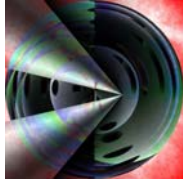


Lecture #6: Intro to MIPS



2005-06-28

Andy Carle



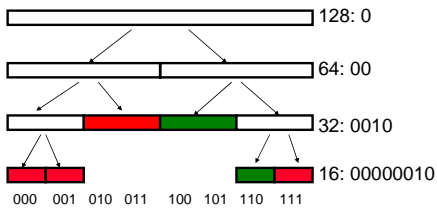
Review

- Several techniques for managing heap w/ malloc/free: best-, first-, next-fit, slab, buddy
- 2 types of memory fragmentation: internal & external; all suffer from some kind of frag.
- Each technique has strengths and weaknesses, none is definitively best
- Automatic memory management relieves programmer from managing memory.
 - All require help from language and compiler
 - Reference Count: not for circular structures
 - Mark and Sweep: complicated and slow, works
- Copying: move active objects back and forth



Buddy System Review

- Legend: FREE ALLOCATED SPLIT



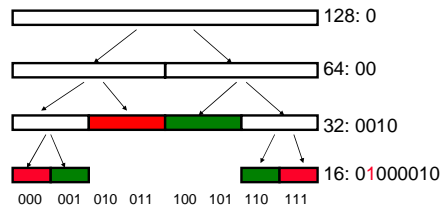
Initial State → Free(001) → Free(000) → Free(111) → Malloc(16)



Kudos to Kurt Meinz for these fine slides

Buddy System

- Legend: FREE ALLOCATED SPLIT

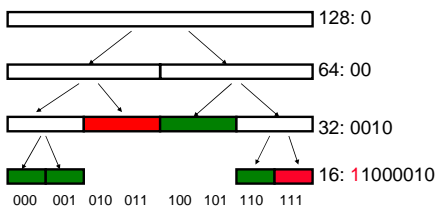


Initial State → Free(001) → Free(000) → Free(111) → Malloc(16)



Buddy System

- Legend: FREE ALLOCATED SPLIT

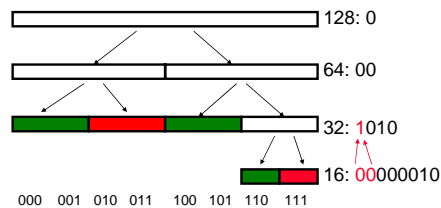


Initial State → Free(001) → Free(000) → Free(111) → Malloc(16)



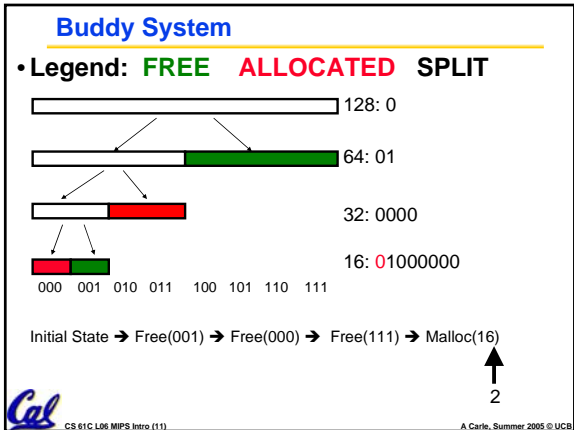
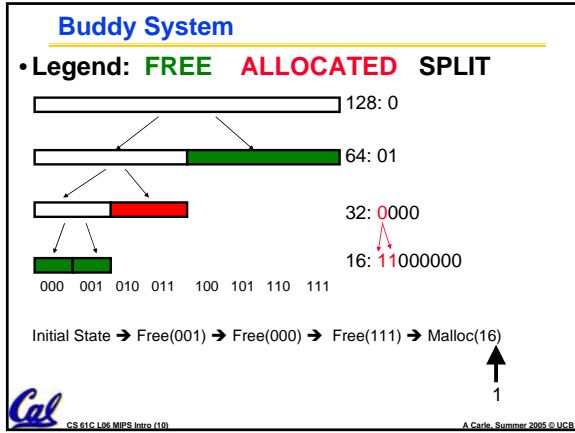
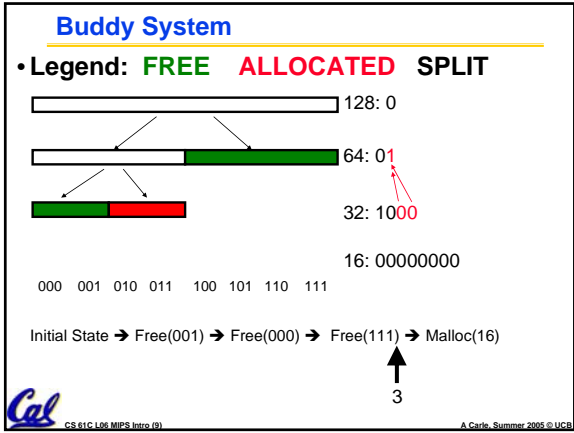
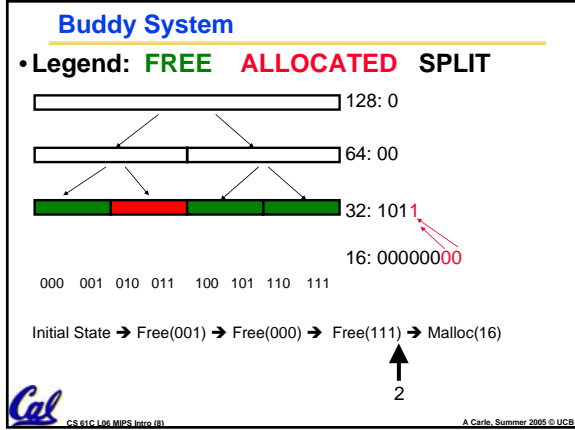
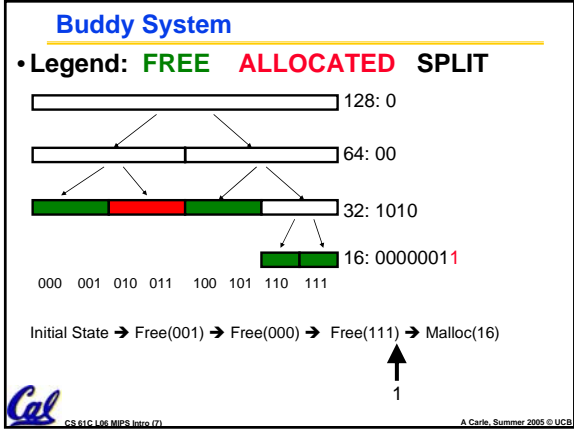
Buddy System

- Legend: FREE ALLOCATED SPLIT



Initial State → Free(001) → Free(000) → Free(111) → Malloc(16)





New Topic!

MIPS Assembly Language

Cal CS 61C L06 MIPS Intro (12) A Carlo, Summer 2005 © UCB

Assembly Language

- Basic job of a CPU: execute lots of *instructions*.
- Instructions are the primitive operations that the CPU may execute.
- Different CPUs implement different sets of instructions. The set of instructions a particular CPU implements is an *Instruction Set Architecture (ISA)*.
 - Examples: Intel 80x86 (Pentium 4), IBM/Motorola PowerPC (Macintosh), MIPS, Intel IA64, ...



CS 61C L06 MIPS Intro (13)

A. Carle, Summer 2005 © UC Berkeley

Instruction Set Architectures

- Early trend was to add more and more instructions to new CPUs to do elaborate operations
 - VAX architecture had an instruction to multiply polynomials!
- RISC philosophy (Cocke, IBM, Patterson, Hennessy, 1980s) – Reduced Instruction Set Computing
 - Keep the instruction set small and simple, makes it easier to build fast hardware.
 - Let software do complicated operations by composing simpler ones.



CS 61C L06 MIPS Intro (14)

A. Carle, Summer 2005 © UC Berkeley

ISA Design

- Must Run Fast In Hardware → Eliminate sources of complexity.

Software

- Symbolic Lookup → fixed var names/#
- Strong typing → No Typing
- Nested expressions → Fixed format Inst
- Many operators → small set of insts

Hardware



CS 61C L06 MIPS Intro (15)

A. Carle, Summer 2005 © UC Berkeley

MIPS Architecture

- MIPS – semiconductor company that built one of the first commercial RISC architectures
- We will study the MIPS architecture in some detail in this class (also used in upper division courses CS 152, 162, 164)
- Why MIPS instead of Intel 80x86?
 - MIPS is simple, elegant. Don't want to get bogged down in gritty details.
 - MIPS widely used in embedded apps, x86 little used in embedded, and more embedded computers than PCs



Most HP LaserJet workgroup printers are driven by MIPS-based™ 64-bit processors.



CS 61C L06 MIPS Intro (16)

A. Carle, Summer 2005 © UC Berkeley

Assembly Variables: Registers (1/4)

- Unlike HLL like C or Java, assembly cannot use variables
 - Why not? Keep Hardware Simple
- Assembly Operands are registers
 - limited number of special locations built directly into the hardware
 - operations can only be performed on these!
- Benefit: Since registers are directly in hardware, they are very fast (faster than 1 billionth of a second)



CS 61C L06 MIPS Intro (17)

A. Carle, Summer 2005 © UC Berkeley

Assembly Variables: Registers (2/4)

- Drawback: Since registers are in hardware, there are a predetermined number of them
 - Solution: MIPS code must be very carefully put together to efficiently use registers
- 32 registers in MIPS
 - Why just 32? **Smaller is faster**
- Each MIPS register is 32 bits wide
 - Groups of 32 bits called a word in MIPS



CS 61C L06 MIPS Intro (18)

A. Carle, Summer 2005 © UC Berkeley

Assembly Variables: Registers (3/4)

- Registers are numbered from 0 to 31
- Each register can be referred to by number or name
- Number references:
\$0, \$1, \$2, ... \$30, \$31



CS 61C L06 MIPS Intro (19)

A. Carls, Summer 2005 © UC Berkeley

Assembly Variables: Registers (4/4)

- By convention, each register also has a name to make it easier to code
- For now:
 - \$16 - \$23 → \$s0 - \$s7
(correspond to C variables)
 - \$8 - \$15 → \$t0 - \$t7
(correspond to temporary variables)Later will explain other 16 register names
- In general, use names to make your code more readable



CS 61C L06 MIPS Intro (20)

A. Carls, Summer 2005 © UC Berkeley

C, Java variables vs. registers

- In C (and most High Level Languages) variables declared first and given a type
 - Example:

```
int fahr, celsius;
char a, b, c, d, e;
```
- Each variable can ONLY represent a value of the type it was declared as (cannot mix and match int and char variables).
- In Assembly Language, the registers have no type; operation determines how register contents are treated



CS 61C L06 MIPS Intro (21)

A. Carls, Summer 2005 © UC Berkeley

Comments in Assembly

- Another way to make your code more readable: comments!
- Hash (#) is used for MIPS comments
 - anything from hash mark to end of line is a comment and will be ignored
- Note: Different from C.
 - C comments have format
/* comment */
so they can span many lines



CS 61C L06 MIPS Intro (22)

A. Carls, Summer 2005 © UC Berkeley

Assembly Instructions

- In assembly language, each statement (called an **Instruction**), executes exactly one of a short list of simple commands
- Unlike in C (and most other High Level Languages), each line of assembly code contains at most 1 instruction
- Instructions are related to operations (=, +, -, *, /) in C or Java



CS 61C L06 MIPS Intro (23)

A. Carls, Summer 2005 © UC Berkeley

MIPS Addition and Subtraction (1/4)

- Syntax of Instructions:
“<op> <dest> <src1> <src2>”
where:
 - op) operation by name
 - dest) operand getting result (“destination”)
 - src1) 1st operand for operation (“source1”)
 - src2) 2nd operand for operation (“source2”)
- Syntax is rigid:
 - 1 operator, 3 operands
 - Why? **Keep Hardware simple via regularity**



CS 61C L06 MIPS Intro (24)

A. Carls, Summer 2005 © UC Berkeley

Addition and Subtraction of Integers (2/4)

• Addition in Assembly

• Example: `add $s0,$s1,$s2` (in MIPS)

Equivalent to: `s0 = s1 + s2` (in C)

where MIPS registers `$s0, $s1, $s2` are associated with C variables `s0, s1, s2`

• Subtraction in Assembly

• Example: `sub $s3,$s4,$s5` (in MIPS)

Equivalent to: `d = e - f` (in C)

where MIPS registers `$s3, $s4, $s5` are associated with C variables `d, e, f`



Addition and Subtraction of Integers (3/4)

• How does the following C statement?

```
a = b + c + d - e;
```

• Break into multiple instructions

```
add $t0, $s1, $s2 # temp = b + c
```

```
add $t0, $t0, $s3 # temp = temp + d
```

```
sub $s0, $t0, $s4 # a = temp - e
```

• Notice: A single line of C may break up into several lines of MIPS.

• Notice: Everything after the hash mark on each line is ignored (comments)



Addition and Subtraction of Integers (4/4)

• How do we do this?

```
f = (g + h) - (i + j);
```

• Use intermediate temporary register

```
add $t0,$s1,$s2 # temp = g + h
```

```
add $t1,$s3,$s4 # temp = i + j
```

```
sub $s0,$t0,$t1 # f=(g+h)-(i+j)
```



Immediates

• Immediates are numerical constants.

• They appear often in code, so there are special instructions for them.

• Add Immediate:

```
addi $s0,$s1,10 (in MIPS)
```

```
f = g + 10 (in C)
```

where MIPS registers `$s0, $s1` are associated with C variables `f, g`

• Syntax similar to `add` instruction, except that last argument is a number instead of a register.



Immediates

• There is no Subtract Immediate in MIPS: Why?

• Limit types of operations that can be done to absolute minimum

• if an operation can be decomposed into a simpler operation, don't include it

• `addi ..., -X = subi ..., X => so no subi`

• `addi $s0,$s1,-10` (in MIPS)

```
f = g - 10 (in C)
```

where MIPS registers `$s0, $s1` are associated with C variables `f, g`



Register Zero

• One particular immediate, the number zero (0), appears very often in code.

• So we define register zero (`$0` or `$zero`) to always have the value 0; eg

```
add $s0,$s1,$zero (in MIPS)
```

```
f = g (in C)
```

where MIPS registers `$s0, $s1` are associated with C variables `f, g`

• defined in hardware, so an instruction

```
add $zero,$zero,$s0
```



will not do anything!

Peer Instruction

- A. **Types** are associated with **declaration in C** (normally), but are associated with **instruction (operator)** in MIPS.
- B. Since there are only **8 local (\$s)** and **8 temp (\$t) variables**, we can't write MIPS for C exprs that contain > 16 vars.
- C. If **p** (stored in \$s0) were a pointer to an array of ints, then **p++**; would be `addi $s0 $s0 1`



"And in Conclusion..."

- **In MIPS Assembly Language:**
 - Registers replace C variables
 - One Instruction (simple operation) per line
 - Simpler is Better
 - Smaller is Faster
- **New Instructions:**
add, addi, sub
- **New Registers:**
C Variables: \$s0 - \$s7
Temporary Variables: \$t0 - \$t9
Zero: \$zero

