- Q: What is a Declaration?
  - A statement that a variable/function exists somewhere in the program
  - e.g. `int bar(int a, int b);`
- Can I have multiple declarations?

Declaration vs Definition

- Q: What is a Declaration?
  - A statement that a variable/function exists somewhere in the program
  - e.g. `int bar(int a, int b);`
- Can I have multiple declarations?
  - Sure. A declaration does not allocate/take up memory

Declaration vs Definition

- Q: What about a Definition?

Declaration vs Definition

- Q: What about a Definition?
  - Defines what the function does, or the value of the variable
  - Allocates memory for function/variable
  - Multiple definitions → Linker Error!

Declaration vs Definition

- Q: What about a Definition?
  - Defines what the function does, or the value of the variable
  - Allocates memory for function/variable
  - Multiple definitions → Linker Error!
- Is it possible to declare a variable without defining it?

Declaration vs Definition

- Q: What about a Definition?
  - Defines what the function does, or the value of the variable
  - Allocates memory for function/variable
  - Multiple definitions → Linker Error!
- Is it possible to declare a variable without defining it?
  - Use extern keyword → specifies external linkage
  - `extern int x;`

Global symbols

- **Global symbols**: defined in module *m* but referenced by other modules
  - See slide 11 example bar.c which defines bar( )
- Global symbols can reference other modules (**External symbols)**
  - Declare without definition
  - Defined by external linkage

    ```
    [extern] int bar(int a, int b);  //"extern" optional
    extern int x;
    ```

Local symbols

- **Local symbols:** defined and referenced only in module *m*

  ```
  static void hiddenFunction(int a, int b) { }
  static int secretKey;
  ```

- A way to “hide” data and code ~ OOP private members
- “static” keyword specifies internal linkage

Relocatable object file

- E.g “.o” files
- Compiled & assembled code and data
- Not yet executable

ELF Header
.text: machine code
.rodata: read only data
.data: initialized globals
.bss: unitialized globals (description)
.symtab: symbol table (globals and external function info)
.rel.text: relocation information for externals
.rel.data: relocation information for cross referenced data
.debug: -g symbols for gdb
.line: -g line numbers for gdb
.strtab: descriptive strings for .symtab
Section header table: which sections are in the table

A Simple Linking Example from Garrett's slides

```c
// Code for main.c
int buf[2] = {1, 2};

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

```c
// Code for swap.c
extern int buf[];

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

What are the symbols created?

A Simple Linking Example from Garrett's slides

```c
// Code for main.c
int buf[2] = {1, 2};

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

Global: buf, main
Local: none
External: swap

```c
// Code for swap.c
extern int buf[];

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

A Simple Linking Example from Garrett's slides

```c
// Code for main.c
int buf[2] = {1, 2};

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

Global: buf, main
Local: none
External: swap

```c
// Code for swap.c
extern int buf[];

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

Global: bufp0, swap
Local: bufp1
External: buf

Strong vs Weak Symbols

- **Strong symbol**: Global symbol that is defined and initialized
  - `int x = 0xabcdef;`
- **Weak symbol:** Global symbol that is uninitialized
  - `int x;    // defined but not initialized`
  - `extern int y; // neither defined nor initialized`

Let's play Does This Compile (and link)!

Does this compile?

```c
//main1.c
int main() {
    printf("Hello\n");
    return 0;
}
```

```c
//main2.c
int main() {
    printf("Goodbye\n");
    return 0;
}
```

gcc main1.c main2.c

Does this compile? (pg 665)

```c
// foo3.c
#include <stdio.h>
void f(void);
int x = 15213;
int main( ) {
    f( );
    printf("x = %d\n", x);
    return 0;
}
```

```c
// bar3.c
int x;
void f( ) {
    x = 15212;
}
```

```
>gcc foo3.c bar3.c
```

Does this compile? (pg 666)

```c
// foo5.c
#include <stdio.h>
void f(void);
int x = 15213;
int y = 15212;
int main( ) {
    f( );
    printf(“x = %x, y=%x\n”, x,y);
    return 0;
}
```

```c
// bar5.c
double x;
void f( ) {
    x = -0.0;
}
```

```
>gcc foo5.c bar5.c
```

Does this compile? (pg 666)

```c
// foo5.c
#include <stdio.h>
void f(void);
int x = 15213;
int y = 15212;
int main( ) {
    f( );
    printf("x = %x, y=%x\n", x,y);
    return 0;
}
```

```c
// bar5.c
double x;
void f( ) {
    x = -0.0;
}
```

```
>gcc foo5.c bar5.c
>./a.out
> x = 0x0
  y = 0x80000000
```

Does This compile? (7.9 in book)

```c
//foo6.c
void p2(void);
int main() {
    p2();
    return 0;
}
```

```c
//bar6.c
char main;
void p2( ) {
    printf("0x%x\n", main);
}
```

gcc foo6.c bar6.c

Does This compile? (7.9 in book)

```c
//foo6.c
void p2(void);
int main() {
    p2();
    return 0;
}
```

```c
//bar6.c
char main;
void p2( ) {
    printf("0x%x\n", main);
}
```

gcc foo6.c bar6.c

Actually, it does. And outputs 0x55

Linking is Weird...

- main is a global symbol in foo6.o symbol table
- During link, main code is relocated to some address (ptr to first instr)
- First instruction of main is a push %rbp → 0x55
- External references to main read 0x55 as value

```
//foo6.c
void p2(void);
int main() {
    p2();
    return 0;
}
```

So What is the Bigger Picture?

- Linking is weird
- Therefore we should avoid global definitions and external declarations where possible

Relocation

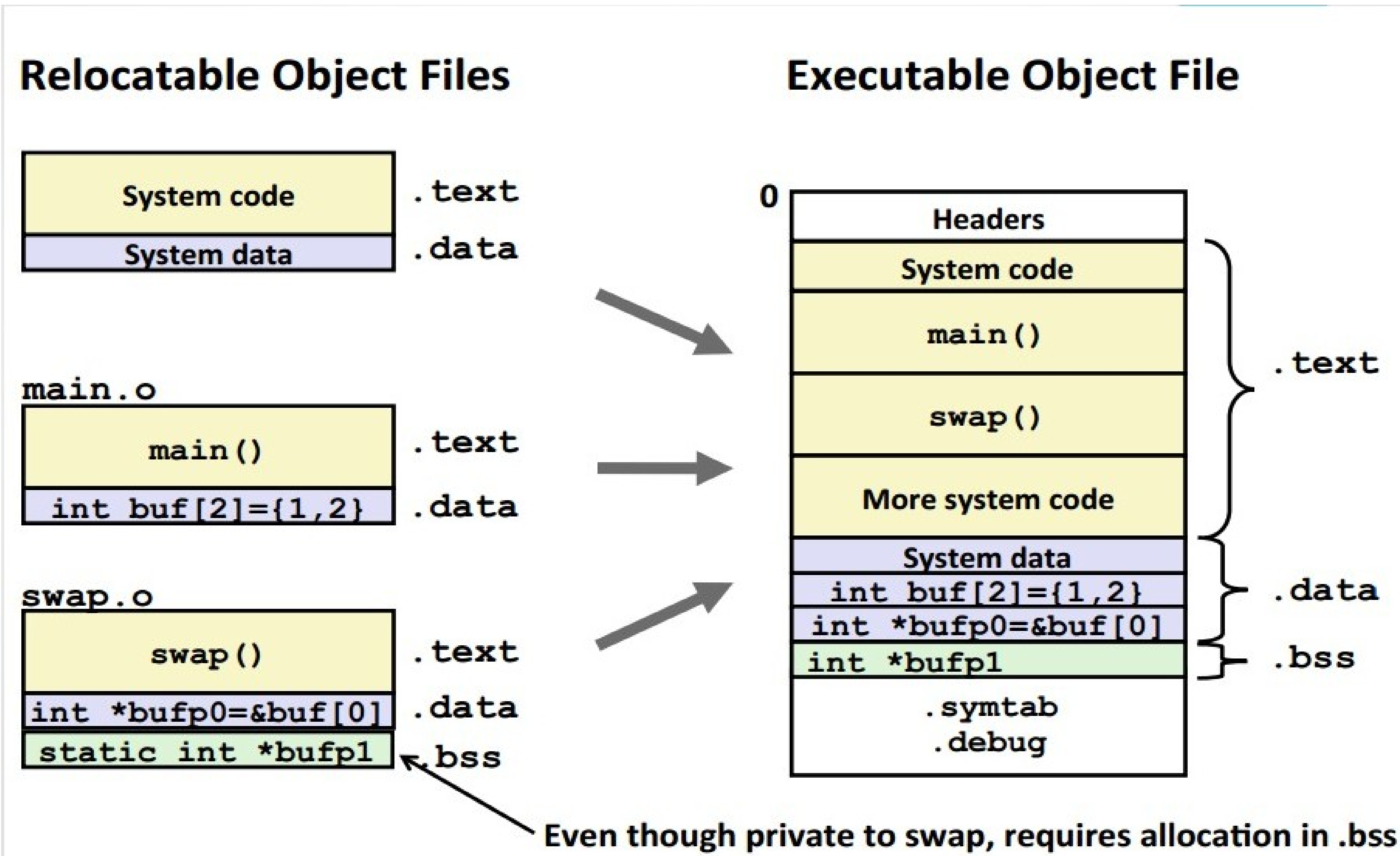
- Linker aggregates .data and .text from all relocatable objects and libraries
- Must relocate each modules .data and .text
- Within each module, all references are relative

e.g consider call instruction

`+0x6: e8` ff 00 00 00

- PC relative jump => jump to PC + jmp_amt

Diagram of Relocation (from Garrett's slides)



Relocation example from book (pg675)

Call swap:

6: e8 fc ff ff ff   call 7 <main+0x7>

relocation entry 7 = swap

- Current reference = +0xfffffffc = -4
- PC at call = <main> + 6 + 5
- Compute the PC relative jump amount after relocation

Relocation example from book (pg675)

- Jump amount = ADDR(swap) + ref - reffAddr
- In this case refAddr = main+0x7
- Lets say
  - ADDR(swap) = 0x80483c8
  - ADDR(main) = 0x80483b4
  - => RefAddr = 0x80483b4+0x7 = 0x80483bb
- Jump amount = 0x80483c8 + (-4) – 0x80483bb

  = 0x80483c8 – (4+0x80483bb)

  = 0x9

Next Instruction After call

Relocation example from book (pg675)

- So in gdb, we can re-examine the instruction after relocation:

  80483ba: e8 09 00 00 00 call 80483c8 <swap>

- So during call instruction, PC = 0x80483ba + 0x5 = 0x80483bf
- A PC relative jump => PC = PC + 0x9

  = 0x80483bf + 0x9

  = 0x80483c8

  - The first instruction of swap!

Static Libraries

- We want to make use of reusable common functions
  - printf, atoi, rand
- We don't want to link one large executable each time we use a single library function
  - libc.o would be massive
- We don't want to explicitly link each module that we use
- So lets use the idea of a **static library**

Static Libraries

- An archive stores a list of relocatable object files corresponding to library modules
  - printf.o, atoi.o, etc
- Linker only copies modules for modules referenced by the relocatable objects

Static Libraries: How they work

- Linker reads the input object files first
  - Builds symbol table, keeps track of unresolved symbols
    - e.g. if I make a call to printf
- Linker makes several passes across archive to match modules with unresolved symbols
- If no unresolved symbols at the end, build executable
- This is done at **Link Time**

Problems with Static Libraries

1. Linking library functions done statically (at link time)
   - If changes made to library, need to build a new executable
2. Static library modules are literally copied into code segment of executable
   - If I have 100 processes that all use the same 10 library invocations, I have 100 copies of these modules in memory at once
3. Library modules are linked, but may never be invoked at run-time

Dynamic (Shared) Libraries

- Microsoft DLL's, Unix “.so”
- Does not fully link the objects during linking phase
  - Partially links references to libraries
- Linking done at run-time of the program
- All running processes can share a single copy of a shared library module

Dynamic Linking: How it Works

- During linking phase, no code from shared modules are copied into executable
  - Only “pointers” to the modules
- Upon execution, program loader runs **dynamic linker**
- Dynamic linker copies executable into memory and shared object into a <u>shared</u> memory segment
- All calls to shared modules are references that “point” to shared memory location