Memory management background

ECE 595

Feb 6

Y. Charlie Hu

First, a Systems Review

- From writing a C code
- .
- .
- .
- .
- .
- till a.out is running in the CPU

What Is "Systems"

- Everything that is "not something else"
  - Well-defined non-systems areas
    - Theory (and algorithms, formal security)
    - Languages (functional side)
    - Graphics
- So what's left?
  - Architecture, OS, Compilers, Networking, etc.
  - Applications

Warning! You May Be Bored

- This material is redundant if
  - You've already had it
  - You already hacked and found it
  - Your first language was assembly
- Feel free to leave
  - I won't be offended
  - You'll still be held responsible for the material

Warning: Approximate Truths

- Some details for general info
- Most details ignored entirely
- Goals?
  - Simplicity
  - Coverage
- C, Unix, Uniprocessors, No Threads

Sample Questions

- What is a processor
- What are registers
- What is memory
- How's memory organized
- What's a cache and how's it organized
- What's a stack?
- Globals, locals, etc.
- PC, SP, conditions

What's a Processor Do?

while (1)
  fetch (get instruction)
  decode (understand instruction)
  execute

Execute: load, store, test, math, branch

Logical Organization

Logically

F1 D1 E1 F2 E2 D2

Processor Operations

Logically

F1 D1 E1 F2 E2 D2

Pipeline

F1	D1	E1			
	F2	D2	E2		
		F3	D3	E3	

8

What Is Memory

- "Slots" that hold values
- Slots are "numbered"
- Numbers are called addresses
- Two operations – read or write
  - e.g., read value from memory address X
- What can you put in memory?
  - Anything. No "intrinsic" meaning

9

What is Cache?

- Another kind of memory, physically
- Closer to the processor
- More expensive, faster

- Operation is logically ***transparent***
  - No naming/access by program

10

What Are Registers?

- Places to hold information
- Built into the processor
- "Named" specially

Why?

- Need a place to put information
- Simplifies the processor design

11

What Is a Program?

- Code
- Global variables
- Dynamically-allocated data
- Parameters, local variables

What Is a Program?


```
int *totalPtr;
Init(void)
{
   totalPtr = calloc(1, sizeof(int));
}
AddToTotal(int y)
{
   int i;
  *totalPtr += y;
}
```

- Code
- Global variables
- Dynamically-allocated data
- Parameters, local variables

Everything Becomes Memory

- Compiler/linker does translation
- Various ranges of memory are ***used*** for different purposes
  - Text/Code (program instructions)
  - Data (global variables)
  - Stack (local variables, parameters, etc)
  - Heap (dynamically allocated memory)

What Is a Stack?

- Data structure that supports push/pop
- Uses?
  - Anything w/ LIFO (last-in first-out behavior)
  - Only care about recent behavior
- For example? Procedure calls!

Procedure calls

- Incoming parameters from caller
  - Don't even know who caller is
- Local variables survive only when in use
- Temporary variables (a+b) * (c+d)

16

Stack Frames

- Frame == info for one procedure call
- Incoming parameters
- Return address for caller
- New local variables
- New temporary variables
- Size of frame

17

Stack Is Just Memory

- Defined, used by convention (agreement)
- Simplest representation?
  - Allocate chunk of memory
  - Have pointer into chunk
- Problems?
  - Must know maximum size of stack?
- How do you allocate?

18

What Does Memory Look Like?

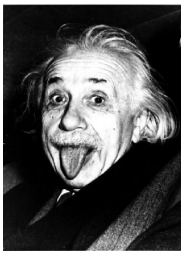
- Logical memory?
  - Code+data, stack, heap
  - Which ones grow?
  - How do you give them the most flexibility
- Physical memory?
  - Another can of worms, entirely
- We will move on to Memory Management

19

Fun Tricks

- What does this program do?

```
static void Loop(void)
{
    static char *startAddr;
    char local;
    printf("locations are %d %d\n", (&startAddr), (&local));
    Loop( );
}
int main(int argc, char *argv[ ])
{
    Loop( );
}
```



Fun Tricks

- Recursive function
- One static variable, one local variable
- Print difference between static variable and address of local variable
- Store address of local into static, recurse
- What does this tell you?
  - Address of local changes
  - Address of static remains fixed

Break

Memory Layout – Division of Responsibility

- Compiler: generates object file
  - Information is incomplete
  - Each file may refer to symbols defined in other files
- Linker: puts everything together
  - Creates one object file that is complete
  - No references outside this file (usually)

Division of Responsibility (cont)

- OS
  - Load object file into memory
  - Allow several different processes to share memory
  - Provide ways of dynamically allocating more memory

Components of Object File

- Header
- Two segments
  - Code segment and data segment
  - OS adds empty heap/stack segment while loading
- Size and address of each segment
  - Address of a segment is the address where the segment begins

Components of Object File – cont'd

- Symbol table
  - Information about stuff defined in this module
  - Used for getting from the name of a thing (subroutine/variable) to the thing itself
- Relocation information
  - Information about addresses linker should fix
    - External references
    - Internal references (e.g. absolute jumps)
- Additional information for debugger

Components of Object File – cont'd

- Type "man 5 a.out" on UNIX for more information on UNIX object files

Linker

- Three functions of a linker
  - Collect all the pieces of a program
  - Figure out new memory organization
    - Combine like segments
  - Touch-up addresses
- The result is a runnable object file (e.g. a.out)

Tasks of a Linker - I

- Compiler does not know final memory layout
  - It assumes everything starts at address zero
  - Compiler puts information in the symbol table to tell the linker how to rearrange safely
    - For exported function, absolute jumps, etc
  - What makes rearrangement tricky?
    - Addresses!

Tasks of a Linker – II

- Compiler does not know all the references
  - e.g. addresses of functions / variables defined in other files
  - Where it does not know, it just puts a zero, and leaves a comment (relocation info) for the linker to fix things up
- These are called *cross references*

Linker – a closer look

- Linker can shuffle segments around at will, but cannot rearrange information within a segment

Example Link Job


```
main( )
{
  static float x, val;

  extern float sin( );
  extern printf( ), scanf( )

  printf("Type number: ");
  scanf("%f", &x);
  val = sin(x);
  printf("Sine is %f", val);
}
```

```
float sin(x)
float x;
{
  static float temp1, temp2,
    result;

  – Calculate Sine –

  return result
}
```


Summary: Tasks of a Linker

- Read in object files produced by the compiler
- Produce a self-sufficient object file

- *Can this be done by scanning the obj files once?*

How Does Linker Work?

- At least two passes required
  - Pass 1: decide how to arrange memory
    - Read symbol table information
    - Read relocation info to see what additional stuff from libraries is required
  - Pass 2: address touch-up
    - Read segment and relocation info, modify addresses
    - Write new module with symbols, segments, addresses

Let us live the life of a linker

Pass 1 – Segment Relocation

- Pass 1 assigns input segment locations to fill-up output segments
- Needs to load only symbol table information at this point

Symbol Table

- Symbol table info:
  - Segments: name, size, old location, new location
  - Symbols: name, input segment containing it, offset within the segment

Pass 2 – Relocation

- In pass 2, linker reads data and relocation information from files, fixes up addresses, and writes a new object file
- Relocation information is crucial for this part

Relocation Information

- Contains function/variable address and offest values to be relocated
- How to relocate:
  - “Place final address of symbol here”
  - “Add final address of symbol to contents of this location”
  - “Add difference between the final and original address of segment to the contents of this location”

Relocation Examples

- LDW R1, _X
  - _X is external
- C reference of the form x = y.q;
  - y is external struct with {int p; int q}
- JAL _SIN
  - _SIN is external
- J _TARGET
  - _TARGET is in the original segment

Putting It Together

- Pass 1:
  - Read symbol table
  - Rearrange segments
- Pass 2:
  - Read symbol table and relocation information
  - Touch-up addresses
  - Write new object file

Dynamic linking

- Static linking – each lib copied into each binary
- Dynamic linking:
  - Instead of system call code, a stub that find code in memory, or load it if it is not present
  - Dynamic loader does the job at loading time
- Pros:
  - all procs can share copy (shared libraries)
    - Standard C library
  - live updates

Dynamic loading

- Program can call dynamic linker via
  - dlopen()
  - library is loaded at running time
- Pros:
  - More flexbility -- A running program can
    - creates a new program
    - invokes the compiler
    - invokes the linker
    - load it!