CS 354 - Machine Organization & Programming
Tuesday Nov 1 and Thursday Nov 3, 2022

Midterm Exam - Thurs Nov 10th, 7:30 - 9:30 pm

If your Lecture number is	and the first letter of your family name is ,	then, your assigned exam room is:
001	A-K	B130 Van Vleck
001	L-Z	B102 Van Vleck
002	A-R	B10 Ingraham
002	S-Z	19 Ingraham

- **UW ID and #2 required**
- **closed book, no notes, no electronic devices (e.g., calculators, phones, watches) see "Midterm Exam 2" on course site Assignments for topics**

Homework hw4: DUE on or before Monday, Nov 7

Homework hw5: will be DUE on or before Monday, Nov 14

Project p4A: DUE on or before Friday, Nov 4th

Project p4B: DUE on or before Friday, Nov 11th

Last Week

Direct Mapped Caches - Restrictive Fully Associative Caches - Unrestrictive Set Associative Caches - Sweet! Replacement Policies Writing to Caches	Writing to Caches (cont) Cache Performance Impact of Stride Memory Mountain C, Assembly, & Machine Code

This Week

Low-level View of Data Registers Operand Specifiers & Practice L18-7 Instructions - MOV, PUSH, POP Operand/Instruction Caveats Instruction - LEAL	Instructions - Arithmetic and Shift Instructions - CMP and TEST, Condition Codes Instructions - SET & Jumps Encoding Targets & Converting Loops
Next Week: Stack Frames and Exam 2 B&O 3.7 Intro - 3.7.5, 3.8 Array Allocation and Access 3.9 Heterogeneous Data Structures	

Copyright © 2016-2022 Jim Skrentny

C, Assembly, & Machine Code

HIGH LEVEL LANGUAGE — A mnemonic for Machine Code. — In the beginning.

C Function	Assembly (AT&T)	Machine (hex)
`int accum = 0;`		
`int sum(int x, int y)`	`sum:`	
`{`	`pushl %ebp`	`55`
	`movl %esp, %ebp`	`89 e5`
	`movl 12(%ebp), %eax`	`8b 45 0C`
`int t = x + y;`	`addl 8(%ebp), %eax`	`03 45 08`
`accum += t;`	`addl %eax, accum`	`01 05 ?? ?? ?? ??`
`return t;`	`popl %ebp`	`5D`
`}`	`ret`	`C3`

C

- High level language that enables more productive coding.
- Helps us write correct code.
- Can be compiled and run on different machines.

→ What aspects of the machine does C hide from us?

Machine Instructions, Addressing Modes, Registers, Condition Codes.

Assembly (ASM)

- Human readable representation of machine code.
- Very machine dependent.

→ What ISA (Instruction Set Architecture) are we studying? IA-32/x86.

→ What does assembly remove from C source?

Logical control structures (IF, SWITCH, LOOPS)
Local variable names + data types.
Composites - Arrays, Structures, Unions.

→ Why Learn Assembly?

1. To better understand the stack.
2. Helps you identify any inefficiencies.
3. To understand compiler optimization.

Machine Code (MC) **is**

- Elementary CPU instructors and data in binary.
- The encodings that a particular machine understands.

→ How many bytes long is an IA-32 instructions? All of the instructions are 1-15 bytes

The most common instructions take the fewest amount of bytes.

Copyright © 2016-2022 Jim Skrentny

Low-Level View of Data

C's View

- Variables declared of a specific type.
- Types can be complex composites built from arrays, structs, unions.

Machine's View

Memory is an array of bytes indexed by address, where each element is a byte.

❈ *Memory contains bits that do not* distinguish instructions from data from pointers.

Everything is the SAME!

→ How does a machine know what it's getting from memory?

1. By how it is accessed. Instruction Fetch vs Operand Load.
2. By the instruction itself.

Assembly Data Formats

C	IA-32	Assembly Suffix	Size in bytes
`char`	byte	b	1
`short`	word	w	2
`int`	double word	l	4
`long int`	double word	l	4
`char*` (any ptr)	double word	l	4
`float`	single precision	s	4
`double`	double prec	l	8 - quad word
`long double`	extended prec	t	10 - typically 12

❈ *In IA-32 a word* is actually 2 bytes.

Copyright © 2016-2022 Jim Skrentny

Registers

What? Registers

The fastest memory directly accessed by the CPU.

They can store 1,2,4 bytes (4 bytes for addresses, ints…).

General Registers

Pre-Named location that stores up to 32-bit values (4 bytes).

The e in front of it treats it as a 32 bit register.

WHEN YOU WRITE TO %eax for example, IT OVERWRITES %ax and %al!!!!

%ax means that we want to treat the registers as ONE 16 bit value rather than 2 8 bit values. X means extended.

bit 31 16 15 High Registers 8 7 Low registers 0

%eax	Accumulator	**%ax**	**%ah**	**%al**
%ecx	Count	**%cx**	**%ch**	**%cl**
%edx	Data	**%dx**	**%dh**	**%dl**
%ebx	Base	**%bx**	**%bh**	**%bl**
%esi	Source Index	**%si**		
%edi	Destination Index	**%di**		
%esp	Stack Pointer	**%sp**		
%ebp	Base Pointer	**%bp**		

You don't have to listen to the labels, but it is best to.

Program Counter PC **%eip**

Extended Instruction Pointer - tells you the next instruction.

Condition Code Registers E-Flags (Error Flags also)

1 bit registers that store the status of the most recent ALU operation.

Copyright © 2016-2022 Jim Skrentny

Operand Specifiers

What? Operand specifiers are

- S Source - Location of value to be used by the instruction.
- D Destination - Location to store the result of the instruction.

Why?
Enables instruction to specify constants, registers, and memory locations.

How? IA-32 ASM has 3 kinds of operand specifiers

1.) IMMEDIATE specifies an operand value that's a constant.

specifier	**operand value**	
$*Imm*	*Imm*	The specifier is the $. Immediate is in C's int literal format.

Literal Formats:
0x801 = HEX
073 = OCTAL (not on test lolo)
10 = DECIMAL

2.) REGISTER specifies an operand value that's in a register.

specifier	**operand value**	
$\%E_a$	$R[\%E_a]$	specifier is a %, E is extended and a is just a sub for any register.

3.) MEMORY specifies an operand value that's in memory at an effective address.

specifier	**operand value**	**effective address**	**addressing mode name**
Imm	M[EffAddr]	*Imm*	ABSOLUTE (in memory) - Address as part of instruction.
$(\%E_a)$	M[EffAddr]	$R[\%E_a]$	INDIRECT - Access from register
$Imm(\%E_b)$	M[EffAddr]	$Imm+R[\%E_b]$	BASE + OFFSET - Add some offset to a stored pointer.
$(\%E_b,\%E_i)$	M[EffAddr]	$R[\%E_b]+R[\%E_i]$	INDEXED - Add 2 registers
$Imm(\%E_b,\%E_i)$	M[EffAddr]	$Imm+R[\%E_b]+R[\%E_i]$	BASE + INDEX + OFFSET - Add 2 registers and an offset.
$Imm(\%E_b,\%E_i,s)$	M[EffAddr]	$Imm+R[\%E_b]+R[\%E_i]*s$	SCALED INDEXING Offset + Base + (Index * ScaleFactor)
$(\%E_b,\%E_i,s)$	M[EffAddr]	$R[\%E_b]+R[\%E_i]*s$	Base + (Index * ScaleFactor) [No offset]
$Imm(,\%E_i,s)$	M[EffAddr]	$Imm+R[\%E_i]*s$	Offset + (Index * ScaleFactor) [No base]
$(,\%E_i,s)$	M[EffAddr]	$R[\%E_i]*s$	(Index * ScaleFactor) [No offset or base]

s = scale factor must be 1,2,4,8

Copyright © 2016-2022 Jim Skrentny

Operands Practice

Given:

Memory Addr	Value	Register	Value
0x100	0xFF	%eax	0x104
0x104	0xAA	%ecx	0x1
0x108	0x11	%edx	0x4
0x10C	0x22		
0x110	0x33		

→ What is the value being accessed? Also identify the type of operand, and for memory types name the addressing mode and determine the effective address.

	Operand	Value	Type:Mode	Effective Address
1.	`(%eax)`	0xAA	Memory Indirect	0x104
2.	`0xF8(,%ecx,8)`	0xFF	Scaled Index	
3.	`%edx`	0x4	Register	
4.	`$0x108`	0x108	Immediate	
5.	`-4(%eax)`	0xFF	BASE + OFFSET	(0x104 - 0x04) = 0x100
6.	`4(%eax,%edx,2)`	0x33	Memory: Scaled Index	0x4 + 0x104 + (0x4 * 2) = 0x108 + 0x8 = 0x110
7.	`(%eax,%edx,2)`	0x22	Scaled Index	(0x104 + 0x4 * 2) = 0x10C
8.	`0x108`		Absolute	
9.	`259(%ecx,%edx)`	0x11	Base + Index + Offset	0x1 + 0x4 = 0x5 + 259 = 0x5 + 0x103 = 0x108

Copyright © 2016-2022 Jim Skrentny

Instructions - MOV, PUSH, POP

What? These are instructions to copy data from S to D.

Why? Enable info to move around in memory and registers.

How?

instruction class	operation	description
MOV S, D movb, movw, movd. Byte, Word, Double Word Move.		Copy from S to D when S and D are the same size.
MOVS S, D movsbw, movsbl, movswl. Move Byte to World, Word to Double Word	Sign Extend	Copy from smaller S to larger with Sign Extend (SEXT)
MOVZ S, D movzbw, movzbl, movzwl. Move Byte to World, Word to Double Word	Zero Extend	Copy from smaller S to large D with Zero Extend (ZEXT)

pushl S

R[%esp] <— R[%esp] - 4 Makes room on the stack (goes down because stack is inverted.)

M[R[%esp]] <— S copy S into stack at top of stack.

popl D

D <— M[R[%esp]] copy from top of stack to D
R[%esp] <— R[%esp] + 4 shrink the stack, move stack pointer up by 4 bytes.

Practice with Data Formats

→ What data format suffix should replace the _ given the registers used?

l	1. `mov_ %eax, %esp`	Register to Register - 4 bytes
l	2. `push_ $0xFF`	Immediate to Memory - 4 bytes
w	3. `mov_ (%eax), %dx`	Memory to Register - 2 bytes
b	4. `mov_ (%esp, %edx, 4), %dh`	Memory to Register - 1 byte
b	5. `mov_ 0x800AFFE7, %bl`	Memory to Register - 1 byte
w	6. `mov_ %dx, (%eax)`	Register to Memory - 2 bytes (16 bits so its 2 bytes)
l	7. `pop_ %edi`	Memory to Register - 4 bytes.

✵ *Focus on register type operands*
Since they can be 1, 2, or 4 bytes.

What determines the suffix for mov is that depending on the register or immediate's size, when we go into memory, that amount of bytes are read/transfered. So 6 is saying that we take a word-sized register, and place its contents into memory, so that's why it's movw.

Copyright © 2016-2022 Jim Skrentny

Operand/Instruction Caveats

Missing Combination?

→ Identify each source and destination operand type combinations.

1. `movl $0xABCD,%ecx` Immediate to Register - 4 bytes
2. `movb $11,(%ebp)` Immediate to Memory Indirectly - 1 byte
3. `movb %ah,%dl` Register to Register - 1 byte
4. `movl %eax,-12(%esp)` Register to Memory (Base + Offset) - 4 bytes
5. `movb (%ebx,%ecx,2),%al` Memory to Register - 1 byte

→ What combination is missing?

Memory to Memory is not allowed in IA-32.

Instruction Oops!

→ What is wrong with each instruction below?

1. `movl %bl,(%ebp)` You can't move 4 bytes from a register with 1 byte (Instruction & Source mismatch).
2. `movl %ebx,$0xA1FF` You can't change immediate with a move. (Immediate cannot be a dest).
3. `movw %dx,%eax` Instruction and destination don't match (use movswl)
4. `movb $0x11,(%ax)` Memory Registers MUST be 32 bits (need to use %e…)
5. `movw (%eax),(%ebx,%esi)` Memory to Memory is not allowed!
6. `movb %sh, %bl` There is no sh register!

Copyright © 2016-2022 Jim Skrentny

Instruction - LEAL

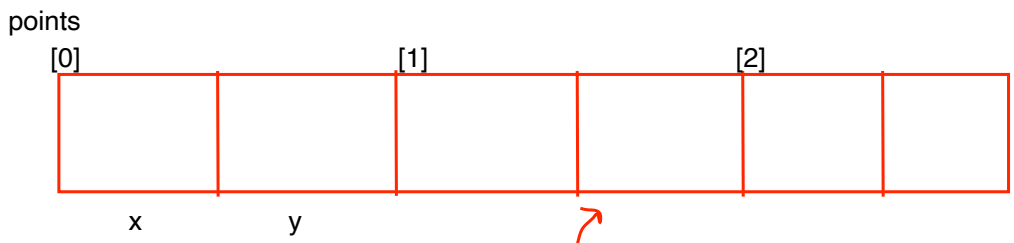
Load Effective Address

S must be a memory operand, but memory is not accessed, also Condition Codes are not set.

```
leal S,D      D <-- &S
```

We copy the address of the source into the destination (say we wanted to copy the start of the array somewhere).

LEAL vs. MOV




```
struct Point {
   int x;
   int y;
} points[3];
```

```
int y = points[i].y;        mov 4(%ebx,%ecx,8),%eax
```

points

counter/index

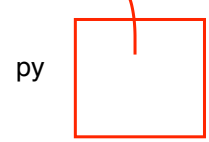
2 int jump

```
        points[1].y;
```

The assembly here grabs the memory starting at %ebx (array start), goes up by i * 8 bytes, and i is in %ecx register. We then jump by an offset of 4 to get the y coordinate and store the value of y into %eax.

```
int *py = &points[i].y;     leal 4(%ebx,%ecx,8),%eax
```

Same idea as above, except we copy the address of points[i].y into the %eax register.



LEAL Simple Math

```
leal  -3(%ebx), %eax        subl $3, %ebx
                            movl %ebx, %eax
```

Almost equivalent, but it changes %ebx!

leal doesn't actutally deref by the way, it merely is be treated as an address. That's why its good for math because you can do algebra on the Source side and have it keep its value as oppose to mov where it will try to do a lookup when you're done with your algebra.

→ Suppose register %eax (X) holds x and %ecx (Y) holds y.
What value in terms of x and y is stored in %ebx for each instruction below?

1. `leal (%eax,%ecx,8),%ebx` %eax + %ecx*8 = x + 8y => %ebx
2. `leal 12(%eax,%eax,4),%ebx` %eax + %eax*4 + 12 = x + 4x + 12 => %ebx
3. `leal 11(%ecx),%ebx` %ecx + 11 = y + 11 => %ebx
4. `leal 9(%eax,%ecx,4),%ebx` %eax + %ecx*4 + 9 = x + 4y + 9 => %ebx

Scale factor can only be 1,2,4,8

Copyright © 2016-2022 Jim Skrentny

Instructions - Arithmetic and Shift

Unary Operations - One operand

```
INC D      D <-- D + 1
DEC D      D <-- D - 1
NEG D      D <-- -D
NOT D      D <-- ~D
```

Can't use this with immediates, they're considered constant.

2's complement negation, so NEG 2 = -2.

Bitwise complement, so it just flips the bits.

Binary Operations

```
ADD S,D    D <-- D + S
SUB S,D    D <-- D - S
IMUL S,D   D <-- D * S
XOR S,D    D <-- D ^ S
OR S,D     D <-- D | S
AND S,D    D <-- D & S
```

Add S to D.
Subtract S from D
Multiply S from D
XOR S to D
OR S to D
AND S to D

Given:

Memory	
0x100	0xFF
0x104	0xAB
0x108	0x10

CPU	
%eax	0x100
%ecx	0x1
%edx	0x2

→ What is the destination and result for each? (do each independently)

1. `incl 4(%eax)` Destination is memory at %eax + 4 = 0x100 + 4 = 0x104. Read 0x104 to get 0xAB, now increment by 1 to do 0xAB + 0x1 = 0xAC
2. `addl %ecx,(%eax)` Add to memory at %eax which is 0xFF the value of %ecx, so you do 0xFF + 0x1 = 0x100
3. `addl $32,(%eax,%edx,4)` Add to memory located at %eax + %edx*4, which is 0x100 + 0x2*4 = 0x108 and the value there is 0x10. We then add 0x20 (32 in hex) to 0x10 to get 0x10 + 0x20 = 0x30
4. `subl %edx,0x104` Subtract to memory located at 0x104, which is 0xAB the value in %edx which is 0x2. So the value is 0xA9 in 0x104

Shift Operations

- Move bits left or right K positions.
  0 <= k <= 31, k is 1 byte. It shifts based on the least significant 5 bytes using %cl.
- Fast int multiplication + division by powers of 2.

logical shift (zero filled, treated as an unsigned integer)

```
SHL k,D    D <-- D << K
SHR k,D    D <-- D >> K
```

0110 << 1 results in 1100. Went from 6 to 12. 0110 >> 1 results in 0011.

Shift Left/Right

arithmetic shift (What if its negative, it sign extends!)

```
SAL k,D    D <-- D << K
SAR k,D    D <-- D >> K
```

1110 (-2) << 1 results in 1100 (-4)

1110 (-2) >> 1 results in 1111 (-1)

Shift Arithmetic Left/Right

Copyright © 2016-2022 Jim Skrentny

Instructions - CMP and TEST, Condition Codes

What?

- Comparing Values Arithmetically (CMP) or Logically [Bitwise] (TEST).

- Set condition codes, do not change values.

Why?

Enable relational and logical operations.

How?

```
CMP S2,S1              CC <-- S1 - S2
```

Subtract S2 from S1, sets the condition code accordingly but does not return or store anything.

compb, compw, compl exist of course.

```
TEST S2,S1             CC <-- S1 & S2
```

Bitwise AND S2 and S1, sets the CC accordingly, but again does not store anything.

➢ What is done by `testl %eax, %eax`
Does a bitwise AND on two 32 bit registers and sets CC. So it does
%eax & %eax => %eax, and the CC depends on the value of this result, which is just %eax.

Condition Codes (CC)

ZF: zero flag ZF is 1 if the result was 0.

CF: carry flag CF is 1 if unsigned overflow error occurs (adding/multiplying).

SF: sign flag SF is 1 if the result is negative.

OF: overflow flag OF is 1 if signed overflow or underflow occurred.
If you did like a negative + negative and got a positive value, we had an integer underflow.

Copyright © 2016-2022 Jim Skrentny

Instructions - SET

What?

set a <u>byte register</u> to 1 if a condition is true, 0 if false
specific condition is determined from CCs

Sets the destination byte to the condition code value, 0 or 1.

How?

The last operation was….

`sete D`	`D <-- ZF`	If the zero flag is 1, then they're ==
`setne D`	`D <-- ~ZF`	If the zero flag is 0, then they're !=
`sets D`	`D <-- SF`	If the sign flag is 1, then we're < 0
`setns D`	`D <-- ~SF`	If the sign flag is 0, then we're > 0

Unsigned Comparisons:

`setb D`	`D <-- CF`	< below
`setbe D`	`D <-- CF | ZF`	<= below or equal
`seta D`	`D <-- ~CF & ~ZF`	> above
`setae D`	`D <-- ~CF`	>= above

b means below
a means above

R1 >= R2 would correspond to setae to test for this for unsigned values.

Signed (2's Complement) Comparisons

`setl D`	`D <-- SF ^ OF`	< less than
`setle D`	`D <-- (SF ^ OF) | ZF`	<= less than or equal
`setg D`	`D <-- ~(SF ^ OF) & ~ZF`	> greater
`setge D`	`D <-- ~(SF ^ OF)`	>= greater than or equal

l means less

g means greater

Example: a < b (assume int a is in %eax, int b is in %ebx)

1. `cmpl %ebx,%eax` compare double word, so %eax - %ebx (doesn't change values)

2. `setl %cl` set D (%cl) if previous result was a < b (less than) by checking SF XOR OF.

3. `movzbl %cl,%ecx` movz (copy) byte to double word, so it just zero extends %cl to %ecx.

Copyright © 2016-2022 Jim Skrentny

Instructions - Jumps

What? How we transfer program execution to another location.

target: Desired location of next instruction

Why? Enable selection and repetition

How? Unconditional Jump Always JMP to target

What jmp is doing:
%eip <- %eax
%eip <- mem[%eax]

indirect jump: Says the target address is in a register or memory.

```
jmp *Operand
```

jmp *%eax -> register value is the target

jmp *(%eax) -> register value holds an address in memory, and that value in memory is the address target.

direct jump: Target address is in instruction

```
jmp Label
```

jmp .L1

.L1

This is what the label might look like in code

How? Conditional Jumps

- Only going to jump if the condition is met, based on Condition Codes and the Instruction.

- Conditional Jumps can only be direct jumps

Same as above

```
both:       je Label      jne Label     js Label      jns Label
unsigned:   jb Label      jbe Label     ja Label      jae Label
signed:     jl Label      jle Label     jg Label      jge Label
```

==, !=, < 0, > 0.

Encoding Targets

What? Techniques used by direct jump instruction for specific target

Absolute Encoding

Target is specified as a specific 32-bit address.

Problems?

- code is not compact - takes 32 bits to encode our address
- code cannot be moved without changing the target.

Solution? Relative Encoding

Target is a specified as a distance from the jmp instruction

IA-32: distance can be 1, 2, 4 bytes.

Distance is calculated from instruction immediately after jump instruction. (PC + 1 always)
Instructions aren't constant length, so you need to be weary of the size of the previous or next instructions when calculating the jump size.

→ What is the distance (in hex) encoded in the `jne` instruction?

```
        Assembly Code          Address    Machine Code
Sets CC cmpl  %eax, %ecx
        jne .L1                0x_B8      75 ??   0x04
        movl  $11, %eax        0x_BA %eip
        movl  $22, %edx        0x_BC
 .L1:                          0x_BE
```

this is the dist we calculate: 0xBE - 0xBA = 0x04

look very closely at the addresses to calculate distance.

→ If the `jb` instruction is 2 bytes in size and is at 0x08011357 and the target is at 0x8011340 then what is the distance (hex) encoded in the `jb` instruction?

We go to PC + sizeof(instr) from jb, so the next valid address is 0x08011357 + 0x02 (jb instruction) = 0x08011359

Now: 0x08011340 - 0x08011359 = -0x_19

Copyright © 2016-2022 Jim Skrentny

Converting Loops

→ Identify which C loop statement (for, while, do-while) corresponds to each goto code fragment below.

Do-While

```
loop1:
      loop_body
      t = loop_condition
      if (t) goto loop1:
```

do {
  body
} while (cond)

WHILE

```
      t = loop_condition
      if (!t) goto done:
loop2:
      loop_body
      t = loop_condition
      if (t) goto loop2
done:
```

Condition

true

false

Condition

while(cond){body}

```
      loop_init
      t = loop_condition
      if (!t) goto done:
loop3:
      loop_body
      loop_update
      t = loop_condition
      if (t) goto loop3
done:
```

for(loop init; cond; loop_update){
body
}

Most compilers (gcc included)

Base the loop assembly code on the do-while form

Copyright © 2016-2022 Jim Skrentny

CS 354 - Machine Organization & Programming
Tuesday Nov 8, and Thursday Nov 10,2022

Midterm Exam - Thursday Nov 10th, 7:30 - 9:30 pm

If your Lecture number is	and the first letter of your family name is ,	then, your assigned exam room is:
001	A-K	B130 Van Vleck
001	L-Z	B102 Van Vleck
002	A-R	B10 Ingraham
002	S-Z	19 Ingraham

- **UW ID and #2 pencils required**
- **closed book, no notes, no electronic devices (e.g., calculators, phones, watches) See "Midterm Exam 2" on course site Assignments for topics**

Homework hw5: DUE on or before Monday Nov 14

Homework hw6: DUE on or before Monday Nov 21

Project p4B: DUE on or before Friday Nov 11

Project p5: DUE on or before Friday Nov 25

Last Week

C, Assembly, & Machine Code Low-level View of Data Registers Operand Specifiers & Practice L18-7 Instructions - MOV, PUSH, POP	Operand/Instruction Caveats Instruction - LEAL Instructions - Arithmetic and Shift Instructions - CMP and TEST, Condition Codes ---------- END of Exam 2 Material ----------

This Week: From L18: Instructions - SET, Jumps, Encoding Targets, Converting Loops	The Stack from a Programmer's Perspective The Stack and Stack Frames Instructions - Transferring Control Register Usage Conventions Function Call-Return Example
Next Week: Stack Frames B&O 3.7 Intro - 3.7.5 3.8 Array Allocation and Access 3.9 Heterogeneous Data Structures	

Copyright © 2016-2022 Jim Skrentny

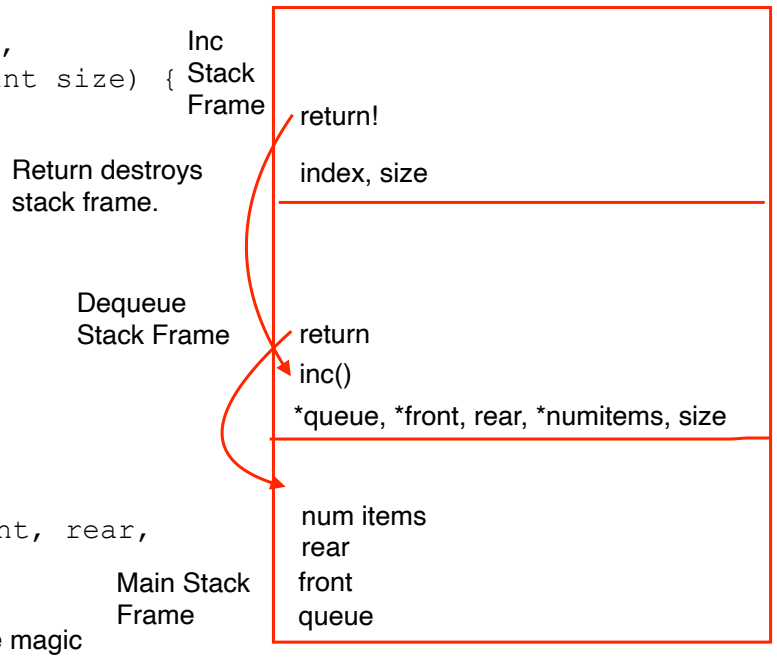
The Stack from a Programmer's Perspective

Consider the following code:

```
int inc(int index, int size) {
   int incindex = index + 1;
   if (incindex == size) return 0;
   return incindex;
}

int dequeue(int *queue, int *front,
         int rear, int *numitems, int size) {
   if (*numitem == 0) return -1;
   int dqitem = queue[*front];
   *front = inc(*front, size);
   *numitems -= 1;
   return dqitem;
}

int main(void) {
   int queue[5] = {11,22,33};
   int front = 0;
   int rear = 2;
   int numitems = 3;
   int qitem = dequeue(queue, &front, rear,
         &numitems, 5);
   ...
```



The value of return has to be 4 bytes (%eip). Structs are magic and somehow can be returned.

What does the compiler need to do to make function calls work?

- Transfer control to the callee
- Handle Passing Arguments
- Alloc and free stack frames
- Allocate and free parameters and locals
- Handle the return value
- Other details (debruh)

Copyright © 2016-2022 Jim Skrentny

The Stack and Stack Frames

Stack Frame aka Activation Record

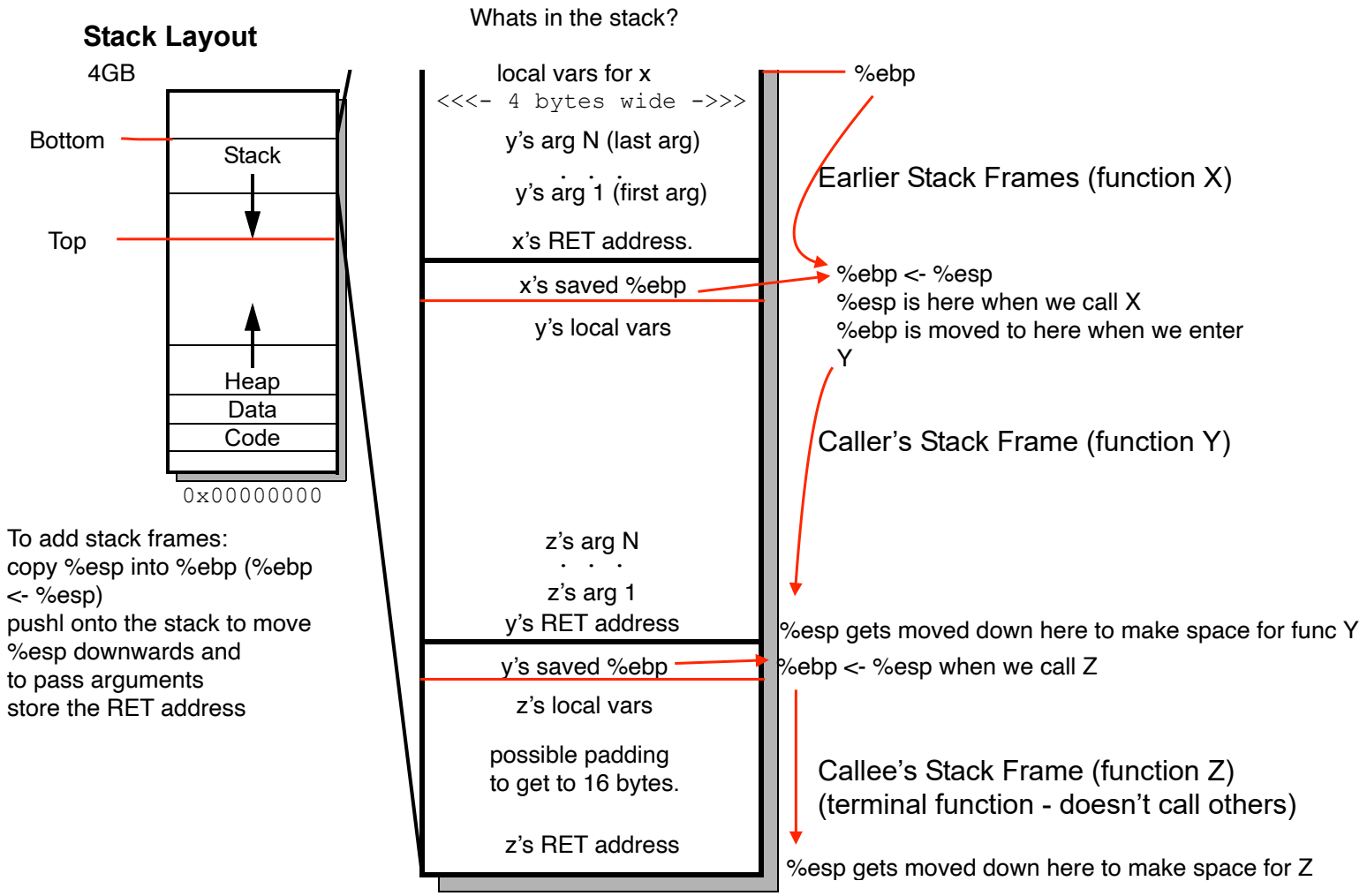
A block of memory used by a single function call (remember the stack is UPSIDE DOWN)

IA-32: Must be a multiple of 16 bytes

%ebp Base Ptr register points to bottom 4 bytes of stack

%esp Stack Pointer Register points to the top of the stack

Stack Layout



To add stack frames:
copy %esp into %ebp (%ebp <- %esp)
pushl onto the stack to move %esp downwards and
to pass arguments
store the RET address

✵ *A Callee's args* are in the caller's stack frame (Callee - Function that was called)

→ What is the offset from the %ebp to get to a callee's first argument?

8(%ebp). This is because x's saved %ebp is the start, we go back 4 bytes (sizeof %ebp), then we need to go back another 4 bytes (sizeof RET address as addresses are always 4 bytes) to get to arg 1.

→ When are local variables allocated on the stack?

1. Not enough registers
2. If they're arrays, structs, unions, or other complex data types
3. Code uses "address-of" (&), because we need a "permanent" memory address!

Copyright © 2016-2022 Jim Skrentny

Instructions - Transferring Control

Flow Control

function call:

`call *Operand` Indirect call (something holds the address to the subroutine and then is deref'd)

`call Label` Direct Call (address is referenced by a label directly)

steps (for both forms of call)

1. push return address onto stack

pushl %eip → subl $4, %esp
movl %eip, (%esp)

2. jump to start of callee function

jmp *Operand
jmp Label

function return:

`ret`

step

1. Jump to the return address that is popped off the stack

popl %eip → movl (%esp), %eip
addl %4, %esp

Stack Frames

allocate stack frame:

No special instructions

Use: subl $X [Size in bytes of new stack frame], %esp

free stack frame:

`leave` Free callee's stack frame

steps

1. Remove all of callee's Stack Frame except caller's saved base pointer %ebp.

Equivalent to: movl %ebp, %esp (shifts it back up to the top)

2. Restore the caller's stack frame

Equivalent to: popl %ebp

Copyright © 2016-2022 Jim Skrentny

Register Usage Conventions Requirements

Return Value

%eax stores the return value

Frame Base Pointer `%ebp`

callee uses to %ebp to…

1. Access callee's arguments
2. Access callee's local variables

Stack Pointer `%esp`

caller uses to %esp to…

1. setup arguments to function calls
2. save return address

callee uses to

1. Restore the return address
2. Save and restore the caller's stack frame

Registers and Local Variables

→ Why use registers? FAST! But data size is limited to 1,2,4 bytes.

→ Potential problem with multiple functions using registers?

Registers are shared -> conflicts might result
Caller and Callee must have consistent approach to saving and restoring register values to prevent overwriting value in a register another function needs.

IA-32

caller-save: %eax, %ecx, %edx

callee-save: %ebx, %esi, %edi

Copyright © 2016-2022 Jim Skrentny

Function Call-Return Example

```
int dequeue(int *queue, int *front, int rear, int *numitems, int size) {
   if (*numitem == 0) return -1;
   int dqitem = queue[*front];
   *front = inc(*front, size);
                                  1ab setup calleE's args
                                  2  call the calleE function
                                   a  save caller's return address
                                   b  transfer control to calleE
                                  7  caller resumes, assigns return value
   *numitems -= 1;
   return dqitem;
}
int inc(int index, int size) {    3  allocate callee's stack frame
                                   a  save calleR's frame base
                                   b  set callee's frame base
                                   c  set callee's top of stack
   int incindex = index + 1;      4  callee executes ...
   if (incindex == size) return 0;
   return incindex;               5  free callee's stack frame
}                                  a  restore calleR's top of stack
                                   b  restore calleR's frame base
                                  6  transfer control back to calleR
```

CALL code in `dequeue`

Address of Instruction Instruction

1a `0x0_2C  movl` index`,(%esp)` Copies index into memory at address pointed to by %esp, which is 0x58

b `0x0_2E  movl` size`,4(%esp)` Copies size into memory at address pointed to by 4 + %esp, which is 0x5C

2 `0x0_30  call inc`

a pushl %eip -> saves return address (puts 0x_55 (%eip) as that is the address of the next instr) at memory pointed to by %esp - 4)
NOTE: call stores basically PC + sizeof(instr) in the stack, as %eip get iterated to the next instruction before the jump internally.

b jmp to 0x0_F0 (where inc points to, so we change the %eip to this value)
Now %eip is 0x_F0, overwriting 0x_55 but 0x_55 is in the stack so its okay! %esp is still at 0x54 now.

RETURN code in `dequeue`

7 `0x0_55 movl %eax,(%ebx)`

CALL code in `inc` (Now $12 isn't actually allowed for stack sizes, but for notes its okay)

3a `0x0_F0  pushl %ebp` %esp - 4 and stores the value of %ebp (xE_70) in the memory pointed to by %esp

b `0x0_F2  movl %esp,%ebp` Store value of %esp in %ebp

c `0x0_F4  subl $12,%esp` Move the stack pointer down by 12 bytes to allocate space on stack

4 `0x0_F6` execute `inc` function's body %eip is updated to this point and will begin running code

RETURN code in `inc`

5 `0x0_FA  leave`

a Restore %esp for the caller -> movl %ebp, %esp. Undoes all of the allocated space we gave to %esp. Back to 0xE_70!

b popl %ebp (Take the value pointed to be %esp (address 0xE_50 right now) and copy it to %ebp, so %ebp is now 0xE_70, then do %esp + 4). %esp is now 0x0_55, and we're back to our RETURN address.

6 `0x0_FB  ret`

popl %eip (Take the value pointed to by %esp (address 0xE_54) and places it into the %eip register. So now %eip has the value 0x0_55 which hte command right after our initial function call, and %esp is now %esp + 4 or 0xE_58)

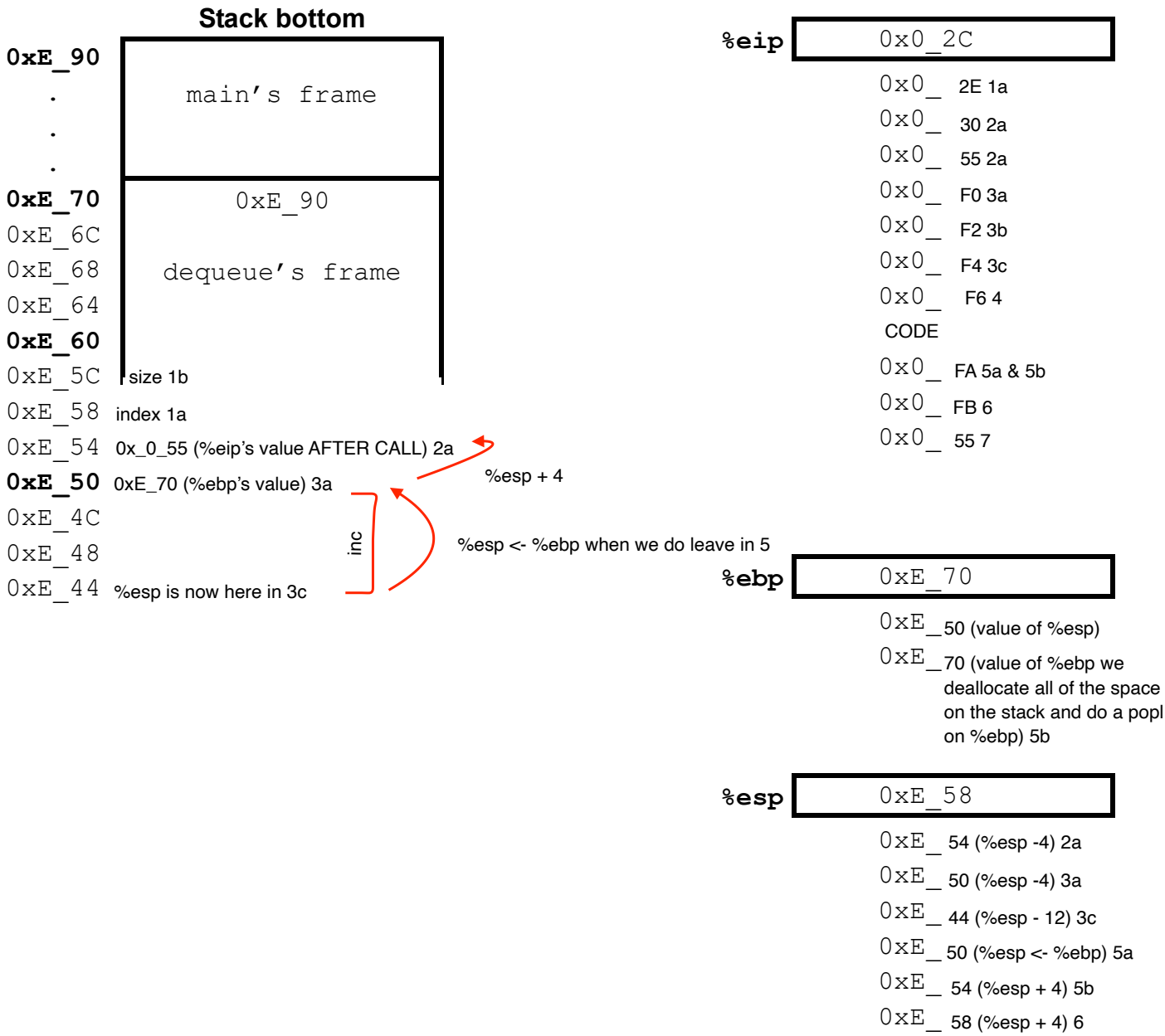
Copyright © 2016-2022 Jim Skrentny

If you check, all of the original values of the stack and base registers should be restored, %eip should be changed.

Function Call-Return Example

%eip will tick by sizeof(instr) to next instruction if there is no stack frame call, so that's why it went from 2C -> 2E

Execution Trace of Stack and Registers



Copyright © 2016-2022 Jim Skrentny

CS 354 - Machine Organization & Programming
Tuesday Nov15, Thursday Nov 17, 2022

Exam Results expected by Friday Nov 18

Homework hw6: DUE on or before Monday Nov 21

Homework hw7: DUE on or before Monday Nov 28

Project p5: DUE on or before Friday Dec 2

Last Week

Instructions - SET Instructions - Jumps Encoding Targets Converting Loops	The Stack from a Programmer's Perspective The Stack and Stack Frames Instructions - Transferring Control Register Usage Conventions Function Call-Return Example

This Week

Function Call-Return Example (L20 p7) Recursion Stack Allocated Arrays in C Stack Allocated Arrays in Assembly Stack Allocated Multidimensional Arrays	Stack Allocated Structs Alignment Alignment Practice Unions
Next Week: Pointers in Assembly, Stack Smashing, and Exceptions B&O 3.10 Putting it Together: Understanding Pointers 3.12 Out-of-Bounds Memory References and Buffer Overflow 8.1 Exceptions 8.2 Processes 8.3 System Call Error Handling 8.4 Process Control through p719	

Copyright © 2016-2022 Jim Skrentny

Recursion

Use a stack trace to determine the result of the call `fact(3)`:

```
int fact(int n) {
   int result;
   if (n <= 1) result = 1; //Base Case
   else        result = n * fact(n - 1); //Recursive Case
   return result;
}
```

direct recursion When a function calls itself (indirect is when a calls b calls c calls a)

recursive case Calls recursive function

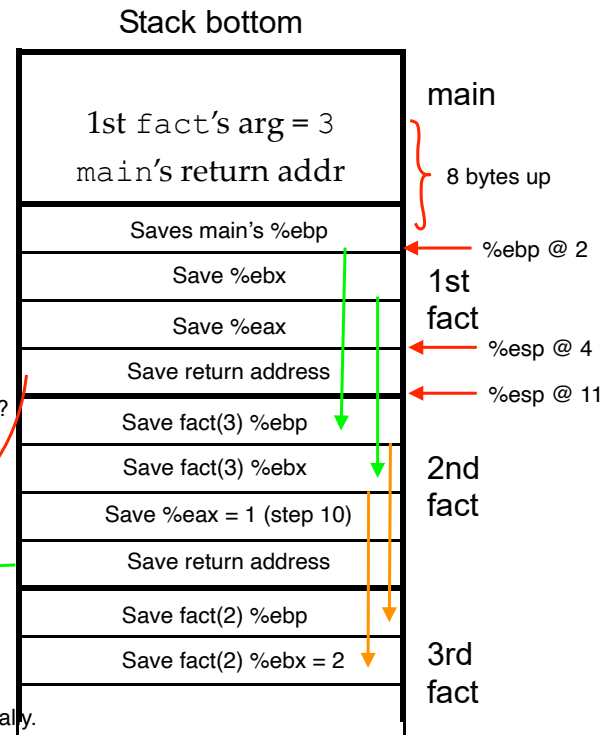
base case Stop recursion

"infinite" recursion Similar to infinite loop

leal doesn't actutally deref by the way, it merely is be treated as an address. That's why its good for math because you can do algebra on the Source side and have it keep its value as oppose to mov where it will try to do a lookup when you're done with your algebra.

Assembly Trace

```
fact:
 1 pushl %ebp      (%esp - 4, and value of %ebp is stored at address %esp - 4)
 2 movl %esp, %ebp  Copies %esp into %ebp
 3 pushl %ebx      %esp - 4, and stores value of %ebx at address %esp - 4
 4 subl $4,%esp    %esp - 4 to make more space
 5 movl 8(%ebp),%ebx  Grabs local variable from prior call (arg 3 in fact(3))
 6 movl $1,%eax    Put 1 into %eax
 7 cmpl $1,%ebx    Just does %ebx - 1
 8 jle .L1         Jump if less than or equal to 0. Based on above so is %ebx - 1 <= 0?
 9 leal -1(%ebx),%eax  Create value by doing %ebx - 1 to get the
                       n-1 step and storing into %eax.
10 movl %eax,(%esp)  Put %eax where %esp is pointing to (free space)
11 call fact       Save RET (next instr), change %eip to fact
   imull %ebx,%eax
      Multiply %ebx and %eax and store into %eax and then goto L1
.L1:
31 addl $4,%esp    Undoes any added space to stack
   popl %ebx       Places the value pointed to by %esp into %ebx, in this case 2 initially.
   popl %ebp       Places the value pointed to by %esp into %ebp, in this case it goes back to the %ebp saved in fact(2).
   ret             popl %eip, Places the value pointed to by %esp into %eip, in this case it is the return address pointed to by fact(2)
```



Keep doing this and you will get the correct result in %eax.

- ***"Infinite" recursion causes***

  Stack overflows, as each recursive case consumes stack memory.

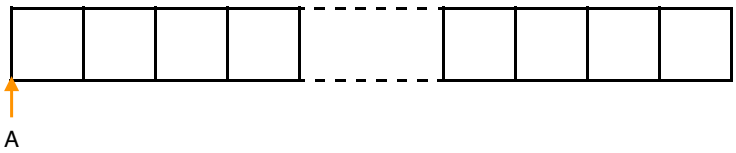
- ***When tracing functions in assembly code***

  Focus on the function body in a high level language

Stack Allocated Arrays in C

Recall Array Basics

T `A[`*N*`];` where *T* is the element datatype of size *L* bytes and *N* is the number of elements



1. Contigious region of the stack

2. Identifier is associated with the start of the array

✵ *The elements of* `A`

Recall Array Indexing and Address Arithmetic are accessed using address arithmetic

`&A[i]` = A + i = X (starting address of A) + L (element size in bytes) * i (index)

→ For each array declarations below, what is L (element size), the address arithmetic for the ith element, and the total size of the array?

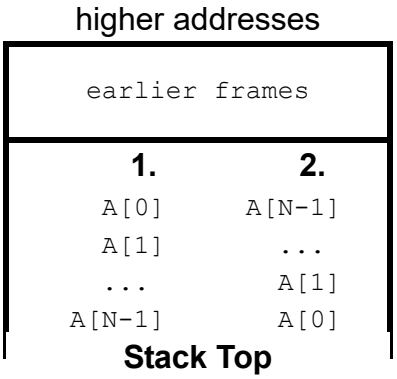
	C code	L	address of ith element	total array size
1.	`int I[11]`			
2.	`char C[7]`			
3.	`double D[11]`			
4.	`short S[42]`			
5.	`char *C[13]`			
6.	`int **I[11]`			
7.	`double *D[7]`			

Copyright © 2016-2022 Jim Skrentny

Stack Allocated Arrays in Assembly

Arrays on the Stack

→ How is an array laid out on the stack? Option 1 or 2:



✵ *The first element (index 0) of an array*

Accessing 1D Arrays in Assembly

Assume array's start address in %edx and index is in %ecx

```
movl (%edx, %ecx, 4), %eax
```

→ Assume `I` is an `int` array, `S` is a `short int` array, for both the array's start address is in %edx, and the index `i` is in %ecx. Determine the element type and instruction for each:

	C code	type	assembly instruction to move C code's value into %eax
1.	`I`	int*	I = movl %edx, %eax (leal could work here too)
2.	`I[0]`	int	*(I + 0*sizeof(int)) = movl (%edx), %eax
3.	`*I`		*I = movl (%edx), %eax
4.	`I[i]`		*(I + sizeof(int)*i) = movl (%edx, %ecx, 4), %eax
5.	`&I[2]`		&(I + sizeof(int)*2) = leal 8(%edx), %eax
6.	`I+i-1`	int*	I + sizeof(int)*i - (1 * sizeof(int)) = I + 4*i - 4 = leal -4(%edx, %ecx, 4), %eax
7.	`*(I+i-3)`		
8.	`S[3]`		
9.	`S+1`		
10.	`&S[i]`		
11.	`S[4*i+1]`	short	*(S + 4*sizeof(short)*i + 1*sizeof(short)) = *(S + 8i + 2) = movswl 2(%edx, %ecx, 8), %eax
12.	`S+i-5`		

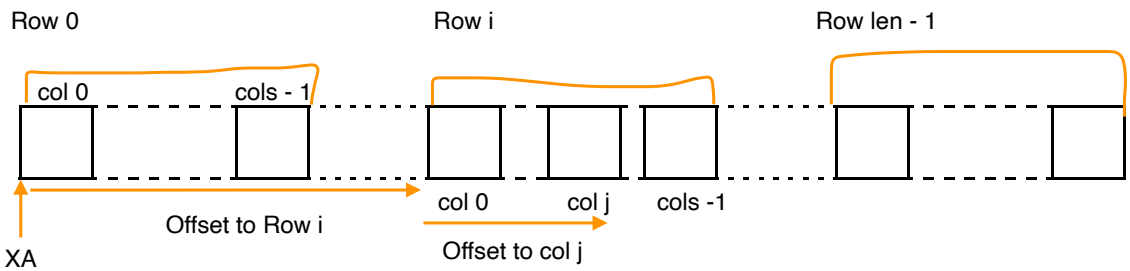
Copyright © 2016-2022 Jim Skrentny

Stack Allocated Multidimensional Arrays

Recall 2D Array Basics

Text

T `A[`*R*`][`*C*`];` where *T* is the element datatype of size *L* bytes,
R is the number of rows and *C* is the number of columns



✵ *Recall that 2D arrays are stored on the stack*

Row-Major Order

```
int A[5][3];        typedef int row_t[3];
                    row_t A[5];
```

Columns per row

Rows

Accessing 2D Arrays in Assembly

`&A[i][j]` = Start Address + Offset to Row i + offset to row j

= XA + (C * i * L) + (L * j) where L is the sizeof the type that the array holds, C is the number of columns

Given array `A` as declared above, if x_A in %eax, i in %ecx, j in %edx
then `A[i][j]` in assembly is:

```
leal (%ecx, %ecx, 2), %ecx
```
%ecx = i + i*2 = 3i.

```
sall $2, %edx
```
%edx = 2^2 * j = 4j

```
addl %eax, %edx
```
%edx = (XA + 4j)

```
movl (%edx, %ecx, 4), %eax
```
%eax = (XA + 4j + 4*(3i)) => %eax = (XA + 4j + 12i)

Compiler Optimizations

- If only accessing part of array

  Then the compiler will make a pointer to that point of the array

- If taking a fixed stride through the array

  Compiler uses stride*element_size as the offset.

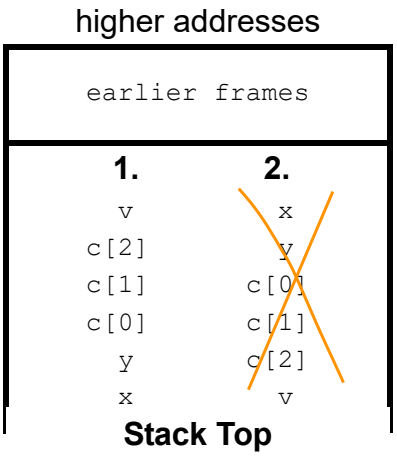
Copyright © 2016-2022 Jim Skrentny

Stack Allocated Structures

Structures on the Stack

```
struct iCell {
   int x;  4 bytes
   int y;  4 bytes
   int c[3]; 12 bytes
   int *v;  4 bytes
};
```

Offsets
0 bytes
4 bytes
8 bytes (to head of list)
20 bytes

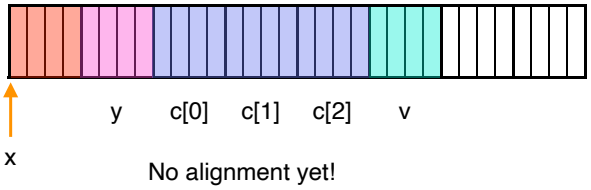


→ How is a structure laid out on the stack? Option 1 or 2:

Option 1, as the stack is upside down!

The compiler

- Associate Data Member names with their offset from the start of the struct



- Uses address arithmetic with offset to access data members

✵ *The first data member of a structure* is closest to the top of the stack

Accessing Structures in Assembly

Given:

```
struct iCell ic = //assume ic is initialized

void function(iCell *ip) {
```

→ Assume `ic` is at the top of the stack (memory address in %esp!), %edx stores `ip` and %esi stores `i`.
Determine for each the assembly instruction to move the C code's value into %eax:

	C code	assembly
1.	`ic.v`	movl 20(%esp), %eax
2.	`ic.c[i]`	movl 8(%esp, %esi, 4), %eax
3.	`ip->x`	Deref ip and grab x = movl (%edx), %eax (No offset as it's x)
4.	`ip->y`	Deref ip and grab y = movl 4(%edx), %eax
5.	`&ip->c[i]`	Deref ip and grab c[i]'s address = leal 8(%edx, %esi, 4), %eax

✵ *Assembly code to access a structure*

Does not have data member names, only offsets.

Copyright © 2016-2022 Jim Skrentny

Alignment

What?

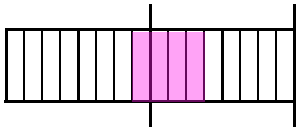
Most computer systems restrict the addresses where primitives can be stored.

Why?

Better performance

Example: Assume cpu reads 8 byte words
f is a misaligned float

Slow: 2 read, extract, combine for the float.



Restrictions

IA-32 has no alignment restrictions.

Linux: short Address must be a multiple of 2.
int, float, pointer, double Must be a multiple of 4

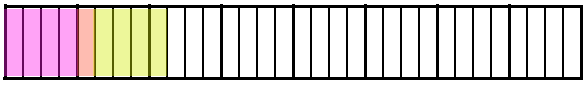
Windows: same as Linux except
double must be a multiple of 8

Implications
Padding added by compiler to meet alignment requirement.

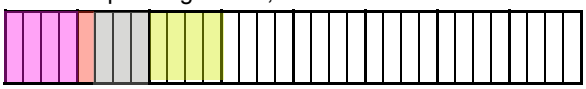
Structure Example

```
struct s1 {
   int i;
   char c;
   int j;
};
```

This is unaligned!!



There was padding added, as the char doesn't need to be aligned, but the int does.



✵ *The total size of a structure*

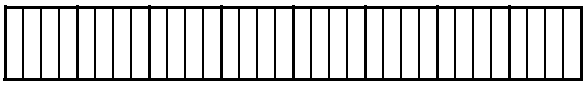
is typically a multiple of the largest data member size!! So above its okay, because its 12 which is a multiple of 4.

Copyright © 2016-2022 Jim Skrentny

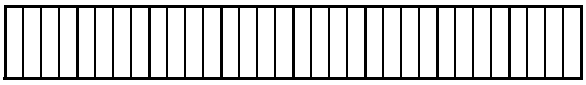
Alignment Practice

→ For each structure below, complete the memory layout and determine the total bytes allocated.

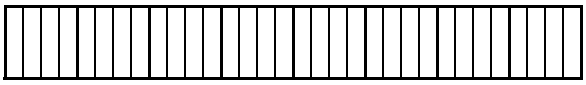
```
1) struct sA {
      int i;
      int j;
      char c;
   };
```



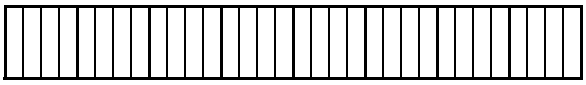
```
2) struct sB {
      char a;
      char b;
      char c;
   };
```



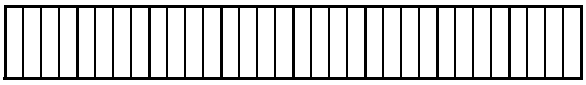
```
3) struct sC {
      char c;
      short s;
      int i;
      char d;
   };
```



```
4) struct sD {
      short s;
      int i;
      char c;
   };
```



```
5) struct sE {
      int i;
      short s;
      char c;
   };
```



❈ *The order that a structure's data members are listed*

Can affect the memory utilization.

Copyright © 2016-2022 Jim Skrentny

Unions

What? A *union* is

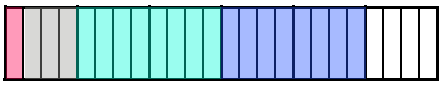
- Like a struct, except fields share the same memory
- Only allocates enough memory for the largest field.

Why?

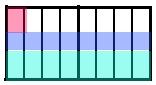
- Allows data to be accessed as different types (as C's typechecking is not done)
- Used to access hardware.
- Allows a *very* rudimentary way to implement polymorphism

How?

```
struct s {
   char c;
   int i[2];
   double d;
};
```



```
union u {
   char c;
   int i[2];
   double d;
};
```



So you can see how everything is written here together.
What we get out is how we interpret those bytes..

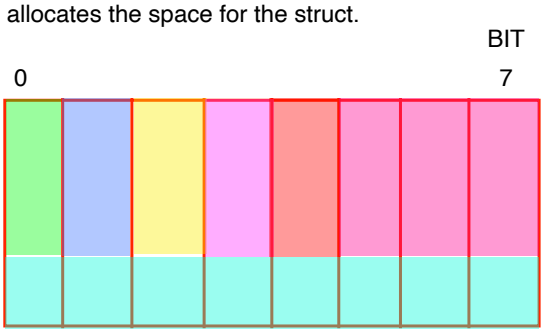
Example

We only have 8 bits to play with here (1 byte) due to our union.
The struct also is only 8 bits due to the bit fielding.

```
typedef union {
   unsigned char cntrlrByte;
   struct {
      unsigned char playbutn  : 1;
      unsigned char pausebutn : 1;
      unsigned char ctrlbutn  : 1;
      unsigned char fire1butn : 1;
      unsigned char fire2butn : 1;
      unsigned char direction : 3;
   } bits;
} CntrlrReg;
```

This basically allocates the space for the struct.

Each of these variables in the struct get 1 bit of this union dedicated to themselves. So we access them by doing bits.playbutn…




```
CntrlrReg c1;
readCtrlReg(c1.cntrlrByte);

if(c1.bits.fire1butn) {…fire shot…}
```

THE ORDER DOES MATTER,
so playbutn gets bit 0, pause gets bit 1 and so on..

Copyright © 2016-2022 Jim Skrentny

CS 354 - Machine Organization & Programming
Tuesday November 22, 2022

Homework hw6: DUE on or before Monday Nov 21
Homework hw7: DUE on or before Monday Dec 5
Project p5: DUE on or before Friday Dec 2nd (do before Wed Nov 23)
Project p6: Assigned soon and Due on last day of classes.

Last Week

Function Call-Return Example (L20 p7) Recursion Stack Allocated Arrays in C Stack Allocated Arrays in Assembly Stack Allocated Multidimensional Arrays	Stack Allocated Structs Alignment Alignment Practice Unions

This Week	**Next Week**
finish w11 Pointers Function Pointers Buffer Overflow & Stack Smashing Flow of Execution Exceptional Events Kinds of ExceptionsTransferring Control via Exception Table THANKSGIVING BREAK	Exceptions/System Calls in IA-32 & Linux Processes and Context User/Kernel Modes Context Switch Context Switch Example
Next Week: Signals, and multifile coding, Linking and Symbols B&O 8.5 Signals Intro, 8.5.1 Signal Terminology 8.5.2 Sending Signals 8.5.3 Receiving Signals 8.5.4 Signal Handling Issues, p.745	

Copyright © 2016-2022 Jim Skrentny

Pointers

Recall Pointer Basics in C

```
int i = 11;
int *iptr = &i;
*iptr = 22;
```

pointer type `int *`

pointee type used by compiler to determine the scaling factor used in ASM

pointer value `0x2A300F87`, `0x00000000` (NULL)
address used with addressing modes to specify an effective address in ASM

address of `&i`
& operator, becomes `leal` instr, which just calculates the effective address

dereferencing `*iptr`
* operator, becomes `mov` instr, which accesses mem at the effective address

Recall Casting in C

```
int *p = malloc(sizeof(int) * 11);

... (char *)p + 2
```

❈ *Casting changes the scaling factor used not the pointer's value.*

Function Pointers

What? A *function pointer*

- is a pointer to code
- stores the address of the first instruction of a function

Why?

enables functions to be

- passed to and returned from functions
- stored in arrays, e.g., faster switch logic - jump tables

How?

```
int func(int x) { ...}      //1. implement some function

int (*fptr)(int);           //2. declare function pointer

fptr = func;                //3. assign its function

int x = fptr(11);           //4. use function pointer
```

Example

```
#include <stdio.h>

void add     (int x, int y) { printf("%d + %d = %d\n", x, y, x+y); }
void subtract(int x, int y) { printf("%d - %d = %d\n", x, y, x-y); }
void multiply(int x, int y) { printf("%d * %d = %d\n", x, y, x*y); }

int main() {
    void (*fptr_arr[])(int, int) = {add, subtract, multiply};
    unsigned int choice;
    int i = 22, j = 11; //user should input

    printf("Enter: [0-add, 1-subtract, 2-multiply]\n");
    scanf("%d", &choice);
    if (choice > 2) return -1;
    fptr_arr[choice](i, j);
    return 0;
}
```

Function pointer array

parameters

0 1 2

Declare & init

Function return type

```
Enter: [0-add, 1-subtract, 2-multiply]
   2
   22 * 11 = 242
```

Buffer Overflow & Stack Smashing

Bounds Checking

```
int a[5] = {1,2,3,4,5};
printf("%d", a[11]);
```

→ What happens when you execute the code?
intermittent error - might print junk, or worse, might crash - seg fault

❈ *The lack of bounds checking array accesses is one of C's main vulnerabilities.*

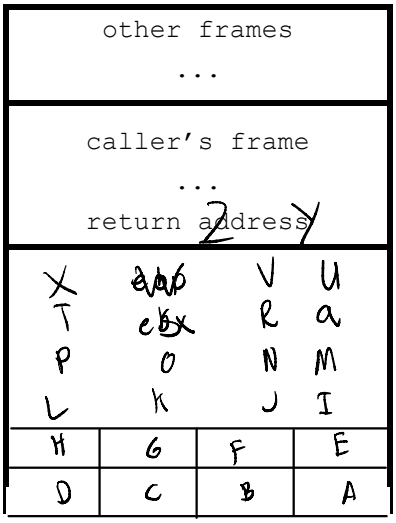
Buffer Overflow

- is exceeding the bounds of an array
- is particularly dangerous with stack allocated arrays

```
void echo() {
   char bufr[8];
   gets(bufr);
   puts(bufr);
}
```

Don't use gets! Use fgets!

Stack bottom



❈ *Buffer overflow can overwrite data outside the buffer.*

❈ *It can also overwrite the state of execution!*

You are overwriting registers!!

Stack Smashing

1. Get "exploit code" in
   enter input crafted to be machine instrs

2. Get "exploit code" to run
   overwrite return address with addr of buffer with exploit code

3. Cover your tracks
   restore stack so execution continues as expected

❈ *In 1988 the Morris Worm brought down the Internet using this kind of exploit.*

Copyright © 2016-2022 Jim Skrentny

Flow of Execution

What?

control transfer : transition from one instruction to another

control flow : a sequence of control transitions

- What control structure results in a smooth flow of execution? sequential

- What control structures result in abrupt changes in the flow of execution? selection
  repetition
  fcn call / return

Exceptional Control Flow

logical control flow : normal exception execution

exceptional control flow special execution
allows system to react to unusual / urgent / anomalies

event a change in processor state
may or may not be related to our current instruction

processor state
the internal storage elements
regs, flags, signals

Some Uses of Exceptions

process
- to ask for kernel services
- to share info. w/ other proc
- to send & receive msg

OS
- to communicate with process & hardware
- to switch execution among process
- to deal with memory page

hardware
- to indicate device status

Exceptional Events

What? An *exception*

- an event which side steps the usual logic flow
- can originate from Hardware or Software
- an indirect function call which abruptly changes execution flow

→ What's the difference between an asychronous vs. a synchronous exception?

asynchronous from event that is unrelated to current instruction

synchronous results from current instruction

General Exceptional Control Flow

0. normal flow

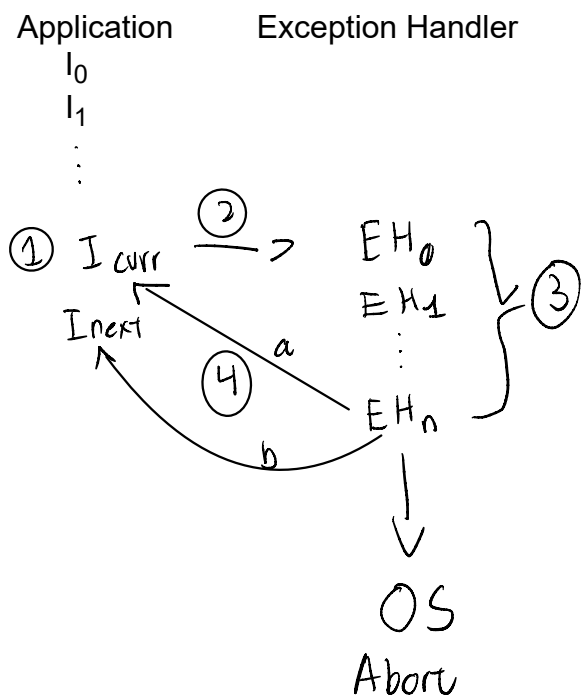
1. exeception occurs

2. transfer control to appropriate exception handler

3. exception handler runs

4. return control to:
   - (4a) current instruction (page fault)
   - (4b) I_{next} (file I/O)
   - (4c) OS- aborts (seg fault?)

default behavior is abort.



Kinds of Exceptions

→ Which describes a ***Trap***? ***Abort***? ***Interrupt***? ***Fault***?

async

1. Interrupt enables device to signal that they need attention
signal from external device
asynchronous
returns to Inext

How? Generally:

1. device signals interrupt
2. Finish current instruction
3. transfer control to appropriate exception handler
4. transfer control back to interrupted process's next instruction

sync vs. *polling* where software periodically checks device

sync

2. Trap enables process to interact with Operating System
intentional exception
synchronous
returns to Inext

How? Generally:

1. process indicates need for OS service

   `int` executes interrupt instruction
2. transfer control to the OS system call handler
   which calls service that is requested
3. transfer control back to process's next instruction

3. Fault handles problem w/ current instruction (page fault) (seg fault)
potentially recoverable error
synchronous
might return to Icurr and re-execute it

4. Abort cleanly ends process
nonrecoverable fatal errors e.g. Hardware failures
synchronous
doesn't retur

Copyright © 2016-2022 Jim Skrentny

CS 354 - Machine Organization & Programming
Tuesday Nov 29, and Thursday Dec 1 , 2022

Homework hw7: DUE on or before Monday Dec 5
Homework hw8: DUE on or before Monday Dec 12
Project p5: DUE on or before Friday Dec 2
Project p6: Available Friday and due on last day of classes.

Last Week

(from Week 11) Alignment Alignment Practice Unions Pointers Function Pointers Buffer Overflow & Stack Smashing	Flow of Execution Exceptional Events

This Week

Kinds of Exceptions (from Week 12) Transferring Control via Exception Table Exceptions/System Calls in IA-32 & Linux Processes and Context User/Kernel Modes Context Switch Context Switch Example	Meet Signals Three Phases of Signaling Processes IDs and Groups Sending Signals Receiving Signals
This Week and Next Week: Signals, and multifile coding, Linking and Symbols B&O 8.5 Signals Intro, 8.5.1 Signal Terminology 8.5.2 Sending Signals 8.5.3 Receiving Signals 8.5.4 Signal Handling Issues, p.745	

Copyright © 2016-2022 Jim Skrentny

Transferring Control via Exception Table

❋ *Exceptions transfer control to the Kernel.*

Transferring Control to an Exception Handler

1. push return addr (I_{curr} or I_{next})

2. push interrupted processor's state so it can be restarted

→ What stack is used for the push steps above?

The Kernel Stack (Not related to any single process)

3. do indirect function call which runs the appropriate exception handler

indirect function call uses exception table to determine what function to execute

Array of function pointers to the specific instruction

EHA = M[M[EBTR] + ENUM], Go to memory, get the "base" of the table from EBTR and then use ENUM to find the specific exception and then go to the location of that function in memory.

ETBR is for exception table base reg like pointer to array's base

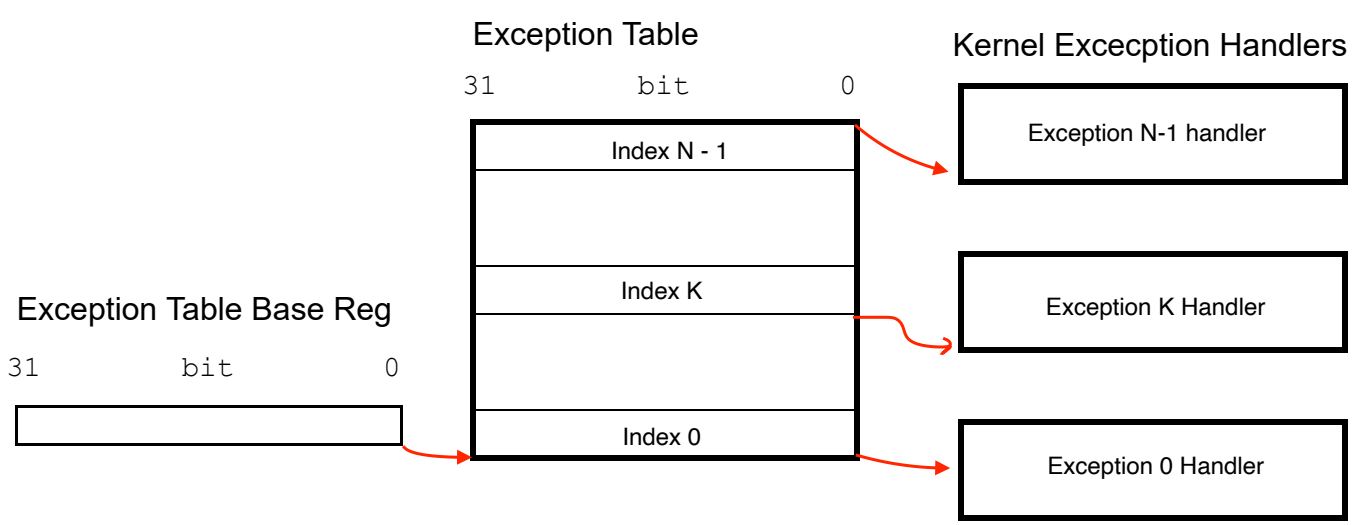
ENUM is for exception number like index

EHA is for exception handler's address like element's value

Exception Table

is a jump table for exceptions that's allocated in memory by the OS on system boot

exception number a unique non-negative integer associate with each type of exception



Copyright © 2016-2022 Jim Skrentny

Exceptions/System Calls in IA-32 & Linux

Don't need to memorize the exception numbers (Maybe 0)

Exception Numbers and Types

0 - 31 are defined by processor

- 0 divide by zero
- 13 protection **fault** — Segfault
- 14 page **fault** — Memory sad
- 18 machine check — Checks Hardware

32 - 255 are defined by OS → 128 ($0x80) **trap** makes system call

You use 128 or 0x80 for the system call

System Calls and Service Numbers

1 exit terminate process
2 fork create a new process
3 read file
4 write file
5 open file
6 close file
11 execve

FILE IO

Making System Calls

1.) put service number in %eax

2.) put sys call args in remaining regs EXCEPT %esp

3.) `int $0x80` trap to start system call handler

System Call Example

```
#include <stdlib.h>
int main(void) {
   write(1, "hello world\n", 12);
   exit(0);
}
```

Both of these are system calls

Assembly Code:

main.s

```
.section .data
string:
   .ascii "hello world\n"
string_end:
   .equ len, string_end - string
.section .text
.global main
main:
```

`.x` are assembler directives

.data section is the data segment of memory

.text section is the text segment of memory

Write

```
movl $4, %eax
movl $1, %ebx
movl $string, %ecx
movl $len, %edx
int $0x80
```

Placing 4 in %eax means that we want syscall 4 to run, which is write
1 is an argument for the syscall, means print to stdout
Place address of start of string in %ecx
Place length (look at string_end to see how its calculated) in %ecx
Do System Call

Exit

```
movl $1, %eax
movl $0, %ebx
int $0x80
```

Placing 1 in %eax means that we call syscall 1, which is exit
0 is an argument for the syscall, means successful exit
Do system call

Copyright © 2016-2022 Jim Skrentny

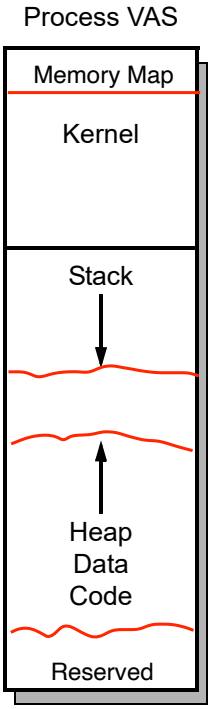
Processes & Context

Recall, a *process*

- is an instance of an executing program
- has *context* the info needed to restart the processes

Why?

It's easier to treat a process as a single entity as if it is running by itself.



Key illusionsprocess exclusively uses

1. CPU
2. Memory
3. Devices

→ Who is the illusionist?

The Operating System

Concurrency (multi-threading, multiprocessing, multitasking)

Combined execution of two or more processes

At the "same" time

scheduler Kernel code to switch execution among processes

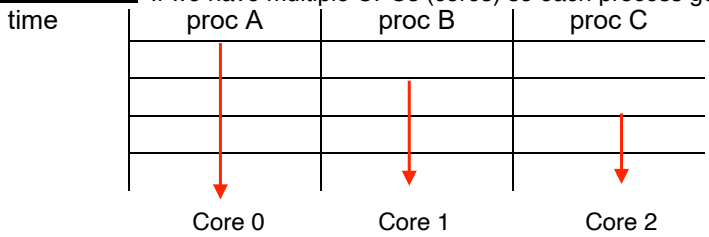
interleaved execution Where one CPU is used among multiple processes

time slice Interval a particular process runs in

The idea is that the processes just take turns over some predefined time interval (can be the same, or be variable)

time	proc A	proc B	proc C
A	Running		
B		Running	
A	Running		
B		Finished	
C			Running
A	Running		

parallel execution If we have multiple CPUs (cores) so each process gets its own core



We can split the proceses on different cores, if we have more processes than cores we can then use interweaved execution to handle this again

Copyright © 2016-2022 Jim Skrentny

User/Kernel Modes

What? Processor *modes* are

different privilege levels a process can run at

mode bit indicates current mode, 1 = kernel/supervisor, 0 = user

kernel mode

Can execute any instruction, access any memory, any device

user mode

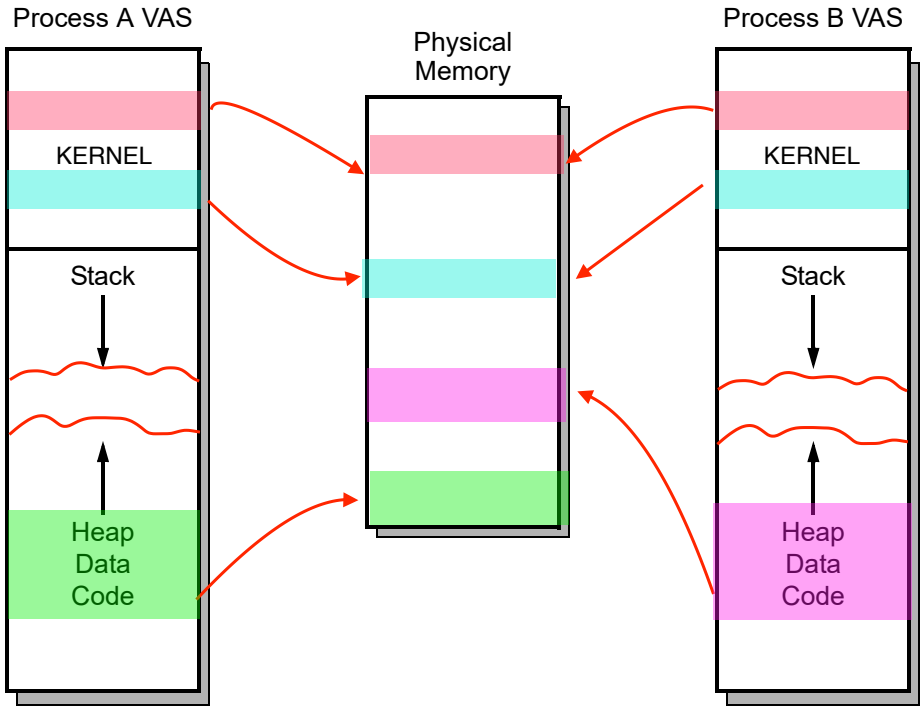
Can execute some instructions, some memory, and can access devices through O.S.

flipping modes

- Start in USER MODE
- Only exceptions can switch to kernel mode (syscalls also fall under this)
- Kernel's exception handler can switch to user mode

Sharing the Kernel

A key part of the Operating system:
+Shared by all user processes
+Memory Resident (Lives in Memory)
Physical memory is only allocated when the program is running, and the entire program is not necessarily physical memory at the same time. The kernel is special in that all programs point the same pages of memory for the kernel, so its not actually duplicated in the VAS like the diagram normally shows.



Copyright © 2016-2022 Jim Skrentny

Context Switch

What? A *context switch*

- is when the OS switches out one running process for another

- requires preservation of Processes context so it can be restored

  1. CPU state
  2. User stack %ebp, %esp
  3. Kernel stack %ebp, %esp
  4. Kernel data structures: Page Table, Process Table, File Table (File System), and others

When? Happens as a result of an exception when kernel needs to execute another process

Why? Allows for exceptions to be processed

How?

1. **save** the context of the current process
2. **restore** the context of the next process
3. **transfer** control to restored process

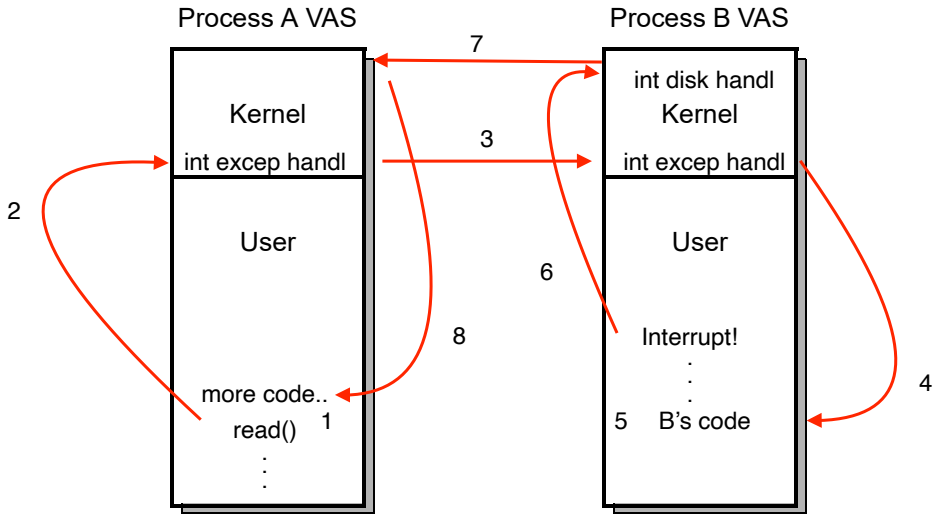
✵ *Context switches are very* EXPENSIVE!!

→ What is the impact of a context switch on the cache?
Really bad cache performance (it makes the cache very sad), "cache pollution" as the new context has no relation to the next process.

Copyright © 2016-2022 Jim Skrentny

Context Switch Example

Stepping through a `read()` System Call



1. In USER mode running process A, and then we hit a syscall for the read() instruction.

2. Switch to KERNEL mode - run exception handler for system call service number 3 (read).
   Arrange Direct Memory Access (DMA) for Disk Controller.

3. In KERNEL mode: Ready to do a context switch to process B.
   Save Process A context, then restore process B context.

4. Switch to USER Mode, running process B.

5. In USER Mode, process B is running.
   Interrupt occurs from Disk Controller - DMA is done for file.

6. Switch to KERNEL mode, run interrupt handler code for disk controller

7. In KERNEL mode, do context save from process B context, restore previous A context.

8. Switch to USER Mode, resume process A.

Copyright © 2016-2022 Jim Skrentny

Meet Signals

✵ *The Kernel uses signals to notify User processes of exceptional events.*

What? A *signal* is

a small message sent

Linux:30 standard signal types each given a unique positive int

`$kill -l`see signal names and numbers

signal(7) **`$man 7 signal`**

Why?

- so Kernel can notify user processes about
  1. Low level hardware exception (0 - 32)
  2. High-Level software events in kernel or user processes
- to enable user processes to communicate with each other and the kernel
- to implement a high-level software form of Exception handling

Examples

1. divide by zero

exception = 0 interrupts to kernel handler

- kernel signals user proc with SIGFPE #8

2. illegal memory reference

exception = 13 interrupts to kernel handler

- kernel signals user proc with SIGSEGV #11

3. keyboard interrupt

- ctrl-c interrupts to kernel handler which signals SIGINT #2

  The foreground (active) process terminates by default.

- ctrl-z interrupts to kernel handler which signals SIGSTP #20

  The foreground process suspends by default.

Copyright © 2016-2022 Jim Skrentny

Three Phases of Signaling

Sending

- when the kernel exception handler runs in response to execution event
- is directed to a destination process.

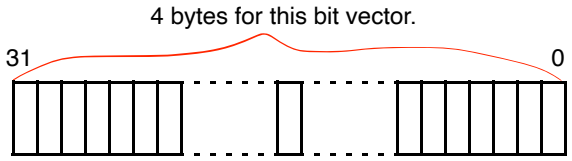
Delivering

when the kernel records a sent signal for its destination process

pending signal is delivered but not received

- each process has a bit vector for recording pending signals

bit vectors are kernel data structures where each bit has a distinct meaning (which signal has occured)



This is part of the context saving portion of context switching.

- Bit k is set to 1 when signal k is delivered.
  Bit k is cleared (zeroed) when signal is recieved.

Receiving

when the kernel causes destination process to react to pending signal

- happens when Kernel transfers control back to process.
- mult pending signals are done in order from low to high signal number

  This can be thought of as a “priority”

blocking prevents a signal from being received

- enables a process to control which signals is pays attention to
- each process has a second bit vector for blocking signal

blocking bit vector (1 = block or ignore signal)

32 bits

Structured like the bit vector above, with 1 saying that we want to block it.

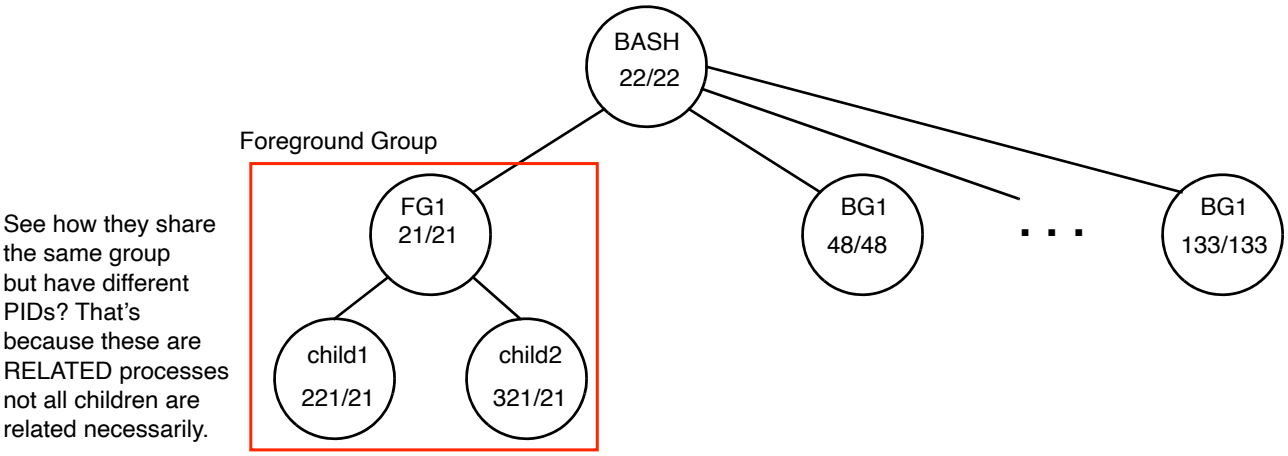
Copyright © 2016-2022 Jim Skrentny

Process IDs and Groups

What? Each process

- is identified by a PID (Process ID Number)
- belongs to exactly 1 group identified by a PGID or a process group ID

PID/PGID is the format we will use.



See how they share the same group but have different PIDs? That's because these are RELATED processes not all children are related necessarily.

Why? Numbers are easier to manage than program names.

How?

Recall:pslists processes

`ps -u` list processes started by user.

`ps -al` list all running processes on this machine.

`jobs` lists process using a single job number

Get process id and get group id for C code.

getpid(2)getpgrp(2) `$man 2` ~~`getgid`~~ getpid

```
#include <sys/types.h>
#include<unistd.h>
```

`pid_t getpid(void)` Returns calling process's PID (its just a number)

`pid_t getpgrp(void)` Returns calling process's PGID (its just a number)

Copyright © 2016-2022 Jim Skrentny

Sending Signals

What? A signal is sent by the kernel or a user process via the kernel

from the command line or in a program using system calls

How? Linux Command From the command line

kill(1)`$man 1 kill`

Signal Number Type

9 is SIGKILL, 2 SIGINT (Control C), 20 SIGTSTR (CONTROL Z), 19 SIGSTOP (Control S), 18 SIGCONT (Control Q)

```
kill -9 <pid>
```

Terminates all processes you can with the -9 flag.

→ What happens if you kill your shell?

Save your work first before you try this, basically just logouts.

How? System Calls From within a program to another process.

kill(2)`$man 2 kill` This gives information of how to use them in a C program.

killpg(2)

```
#include <sys/types.h>
#include<signal.h>

int kill (pid_t pid, int sig)pid of process to signal, sig to send SIGKILL
```

returns 0 on success, -1 on error

Don't use a number for sig, it's recommend to use a predefined macro that matches the signal number.

alarm(2) man 2 alarm

Set alarm that will deliver SIGALRM signal after specified number of seconds.

```
#include <unistd.h>
                    # seconds until alarm signal is sent.
unsigned int alarm(unsigned int seconds)
```

Returns the number of seconds remaining if there was a previous alarm running.

Otherwise returns 0

Copyright © 2016-2022 Jim Skrentny

Receiving Signals

What? A signal is received by its destination process by doing a defined default action or executing code specified by signal handler.

How? Default Actions

- Terminate the process SIGINT #2 (Control C)
- Terminate the process and dump core SIGSEGV #11 (Segfault)
- Stop the process SIGTSTP #20
- Continue the process if it's currently stopped SIGCONT #18
- Ignore the signal SIGWINCH (when the window is changed in size)

How? Signal Handler

1. Code a Signal Handler

   - Looks like a regular function BUT IT IS CALLED BY THE KERNEL (callback)

   - Note: Should not make unsafe system calls.
     AKA don't do printf or writing files because they can cause exceptions (its ok in p6), in general IO instructions bad!

2. Register the Signal Handler

   - To catch one or more signals

   ~~signal(2)~~ DO NOT USE on P6 or CS354
   sigaction(2) POSIX compliant func for examining and changing a signals default action.

Code Example

```
#include <signal.h>   // Struct with Sigaction
#include ...          //General stuff like printf, malloc
#include <string.h>   // For memset

void handler_SIGALRM() { ... }  Signal handler code here (SIGALMRM changes with signal type).
                                This is step 1
int main(...) {
//Step 2 is registering our function
struct sigaction sa;
memset(&sa, 0, sizeof(sigaction)); //sa.sa_flags = 0 now
sa.sa_handler = handler_SIGALRM; //Pass the function pointer.
if(sigaction(SIGALRM, &sa, NULL) != NULL){
    printf("Error binding SIGALRM handler\n");
    exit(1);
}
```

Copyright © 2016-2022 Jim Skrentny

CS 354 - Machine Organization & Programming
Tuesday Dec 6th, and Thursday Dec 8, 2022

https://aefis.wisc.edu Course: CS354 Instructor: DEPPELER

Homework hw7: DUE on or before Monday December 7th

Homework hw8: DUE on or before Monday December 12

Homework hw9: DUE on or before Wednesday December 14

Project p6: Due on last day of classes. ***NOTE: There is no LATE day or OOPS point available for p6. All work must be submitted before 11:59 pm Dec 14th. Please complete p6 this week as labs are very busy last week of classes. If you do plan on getting help during last week of classes, be sure to bring your own laptop in case there is no workstation available.***

Last Week

Kinds of Exceptions Transferring Control via Exception Table Exceptions/System Calls in IA-32 & Linux Processes and Context User/Kernel Modes Context Switch Context Switch Example	Meet Signals Three Phases of Signaling Processes IDs and Groups Sending Signals Receiving Signals

This Week

Issues with Multiple Signals Forward Declaration Multifile Coding Multifile Compilation Makefiles	Relocatable Object Files Static Linking Linker Symbols Linker Symbol Table Symbol Resolution
Next Week: Resolving Globals Symbol Relocation Executable Object File Loader What's next? take OS cs537 as soon as possible and Compilers cs536, too!	**Read:** B&O 7.1 Compiler Drivers 7.2 Static Linking 7.3 Object Files 7.4 Relocatable Object Files 7.5 Symbols and Symbols Tables 7.6 Symbol Resolution 7.7 Relocation

Copyright © 2016-2022 Jim Skrentny

Issues with Multiple Signals

What? Multiple signals of the same type as well as those of different types can be sent during the same period that others are being sent

Some Issues

→ Can a signal handler be interrupted by other signals? Yes, but

Linux signals of same type as running become pending

❋ *Block any signals* sa is sigaction struct

sig emptyset(& sa.sa_mask) // blocks all
sigfillset (& sa.sa_mask) // enables all
sigaddset / sigdelset / sigismember (& sa.sa_mask, signum)

→ Can a system call be interrupted by a signal? Yes, for

<u>slow system calls</u> read-scanf, write-printf

such sys call return immediately with an error

sa.sa_flags = SA_RESTART

→ Does the system queue multiple standard signals of the same type for a process? NO

No, bit vector cannot count due to signals

❋ *Your signal handler shouldn't assume* that a signal was sent only once

Real-time Signals

Linux has 33 additional app defined signals

- They can include integer or pointers in their msg
- Multiple signals of same type are queued in the order delivered
- Multiple signals of different types are recieved (handled) from low to high

Copyright © 2016-2022 Jim Skrentny

Forward Declaration

What? Forward declaration tells compilers certain attr of identifier before it is fully defined

❋ *Recall, C requires that an identifier* be declared before use

Why?

- One-pass compiler (gcc) can ensure identifier exists & is used correctly
- Large programs can be divided into separate functionality which can be independently compile
- mutual recursive A -> B -> A

Declaration vs. Definition

<u>*declaring*</u> tells compiler the name & type of a variable

variables: name & type (extern is used here)

functions: return type, name, arguments & its types

<u>*defining*</u> provides full details

variables: where in memory it is located

functions: function body

example: char c;

❋ *Variable declarations* usually both declare & define

```
void f(){
stack  int i = 11;   // declare, define, initializing
data   static int j; // declare, define
```

❋ *A variable is proceeded with* extern is not defined

```
extern char c;   //This is declared, not defined, not initialized.
                 //Somewhere in the code it SHOULD be defined.
```

Copyright © 2016-2022 Jim Skrentny

Multifile Coding

What? Multifile coding divides program into functional units in different files each with a header & source

Header File (finename.h) - "public" interface

contains things to share
main function declarations
also defines for types, constants & macros

One Definition Rule

recall **heapAlloc.h** from project p3:

```
#ifndef __heapAlloc_h__
#define __heapAlloc_h__

int   initHeap(int sizeOfRegion);
void* allocHeap(int size);
int   freeHeap(void *ptr);
void  dumpMem();

#endif // __heapAlloc_h__
```

public declaration of function

❋ *An identifier* can only be defined only once in global scope (ODR)

#include guard: the ifndef means only one thing is defined with said name

- prevents multiple inclusion of same header file
- prevents linker errors that result from multiple linkages

Source File (filename.c) - "private" implementation

must define things declared in header file
also thing you don't intend to share

recall **heapAlloc.c** from project p3:

```
#include <unistd.h>
. . .
#include "heapAlloc.h"
```

"..." means user defined source

```
typedef struct blockHeader {
   int size_status;
} blockHeader;

blockHeader *heapStart = NULL;

void* allocHeap(int size) { . . . }
int   freeHeap(void *ptr) { . . . }
int   initHeap(int sizeOfRegion) { . . . }
void  dumpMem() {. . . }
```

~~public~~ Private code here.

private function declarations

Multifile Compilation

gcc Compiler Driver

directs all tools needed to build executable from src code

main.c -> preprocessor cpp remove comments & do preprocessor directives
main.i -> compiler cc translates C to ASM
main.s -> assembler as translate ASM to MC (Relocated Obj File)
main.o -> linker ld combines ROF & SOFs to create executable object file

Object Files

contain binary code & data

relocatable object file (ROF)
produced by assembler
can be combined by linker w/ other ROF & SOF to create EOF

executable object file (EOF)
produced by linker
can be loaded by loader into memory & is run

shared object file (SOF)
produced by ASM
can be loaded into memory & linked dynamically during load or runtime

Compiling All at Once

```
gcc align.c heapAlloc.c  -o align
```
EOF

Compiling Separately

```
gcc -c align.c
```
cpp -> cc -> as produces align.o

```
gcc -c heapAlloc.c
```
cpp -> cc -> as produces heapAlloc.o

```
gcc align.o heapAlloc.o -o align
```
ld produces a.out (EOF) align

ROF ROF

❋ *Compiling separately is* more efficient & easier to manage at scale

Copyright © 2016-2022 Jim Skrentny

Makefiles

What? Makefiles are

- text file name Makefile that has rules for making files
- used w/ make command

Why?

- convenient – specify how to build
- efficient – only builds what is necessary

Rules

<target>: <files that target depends on>
\t <commands to make target>

Example

makefile

rules

```
#simplified p3 Makefile <— comment
align: align.o heapAlloc.o
      gcc align.o heapAlloc.o -o align
align.o: align.c
      gcc -c align.c
heapAlloc.o: heapAlloc.c heapAlloc.h
      gcc -c heapAlloc.c
clean:
      rm *.o
      rm align
```

(EOF)
(ROF)
(ROF)

Using

```
$ls
align.c  Makefile  heapAlloc.c  heapAlloc.h
$make
gcc -c align.c
gcc -c heapAlloc.c
gcc align.o heapAlloc.o -o align
$ls
align  align.c  align.o  Makefile  heapAlloc.c  heapAlloc.h  heapAlloc.o
$rm heapAlloc.o
rm: remove regular file 'heapAlloc.o'? y
$make
gcc -c heapAlloc.c
gcc align.o heapAlloc.o -o align
$make heapAlloc.o
make: 'heapAlloc.o' is up to date.
$make clean
rm *.o
rm align
$ls
align.c Makefile heapAlloc.c heapAlloc.h
```

rule 2
rule 3
rule 1
rule #3
rule #1
rule #4

Copyright © 2016-2022 Jim Skrentny

Relocatable Object Files (ROFs)

What? A *relocatable object* file is

- An .o file that contains object code.
  An .o file that contains object code.
- in a format that a linker can only use to create an executable object file (EOF)

Executable and Linkable Format (ELF)

Object file format used by Linux

Index		
0	ELF Header	The header informs us where the sections are, all start at 0x0
1	.text	Machine Code
2	.rodata	String Literals and Jump Tables
3	.data	Initialized Globals and Static Locals
	.bss	Uninitialized or 0 Globals and Static Locals
	.symtab	Linker Symbol Table, globals variables and extern functions
	.rel.text	Code relocations (when we move stuff into files we have to update addresses of code)
	.rel.data	Data relocations (when we move stuff into files we have to update addresses of code)
	.debug	gcc -g: debug symbol table
	.line	maps executable with src code
	.strtab	Table of names used in the relocatable object file
	Section Header Table	SHT

CODE

DATA

ELF Header

Contains general information about the file.

ELF header size, object file type (ROF, EOF, SOF), offset to SHT, size of the section header table & number of entries in it

Also contains the word size, byte ordering, machine type

Section Header Table (SHT)

Location and size of each section in the file.

Copyright © 2016-2022 Jim Skrentny

Static Linking

What? *Static linking* generate a complete EOF without variables or function identifiers.

	static	vs.	dynamic
executable size:	larger		smaller
library code:	smaller		not included (dynamically linked during LOAD or runtime)

How?

All language translation has been done (cpp, cc, as)

Need to only combine ROFs and SOFs together into an EOF

→ What issues arise from combining ROFs?

1. variable and function identifiers must follow the one definition rule (ODR)

   This is called RESOLUTION

2. variable and function identifiers need to be replaced with their addresses

   This is called RELOCATION

Making Things Private

→ Are functions and global variables only in a source file actually private if they're not in the correspondng header file?

No!!! They can still be accessed by other source files that declare it.

→ How do you make them truly private? Make it static

This is because the compiler can just use offsets to use the static variable
Globals that aren't static are truly global, if static they are only global to things inside the file.

Copyright © 2016-2022 Jim Skrentny

Linker Symbols

What?

Symbols are identifiers used for variables and functions in src code.

Linker Symbols are symbols used by the linker.

→ Which kinds of variables need linker symbols? Those allocated in the DATA segment.

Are they in the data segment?

1. local variables on the stack or heap
2. `static` local variables Yes! They are either in the .bss if uninitialized or .data if initialized and have local scope.
3. parameter variables No on the stack
4. global variables Yes, they have true global scope
5. `static` global variables Yes, global scope & "private" to source file, must be relocated.
6. `extern` global variables Yes, global scope, defined elsewhere.

→ Which kinds of functions need linker symbols?

All functions for relocation, likely all of resolution.

1. externed functions
   Linker must connect functions in this ROF with a definition in another ROF

2. Non-static functions - resolution
   Linker may need to connect function in ROF with a function call in another.

3. static functions "private" to source file
   Possibly no resolution required since they're private, but still need to relocate in our EOF.

Copyright © 2016-2022 Jim Skrentny

Linker Symbol Table

What? The linker *symbol table* is

- built by the assembler using symbols exported by the compiler
- represented as an array of ELF_Symbol.

`ELF_Symbol` Data Members and their Use

`int name` Byte offset into .strtab containing null terminated strings

`int value` Symbol's address

if ROF Offset from the start of the section (relative address from section)

if EOF Virtual address or absolute address

`int size` Number of bytes needed to allocate this symbol

`char type:4` Data (OBJECT), Function (FUNC), extern (NOTYPE)

`binding:4` Global or Local (to this source file)

`char section` Index into the section header table (SHT)

pseudo sections:

ABS:olute The symbol should not be relocated

UND:efined If the symbol was declared as extern

COM:mon Symbols that are uninitialized or zero

`value` Now our value needs its alignment

`size` Now we need the minimum size needed for this symbol

Example

```
Num:  Value Size  Type    Bind    Ot Ndx  Name
1 - 7 not shown
8:    0     4     OBJECT  GLOBAL  0  3    bufp0
9:    0     0     NOTYPE  GLOBAL  0  UND  buf
10:   0     39    FUNC    GLOBAL  0  1    swap
11:   4     4     OBJECT  GLOBAL  0  COM  bufp1
```

→ Is `bufp0` initialized? Yes, look at the ELF table and go to index 3 (0 is the ELF HEADER) and can see its in .data, which means it's an initialized static local or global var.

→ Was `buf` defined in the source file or declared `extern`? It's extern as type is NOTYPE and no section header

→ What is the function's name? Only one function, and it's name is swap

→ What is the alignment and size of `bufp1`? COM means its in the .bss section and value = alignment so 4 bytes and size = min 4 bytes.

Copyright © 2016-2022 Jim Skrentny

Symbol Resolution

(Make sure we follow the One Definition Rule)

What? *Symbol resolution*

- Checks ODR across one or multiple EOFs
  In one ROF, the compiler can do the resolution, across multiple then the linker has to do it.

- Work is divided between compiler and linker.

Compiler's Resolution Work

Resolves local symbols in one src file.

- locals

  `static` locals Ensures each has a unique name for linker.

  Might name them like x.add, y.swap, etc. by appending the function or scope name to the variable

- globals Leave for the linker.

  `static` globals Can check ODR, since private to file.

EXTERN

✵ *If a global symbol is only declared in this source file*

the compiler assumes that it is defined in another ROF (externed variable)

Linker's Resolution Work

Resolve global symbols across multiple EOFs

- `static` locals Linker doesn't resolve these, but it does relocate these.

- globals Check the one definition rule on each symbol across all EOFs

✵ *If a global symbol is not defined or is multiply defined* that is a LINKER error.

Copyright © 2016-2022 Jim Skrentny

CS 354 - Machine Organization & Programming
Tuesday December 13th, 2022

Course Evals

https://aefis.wisc.edu Course: CS354 Instructor: DEPPELER

Final Exam -Wednesday Dec 21st, 7:25 PM - 9:25 PM

Your final exam room will be randomly assigned and sent to you via email.
You must attend the exam room as assigned in the email you receive.
Arrive early if possible with UW ID and #2 pencils. See additional exam info on course web site.

All office hours, TA consulting, and Peer Mentoring end on Wed December 14th

Homework hw8: DUE on or before Monday Dec 12

Homework hw9: DUE on or before Wednesday Dec 14 (NO LATE DAY)

Project p6: Due on last day of classes (NO LATE DAY or OOPS PERIOD). If you plan on getting help in labs, be sure to bring your own laptop in case there is no workstation available.

Last Week

Issues with Multiple Signals Forward Declaration Multifile Coding Multifile Compilation Makefiles	Relocatable Object Files Static Linking Linker Symbols Linker Symbol Table Symbol Resolution

This Week

Resolving Globals Symbol Relocation Executable Object File Loader What's next? take OS cs537 as soon as possible and Compilers cs536, too!	
Next Week: FINAL EXAM Watch your email for your exam room assignment. All students must take the final exam in their assigned final exam room. **Students with accomodations should receive email with exam date/time/venue by 12/13.**	

Copyright © 2016-2022 Jim Skrentny

Resolving Globals

Confusing Globals

despite defaults to zero, no assignment means uninit

Linker Error! Multiple definitions of m, n,

main.c

```
int m;
int n = 11;
short o;

extern int x;
int y;
static int z = 66;

//code continues...
```

fun1.c

```
int m = 22;
int n;
int o;

int x;
static int y = 33;
static int z = 77;

//code continues...
```

fun2.c

```
int m;
extern int n;
char o;

static int x = 33;
static int y;
int z;

//code continues...
```

gives def to extern

Strong and Weak Symbols

strong: function definition & init global vars

weak: function declaration & uninit global vars

→ Which code statements above correspond to strong symbols?

Rules for Resolving Globals

→ Which code statements above correspond to definitions?

Recall: 'extern' is declaration only, rest are definition (above)

1. Multiple ~~strong~~ symbols definition in public global sope are not allowed unless you use linker option -z muldefs
2. Given one strong symbol and one or more weak symbols,
3. Given only weak symbols, declare weak symbols w/ extern to comply w/ standard

linker choose first definition found

- *Use* `extern` *to clearly indicate when* a global is declaration only
- *Use* `static` *to clearly indicate when* a global var is private

Copyright © 2016-2022 Jim Skrentny

Symbol Relocation

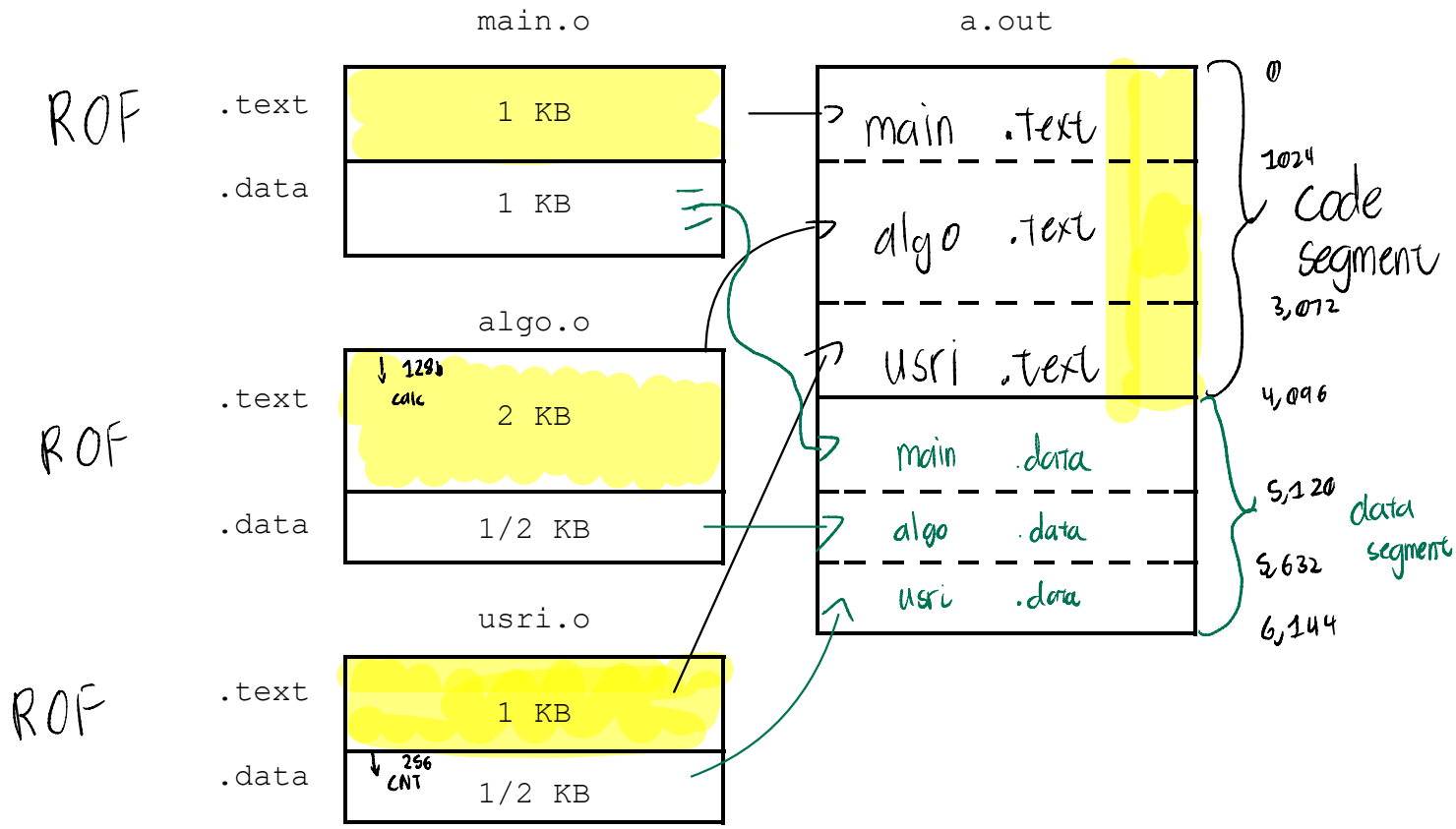
What? *Symbol relocation* combine ROFs & SOFs so that addresses can be determined for remaining symbol in code

How?

1. Merges the same sections of ROFs into one aggregate for each section type
2. Assigns virtual addresses to each aggregate section & each symbol data
3. Updates symbol references listed in ROFs & relocate sections w/ their virtual address

Example

Consider the .text and .data sections of 3 object files below combined into an executable:



address = 1. start address of section +
2. offset to subsection +
3. offset within subsection +

$$\begin{array}{r} 0 \\ +\ 1024 \\ +\ 128 \\ \hline 1152 \end{array}$$ bytes

VAS for start of calc

$$\begin{array}{r} 4096 \\ +\ 1536 \\ +\ 256 \\ \hline 5888 \end{array}$$ bytes

VAS for start of CNT

Copyright © 2016-2022 Jim Skrentny

Excutable Object File (EOF)

What? An EOF, like an ROF, is a file containing object code
it is produced by linker & can be loaded into memory & run

Executable and Linkable Format

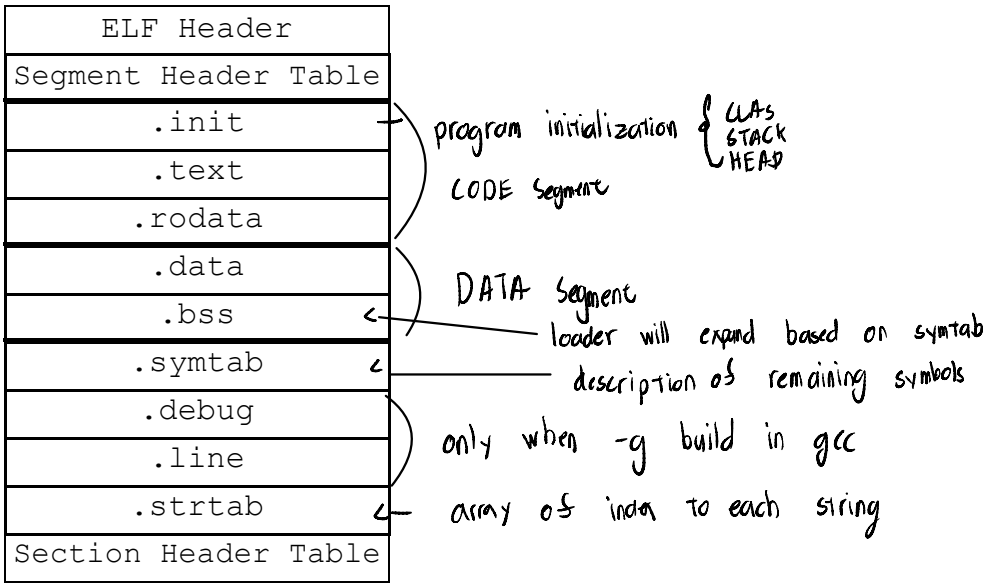
Same as ROF w/ extra section & info

ELF Header

+ Entry point – address of 1st instruction

+ Segment Header Table

Info for each segment to be loaded into memory
offset for file
alignment
page size
size (bytes) in file, size in memory
runtime permissions (r, w, x)



→ Why aren't there relocation sections (.rel.text or .rel.data) in EOF?

All symbols replaced with their address
since we assume static linking

➢ Why is the data segment's size in memory larger than its size in the EOF?

at load-time, .bss is expanded to have space in memory

Copyright © 2016-2022 Jim Skrentny

Loader

What? The *loader*

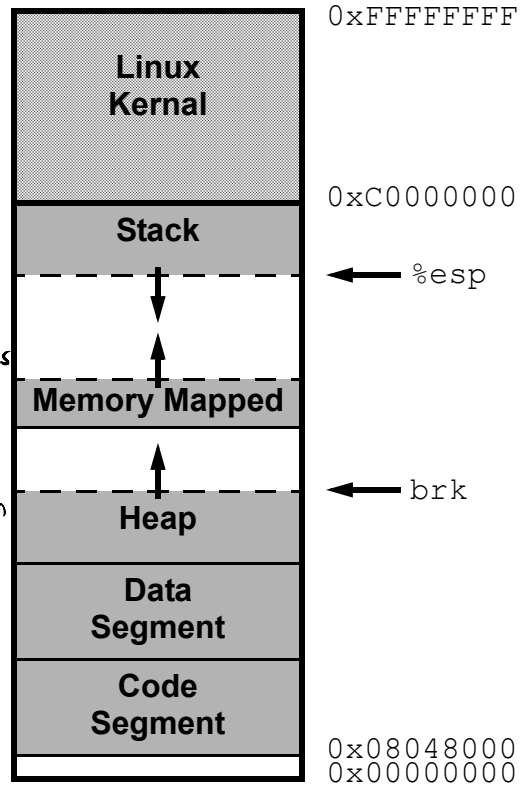
- Kernel code to start program execution
- can be invoked by any Linux program

Loading

1. Copy CODE + DATA segment from EOF into memory
2. Start program by jumping to entry point

Execution - the final story

1. shell creates child with fork()
2. child process invokes the loader w/ execve
3. loader creates a new runtime image
   a. delete current segment
   b. creates current segment
   c. heap & stack (size = 0)
   d. EOFs code & data segments are mapped into page sized chunks but not copy to memory
4. loader jumps to start address, causing page fault, which then copies page into memory




```
setup    call __libc_init_first
         call _init
         call atexit
program  call main
clean up call _exit
```

CS 354 - Machine Organization & Programming
Tuesday December 13th, 2022

Course Evals

https://aefis.wisc.edu Course: CS354 Instructor: DEPPELER

Final Exam -Wednesday Dec 21st, 7:25 PM - 9:25 PM

Your final exam room will be randomly assigned and sent to you via email.
You must attend the exam room as assigned in the email you receive.
Arrive early if possible with UW ID and #2 pencils. See additional exam info on course web site.

All office hours, TA consulting, and Peer Mentoring end on Wed December 14th

Homework hw8: DUE on or before Monday Dec 12

Homework hw9: DUE on or before Wednesday Dec 14 (NO LATE DAY)

Project p6: Due on last day of classes (NO LATE DAY or OOPS PERIOD). If you plan on getting help in labs, be sure to bring your own laptop in case there is no workstation available.

Last Week

Issues with Multiple Signals Forward Declaration Multifile Coding Multifile Compilation Makefiles	Relocatable Object Files Static Linking Linker Symbols Linker Symbol Table Symbol Resolution

This Week

Resolving Globals Symbol Relocation Executable Object File Loader What's next? take OS cs537 as soon as possible and Compilers cs536, too!	
Next Week: FINAL EXAM Watch your email for your exam room assignment. All students must take the final exam in their assigned final exam room. **Students with accomodations should receive email with exam date/time/venue by 12/13.**	

Copyright © 2016-2022 Jim Skrentny

Resolving Globals

Confusing Globals

main.c

```
int m;
int n = 11;
short o;

extern int x;
int y;
static int z = 66;

//code continues...
```

fun1.c

```
int m = 22;
int n;
int o;

int x;
static int y = 33;
static int z = 77;

//code continues...
```

fun2.c

```
int m;
extern int n;
char o;

static int x = 33;
static int y;
int z;

//code continues...
```

Strong and Weak Symbols

strong:

weak:

→ Which code statements above correspond to strong symbols?

Rules for Resolving Globals

→ Which code statements above correspond to definitions?

1. Multiple strong symbols

2. Given one strong symbol and one or more weak symbols,

3. Given only weak symbols,

- *Use `extern` to clearly indicate when*
- *Use `static` to clearly indicate when*

Copyright © 2016-2022 Jim Skrentny

Symbol Relocation

What? *Symbol relocation*

How?

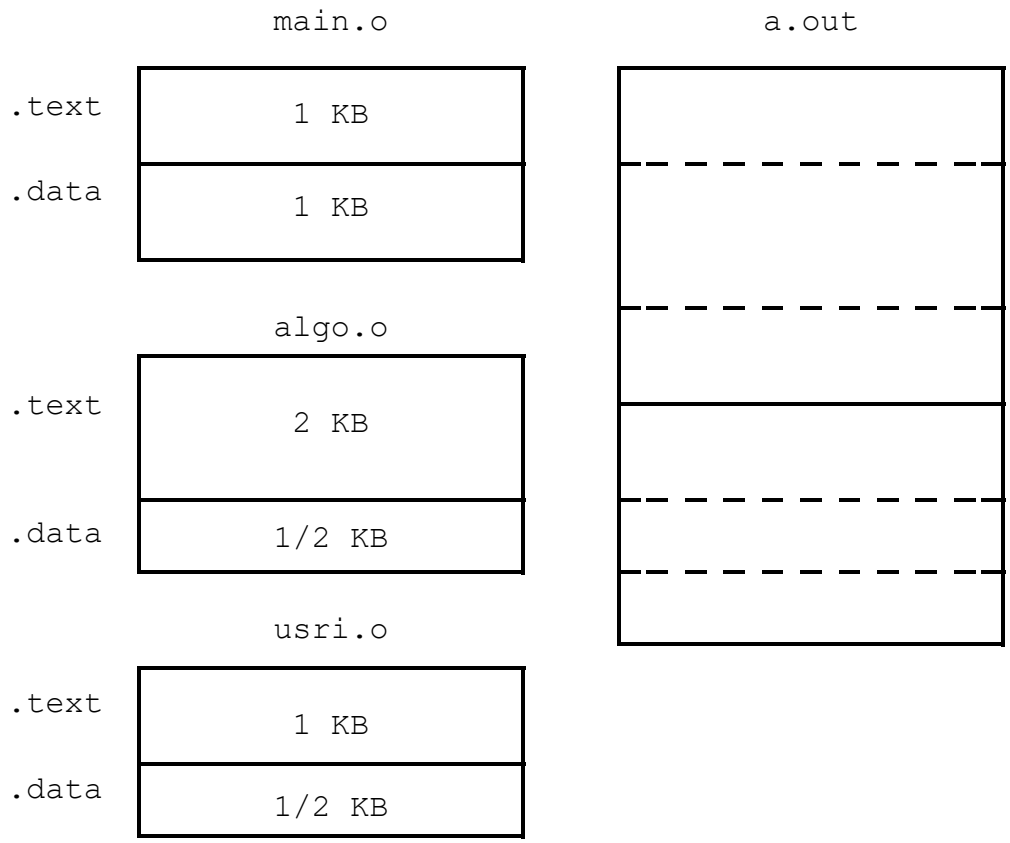
1. Merges the same sections

2. Assigns virtual addresses

3. Updates symbol references

Example

Consider the .text and .data sections of 3 object files below combined into an executable:



address =

Copyright © 2016-2022 Jim Skrentny

Excutable Object File (EOF)

What? An EOF, like an ROF, is

Executable and Linkable Format

ELF Header

+ Segment Header Table

ELF Header
Segment Header Table
.init
.text
.rodata
.data
.bss
.symtab
.debug
.line
.strtab
Section Header Table

→ Why aren't there relocation sections (.rel.text or .rel.data) in EOF?

➢ Why is the data segment's size in memory larger than its size in the EOF?

Copyright © 2016-2022 Jim Skrentny

Loader

What? The *loader*

-
-

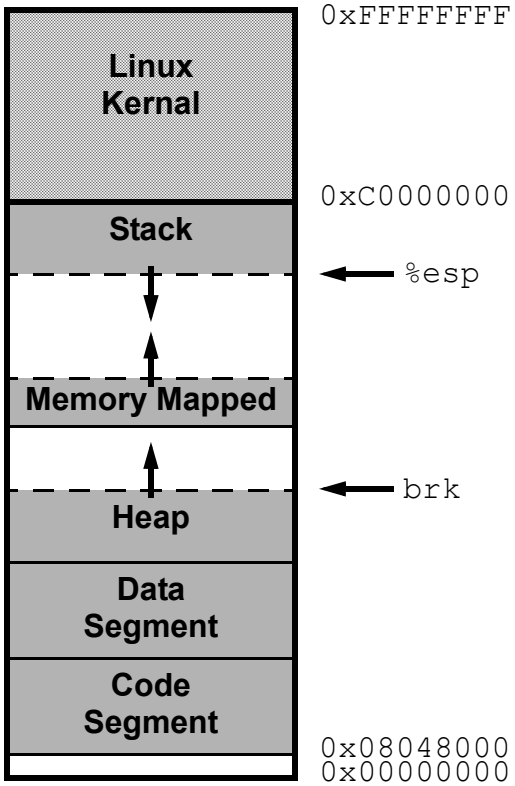
Loading

1.
2.

Execution - the final story

1. shell
2. child process
3. loader
   a.
   b.
   c.
   d.
4. loader

```
call __libc_init_first
call _init
call atexit
call main
call _exit
```



Copyright © 2016-2022 Jim Skrentny

CS 354 - Machine Organization & Programming
Tuesday December 13th, 2022

Course Evals

https://aefis.wisc.edu Course: CS354 Instructor: DEPPELER

Final Exam -Wednesday Dec 21st, 7:25 PM - 9:25 PM

Your final exam room will be randomly assigned and sent to you via email.
You must attend the exam room as assigned in the email you receive.
Arrive early if possible with UW ID and #2 pencils. See additional exam info on course web site.

All office hours, TA consulting, and Peer Mentoring end on Wed December 14th

Homework hw8: DUE on or before Monday Dec 12

Homework hw9: DUE on or before Wednesday Dec 14 (NO LATE DAY)

Project p6: Due on last day of classes (NO LATE DAY or OOPS PERIOD). If you plan on getting help in labs, be sure to bring your own laptop in case there is no workstation available.

Last Week

Issues with Multiple Signals Forward Declaration Multifile Coding Multifile Compilation Makefiles	Relocatable Object Files Static Linking Linker Symbols Linker Symbol Table Symbol Resolution

This Week

Resolving Globals Symbol Relocation Executable Object File Loader What's next? take OS cs537 as soon as possible and Compilers cs536, too!	
Next Week: FINAL EXAM Watch your email for your exam room assignment. All students must take the final exam in their assigned final exam room. **Students with accomodations should receive email with exam date/time/venue by 12/13.**	

Copyright © 2016-2022 Jim Skrentny

Resolving Globals

Confusing Globals

main.c

```
int m;
int n = 11;
short o;

extern int x;
int y;
static int z = 66;

//code continues...
```

fun1.c

```
int m = 22;
int n;
int o;

int x;
static int y = 33;
static int z = 77;

//code continues...
```

fun2.c

```
int m;
extern int n;
char o;

static int x = 33;
static int y;
int z;

//code continues...
```

Strong and Weak Symbols

strong:

weak:

→ Which code statements above correspond to strong symbols?

Rules for Resolving Globals

→ Which code statements above correspond to definitions?

1. Multiple strong symbols

2. Given one strong symbol and one or more weak symbols,

3. Given only weak symbols,

- *Use `extern` to clearly indicate when*
- *Use `static` to clearly indicate when*

Copyright © 2016-2022 Jim Skrentny

Symbol Relocation

What? *Symbol relocation*

How?

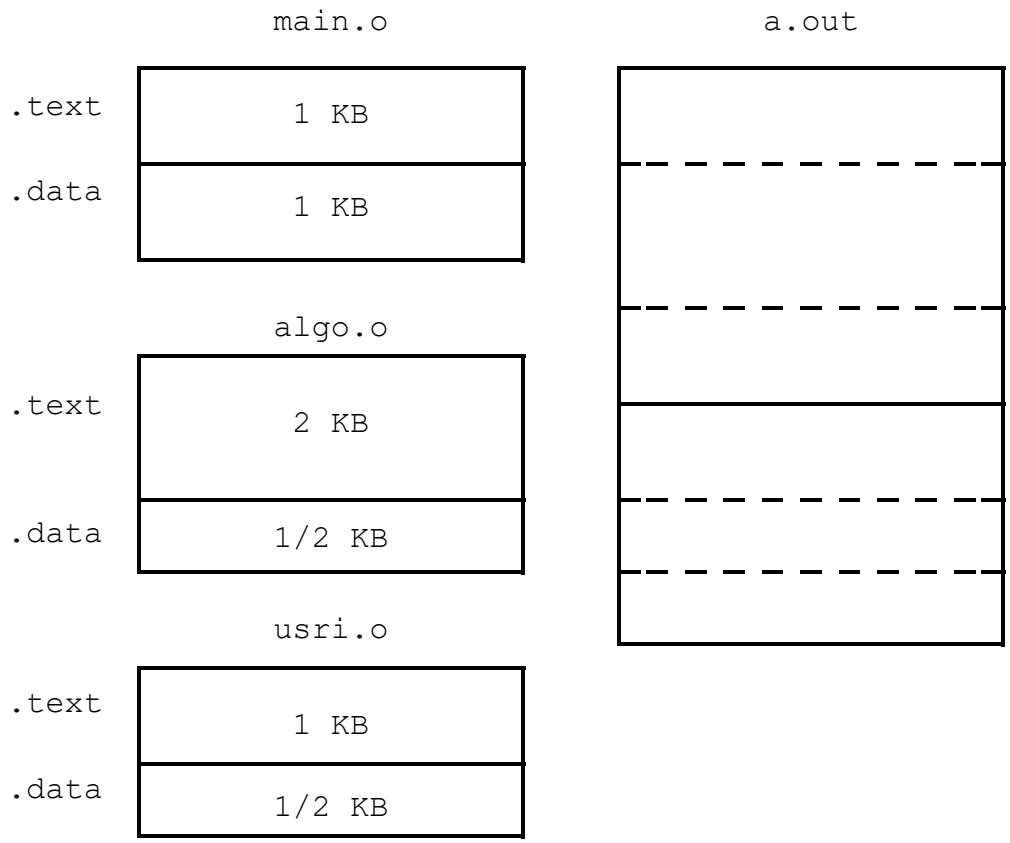
1. Merges the same sections

2. Assigns virtual addresses

3. Updates symbol references

Example

Consider the .text and .data sections of 3 object files below combined into an executable:



address =

Copyright © 2016-2022 Jim Skrentny

Excutable Object File (EOF)

What? An EOF, like an ROF, is

Executable and Linkable Format

ELF Header

+ Segment Header Table

ELF Header
Segment Header Table
.init
.text
.rodata
.data
.bss
.symtab
.debug
.line
.strtab
Section Header Table

→ Why aren't there relocation sections (.rel.text or .rel.data) in EOF?

➢ Why is the data segment's size in memory larger than its size in the EOF?

Copyright © 2016-2022 Jim Skrentny

Loader

What? The *loader*

-
-

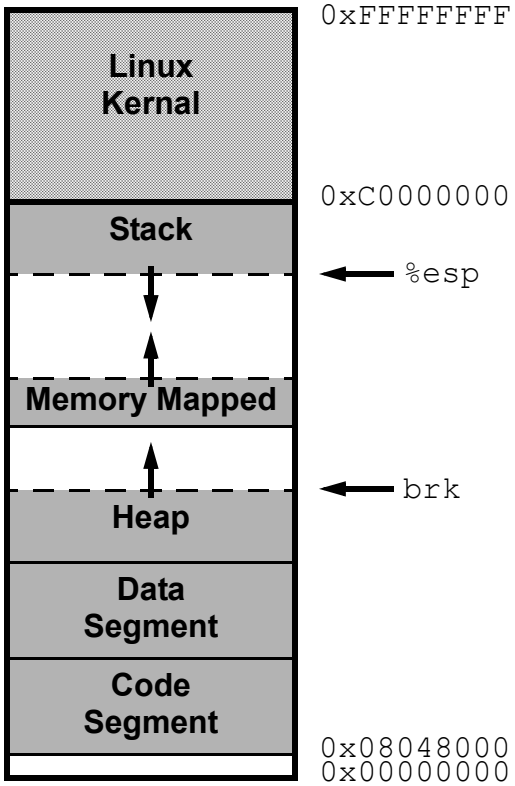
Loading

1.
2.

Execution - the final story

1. shell
2. child process
3. loader
   a.
   b.
   c.
   d.
4. loader

```
call __libc_init_first
call _init
call atexit
call main
call _exit
```



Copyright © 2016-2022 Jim Skrentny