

CS61C : Machine Structures

Lecture 2.1.2 Garbage Collection & Intro to MIPS

2004-06-29

Kurt Meinz

`inst.eecs.berkeley.edu/~cs61c`



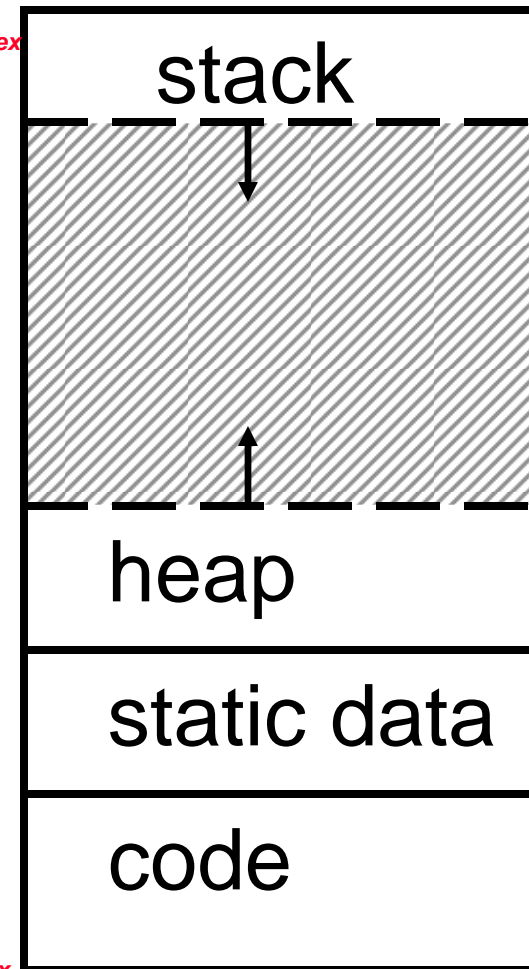
Lecture Outline

- **Buddy System Allocator**
- **Garbage Collection**
- **MIPS**



Memory Management (2/2)

- A program's **address space** contains 4 regions:
 - **stack**: proc frames, grows downward
 - **heap**: space requested for pointers via `malloc()` ; resizes dynamically, grows upward
 - **static data**: variables declared outside main, does not grow or shrink
 - **code**: loaded when program starts, does not change



For now, OS somehow prevents accesses between stack and heap (gray hash lines). Wait for virtual memory



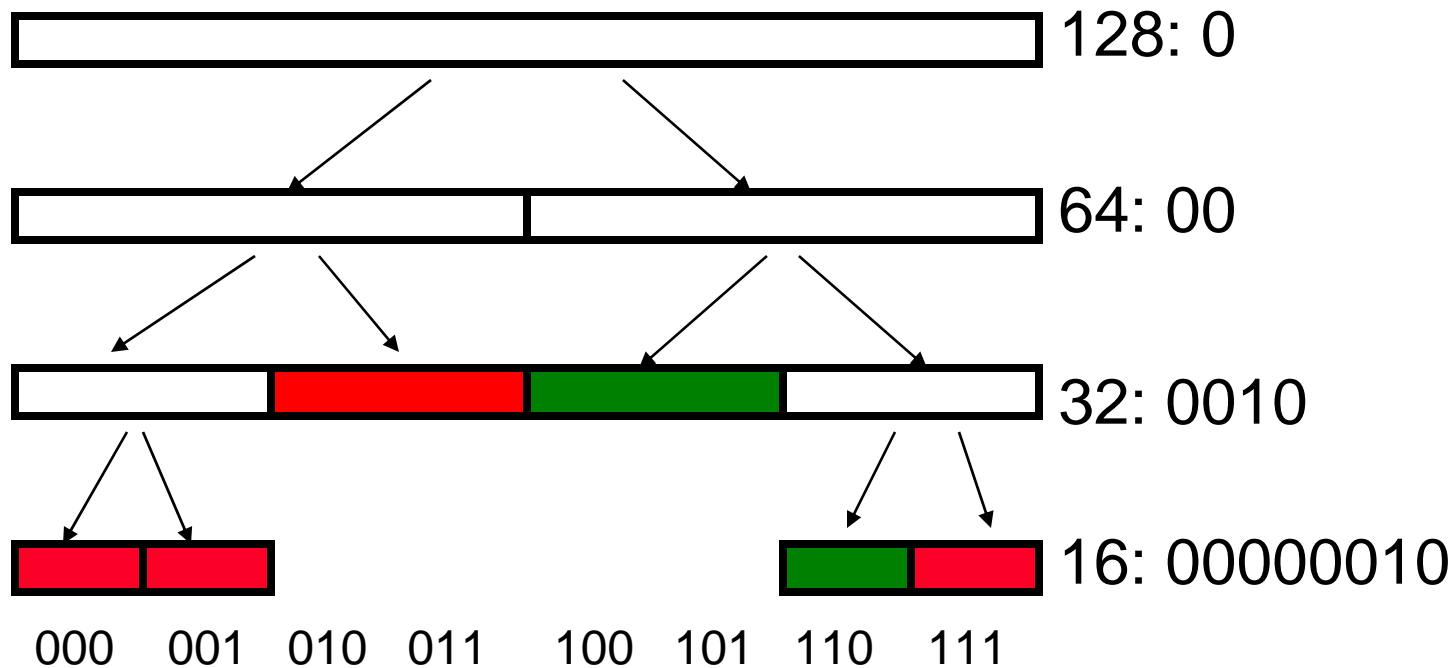
Buddy System

- Yet another memory management technique (used in Linux kernel)
- Like GNU's "slab allocator", but only allocate blocks in sizes that are powers of 2 (internal fragmentation is possible)
- Keep separate free lists for each size
 - e.g., separate free lists for 16 byte, 32 byte, 64 byte blocks, etc.



Buddy System

- Legend: **FREE** **ALLOCATED** **SPLIT**

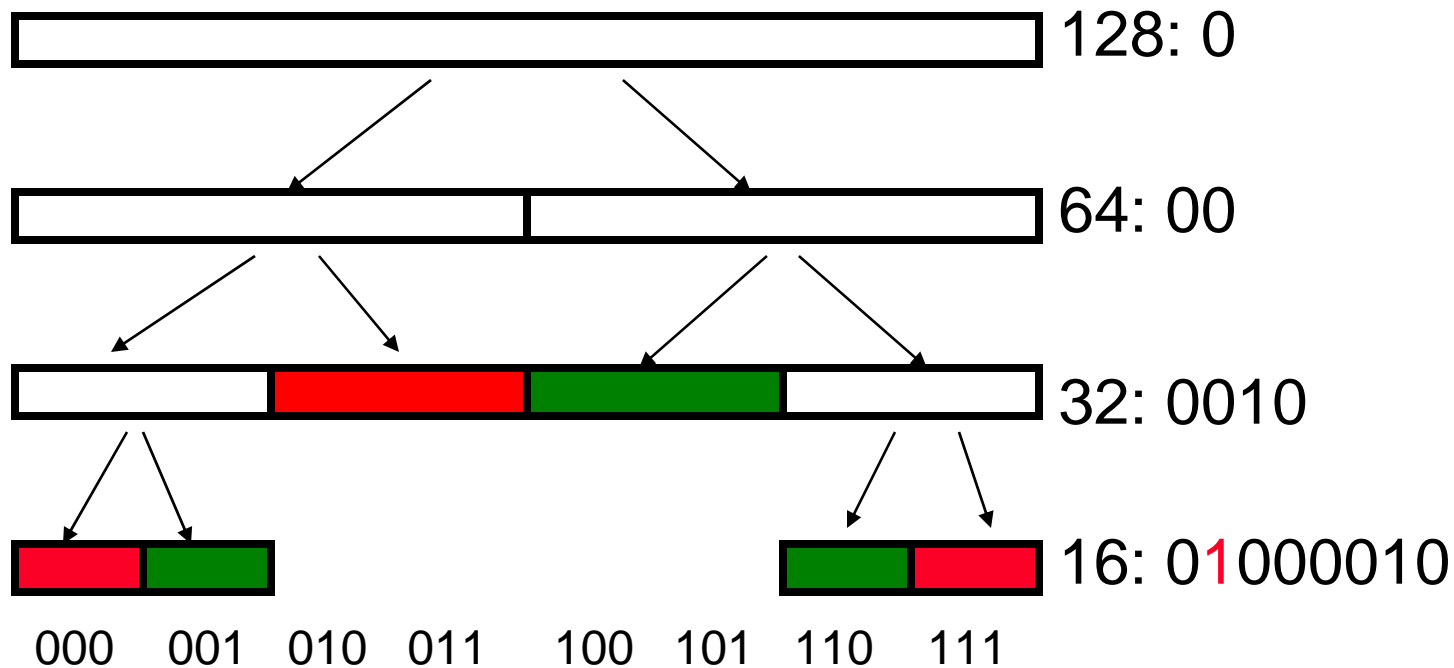


Initial State → Free(001) → Free(000) → Free(111) → Malloc(16)



Buddy System

- Legend: **FREE** **ALLOCATED** **SPLIT**



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

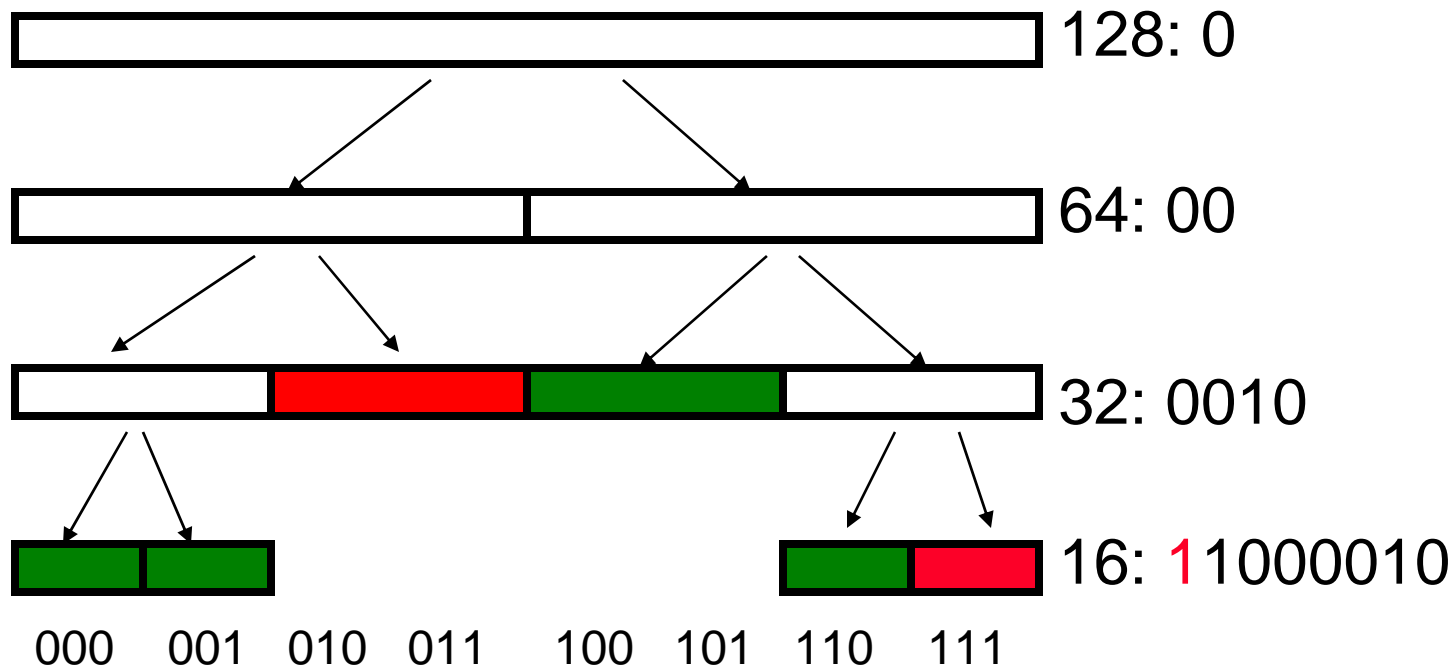


1



Buddy System

- Legend: **FREE** **ALLOCATED** **SPLIT**



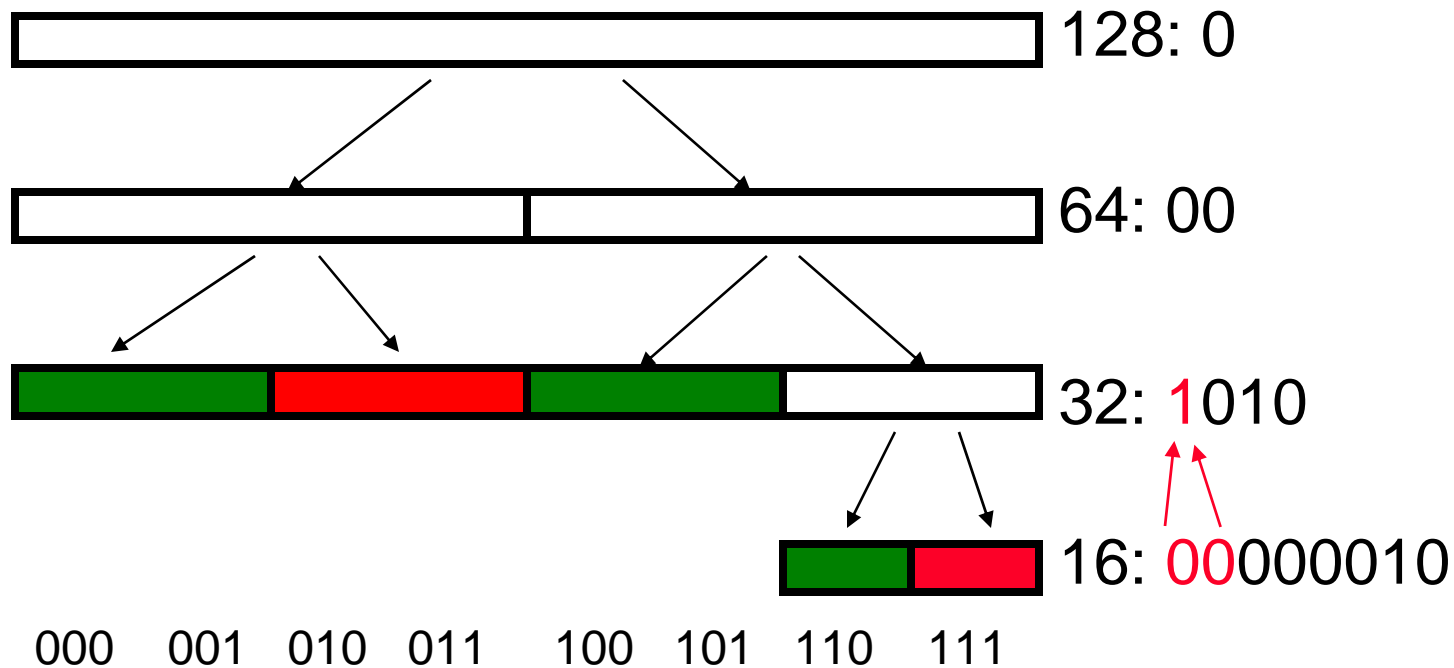
Initial State → Free(001) → Free(000) → Free(111) → Malloc(16)

↑
1



Buddy System

- Legend: **FREE** **ALLOCATED** **SPLIT**



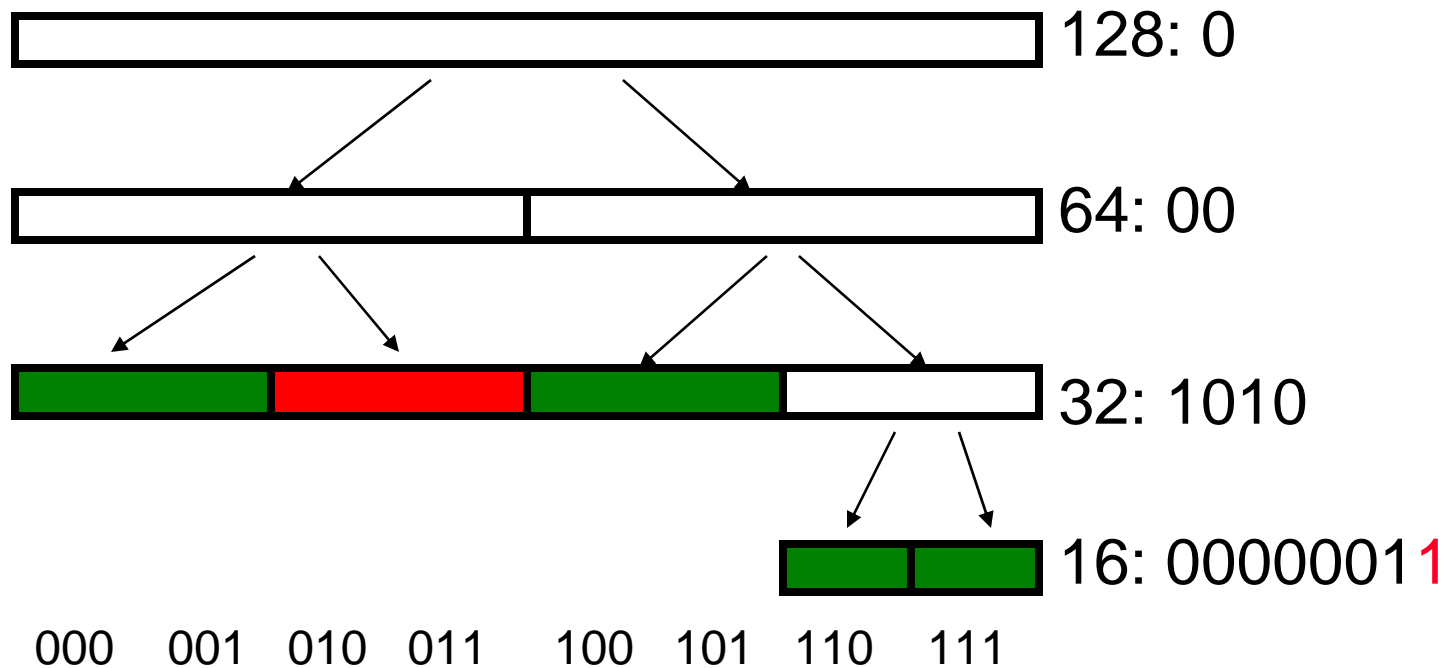
Initial State → Free(001) → Free(000) → Free(111) → Malloc(16)

↑
2



Buddy System

- Legend: **FREE** **ALLOCATED** **SPLIT**

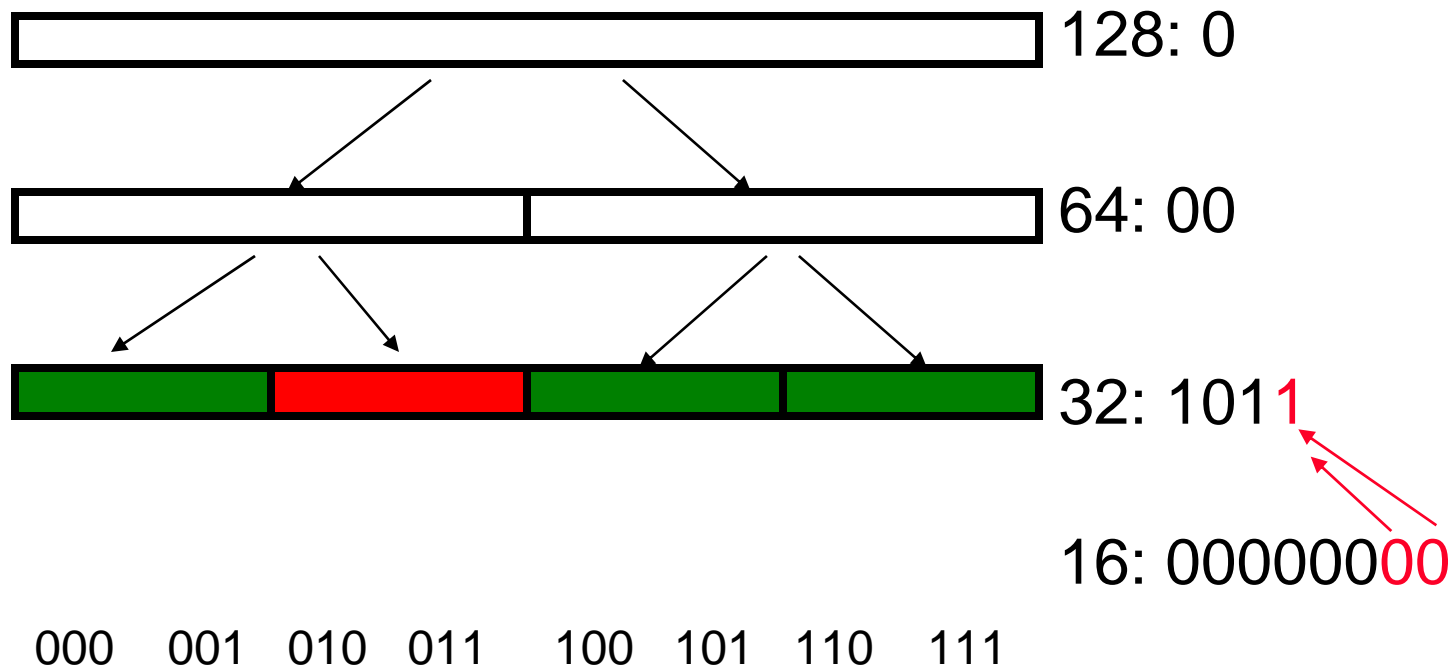


Initial State → Free(001) → Free(000) → Free(111) → Malloc(16)



Buddy System

- Legend: **FREE** **ALLOCATED** **SPLIT**



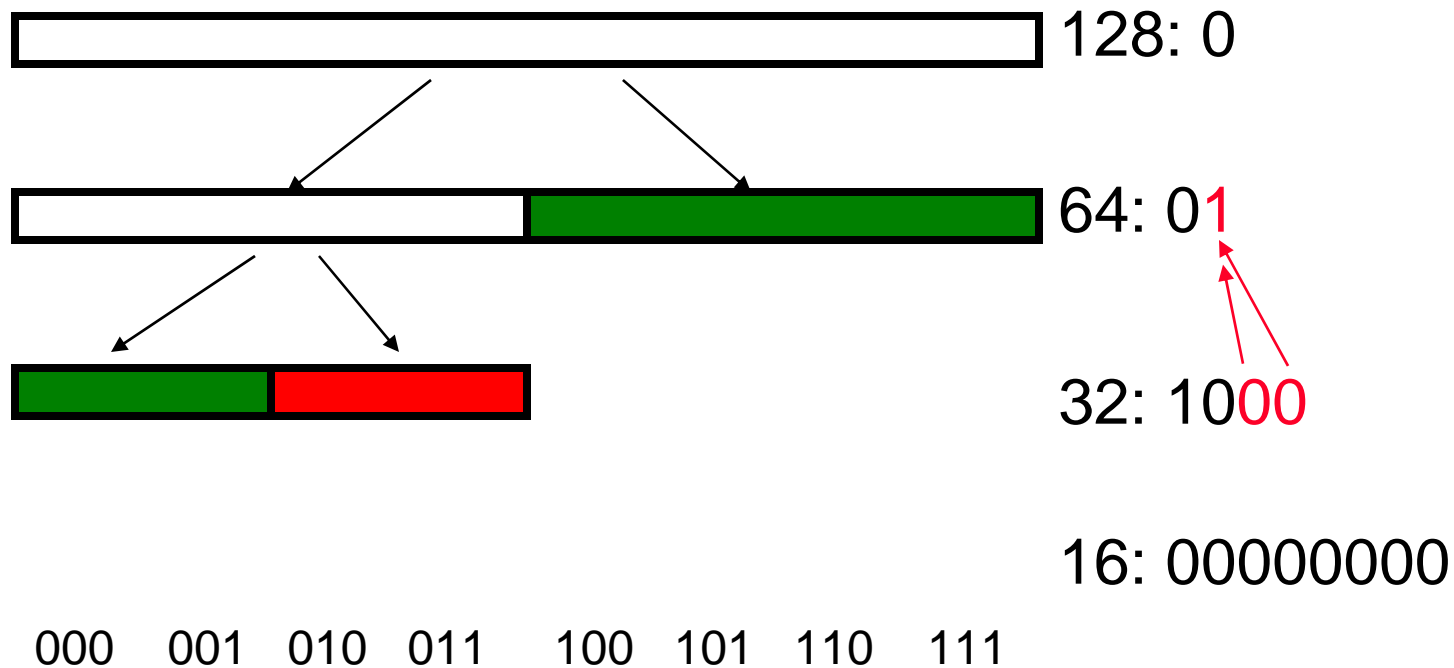
Initial State → Free(001) → Free(000) → Free(111) → Malloc(16)

2



Buddy System

- Legend: **FREE** **ALLOCATED** **SPLIT**



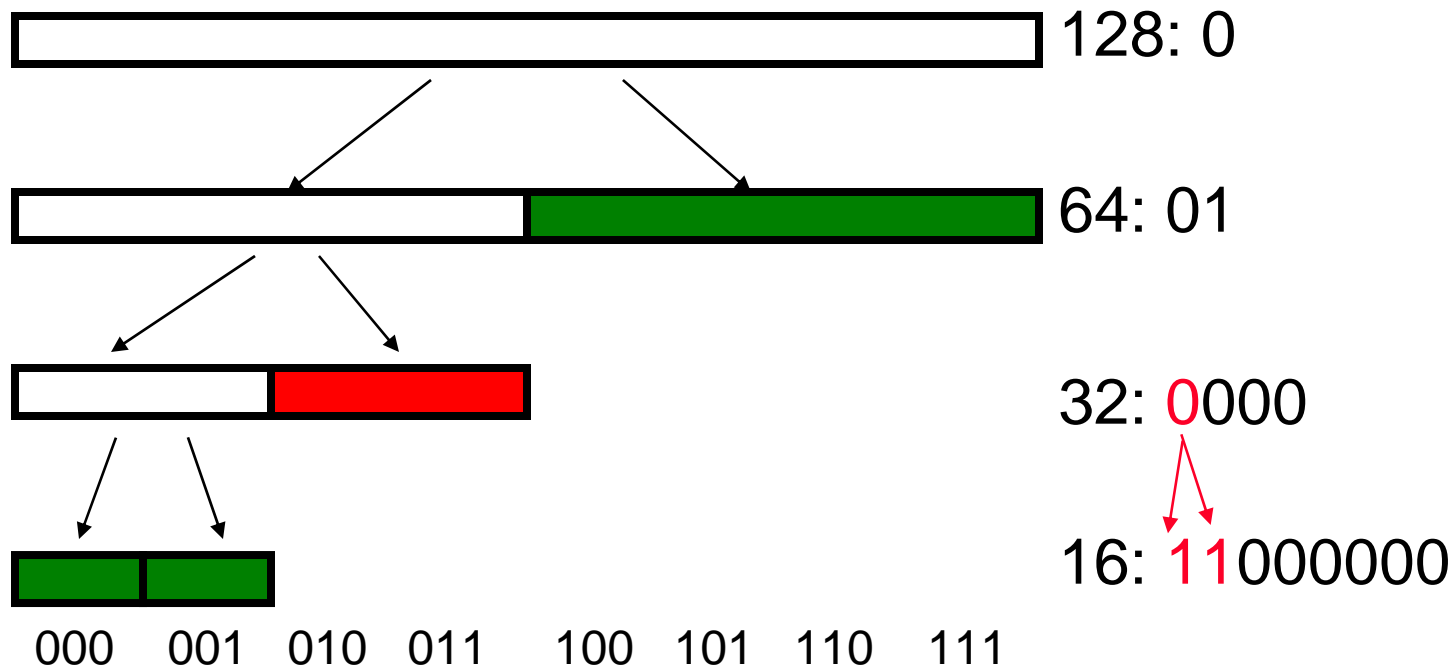
Initial State → Free(001) → Free(000) → Free(111) → Malloc(16)

↑
3



Buddy System

- Legend: **FREE** **ALLOCATED** **SPLIT**

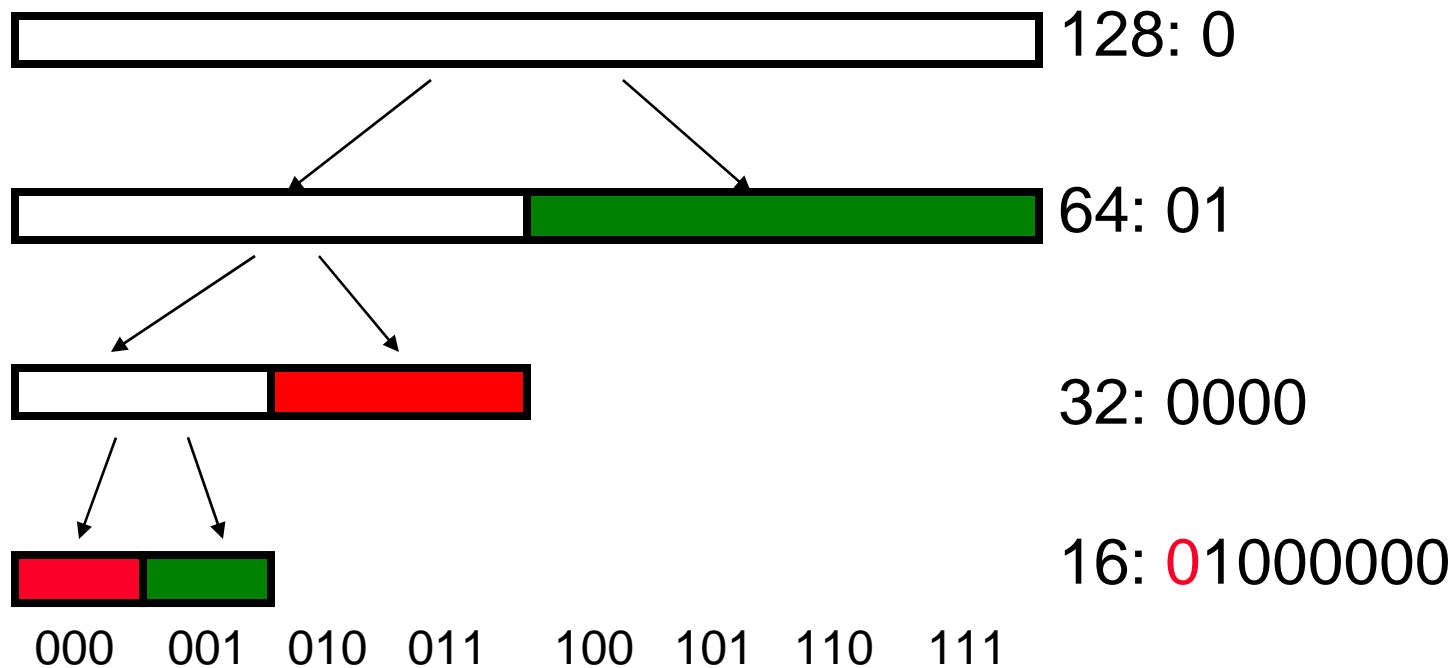


Initial State → Free(001) → Free(000) → Free(111) → Malloc(16)



Buddy System

- Legend: **FREE** **ALLOCATED** **SPLIT**



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

↑
2



Lecture Outline

- **Buddy System Allocator**
- **Garbage Collection**
- **MIPS**



Automatic Memory Management

- Dynamically allocated memory is difficult to track – why not track it **automatically**?
- If we can keep track of what memory is in use, we can reclaim everything else.
 - Unreachable memory is called **garbage**, the process of reclaiming it is called **garbage collection**.
- So how do we track what is in use?



Tracking Memory Usage

- Techniques depend heavily on the programming language and rely on help from the compiler.
- Start with all pointers in global variables and local variables (root set).
- Recursively examine dynamically allocated objects we see a pointer to.
 - We can do this in **constant space** by reversing the pointers on the way down
- How do we recursively find pointers in dynamically allocated memory?



Tracking Memory Usage

- Again, it depends heavily on the programming language and compiler.
- Could have only a single type of dynamically allocated object in memory
 - E.g., simple Lisp/Scheme system with only cons cells (61A's Scheme not “simple”)
- Could use a *strongly typed* language (e.g., Java)
 - Don't allow conversion (casting) between arbitrary types.
 - C/C++ are not strongly typed.



• Here are 3 schemes to collect garbage

Scheme 1: Reference Counting

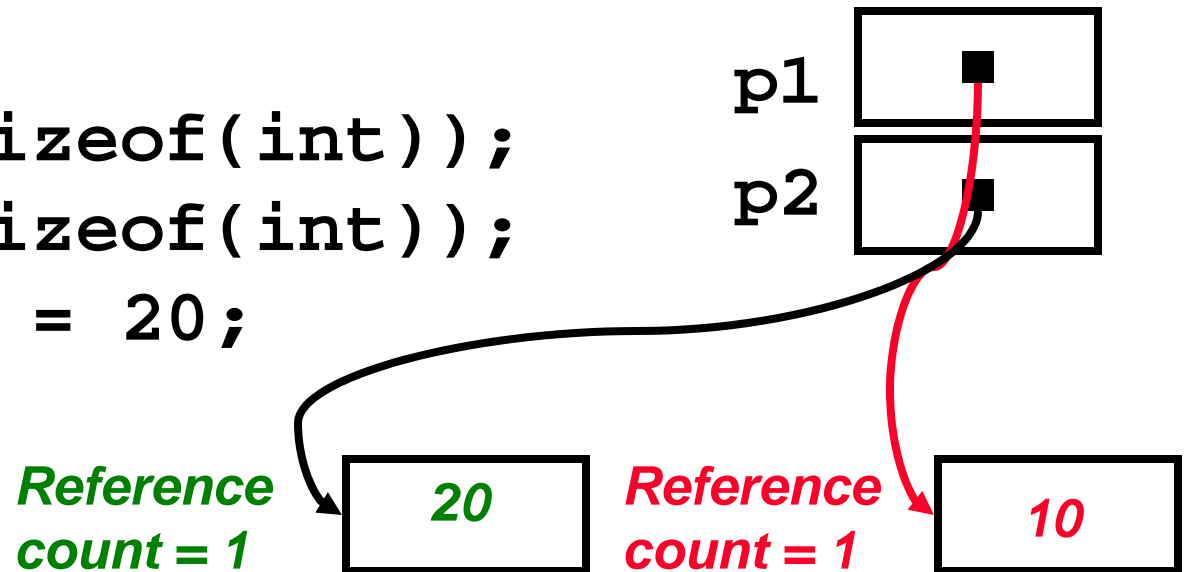
- For every chunk of dynamically allocated memory, keep a count of number of pointers that point to it.
- When the count reaches 0, reclaim.
- Simple assignment statements can result in a lot of work, since may update reference counts of many items



Reference Counting Example

- For every chunk of dynamically allocated memory, keep a count of number of pointers that point to it.
- When the count reaches 0, reclaim.

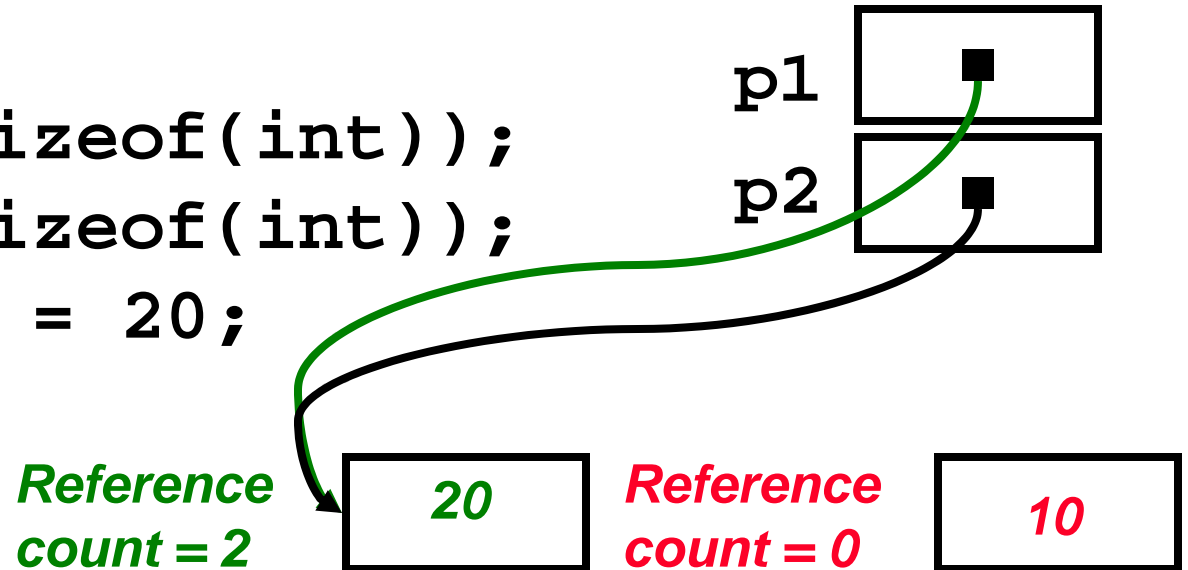
```
int *p1, *p2;  
p1 = malloc(sizeof(int));  
p2 = malloc(sizeof(int));  
*p1 = 10; *p2 = 20;
```



Reference Counting Example

- For every chunk of dynamically allocated memory, keep a count of number of pointers that point to it.
- When the count reaches 0, reclaim.

```
int *p1, *p2;  
p1 = malloc(sizeof(int));  
p2 = malloc(sizeof(int));  
*p1 = 10; *p2 = 20;  
p1 = p2;
```



Reference Counting (*p1*, *p2* are pointers)

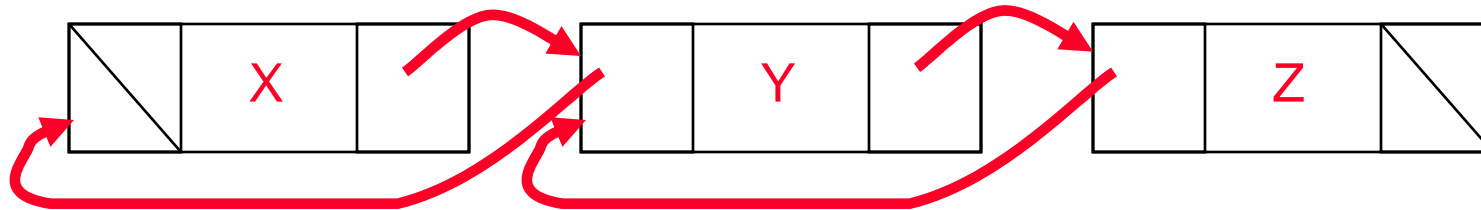
p1 = *p2*;

- Increment reference count for *p2*
- If *p1* held a valid value, decrement its reference count
- If the reference count for *p1* is now 0, reclaim the storage it points to.
 - If the storage pointed to by *p1* held other pointers, decrement all of their reference counts, and so on...
- Must also decrement reference count when local variables cease to exist.



Reference Counting Flaws

- **Extra overhead added to assignments, as well as ending a block of code.**
- **Does not work for circular structures!**
 - **E.g., doubly linked list:**



Scheme 2: Mark and Sweep Garbage Col.

- **Keep allocating new memory until memory is exhausted, then try to find unused memory.**
- **Consider objects in heap a graph, chunks of memory (objects) are graph nodes, pointers to memory are graph edges.**
 - **Edge from A to B \Rightarrow A stores pointer to B**
- **Can start with the root set, perform a graph traversal, find all usable memory!**
- **2 Phases: (1) Mark used nodes; (2) Sweep free ones, returning list of free nodes**



Mark and Sweep

- **Graph traversal is relatively easy to implement recursively**

```
void traverse(struct graph_node *node) {  
    /* visit this node */  
    foreach child in node->children {  
        traverse(child);  
    }  
}
```

- **But with recursion, state is stored on the execution stack.**

- **Garbage collection is invoked when not much memory left**

- **As before, we could traverse in constant space (by reversing pointers)**



Scheme 3: Copying Garbage Collection

- Divide memory into two spaces, only one in use at any time.
- When active space is exhausted, traverse the active space, copying all objects to the other space, then make the new space active and continue.
 - Only reachable objects are copied!
- Use “forwarding pointers” to keep consistency
 - Simple solution to avoiding having to have a table of old and new addresses, and to mark objects already copied (see bonus slides)



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



Lecture Outline

- **Buddy System Allocator**
- **Garbage Collection**
- **MIPS**



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, ...



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.**



ISA Design

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

Software

- Symbolic Lookup
- Strong typing
- Nested expressions
- Many operators

Hardware

- fixed var names/#
- No Typing
- Fixed format Inst
- small set of insts



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)



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 32?** **Smaller is faster**
- **Each MIPS register is 32 bits wide**
 - **Groups of 32 bits called a word in MIPS**



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



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



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



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



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



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**



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 do 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

<code>add \$t0,\$s1,\$s2</code>	<code># temp = g + h</code>
<code>add \$t1,\$s3,\$s4</code>	<code># temp = i + j</code>
<code>sub \$s0,\$t0,\$t1</code>	<code># f=(g+h)-(i+j)</code>



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!

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`



“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`

