

CSO-Recitation 12

CSCI-UA 0201-007

R12: Assessment 10 & Combinational logic

Today's Topics

- Assessment 10
- Lab-4
- Combinational logic
 - How to build a combinatorial logic circuit
 - MUX

Assessment 10


Q1 Basic malloc

Which of the following statements are true w.r.t. malloc?

- A. Every call to malloc results in the memory allocator making a syscall (e.g. sbrk) to request memory from OS.
- B. malloc returns failure if and only if the memory allocator does not have any free chunks. chunk size -> that's why we care about fragmentation
- C. When using the implicit-list design, malloc tends to traverse more chunks than when using the explicit-list design. only free chunks
- D. None of the above.

Q2 Malloc design

Which of the following are true w.r.t. C's dynamic memory allocator design?

- A. The design can move previously allocated space to a different location to reduce fragmentation. Restriction: – Once allocated, space cannot be moved around
- B. The design can assume that users strictly alternate calls to malloc and free. Assumption: Use APIs freely -> Can issue an arbitrary sequence of malloc/free
- C. The design can assume that the argument of free is the return value of some previous malloc calls. Assumption: Use APIs correctly
- D. The design can invoke arbitrary <stdlib.h> functions including standard library's malloc/free library calls.
- E. None of the above.

Q3 Implicit list

Suppose your implicit list design uses both header and footer. Both have the following type (Lecture slides 30):

- `get_status()`
- `get_size()`
- `set_size_status()`
- `set_status()`
- `set_size()`
- `payload2header()`
- `payload2footer()`
- `footer2header()`
- `curr2prev()`
- ...

} Basic helper
Find in lecture slides

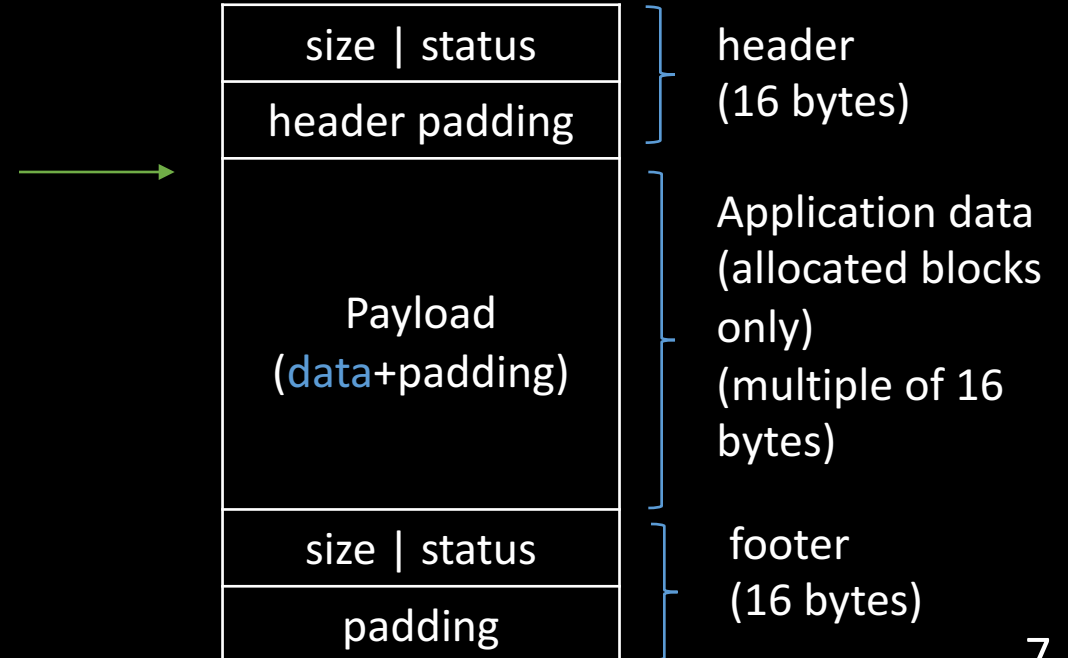
```
typedef struct {  
    unsigned long size_and_status;  
    unsigned long padding;  
} header;
```

Q3.1 payload2header example

- Suppose a user invokes `free(p)` using pointer `p` whose value is `0x789012345670`. What is the memory address for the start of the chunk that contains the allocated space (payload) that should be freed? (To facilitate autograding, please write your answer in hex with prefix `0x`, ignoring leading zeros and using lowercase letters)

• **0x789012345660**

e.g. `p=malloc(20);`
`free(p);`



Q3.2 payload2header

```
header* payload2header(void *p)
{
    header *h;
    _____???:
    return h;
}
```

`payload2header` takes as argument a **pointer to the start of the payload** in the chunk, and returns a **pointer to the chunk's header**.

Which of the following C statement to use for the missing line?

A. `h = (header *)p - sizeof(header);`

B. `h = (header *)p - 1;`

C. `h = (header *)((char *)p - sizeof(header));`

D. `h = (char *)p - 1;`

E. None of the above.

to get to the header, we want

- the addr of payload – the size of header (16 bytes)
- but p is a pointer -> pointer arithmetic
- need to casting to char *
 - `(char *)p - sizeof(header)`
- finally cast it to pointer to header
 - `(header *)((char *)p - sizeof(header))`

another way:

- casting p to header * first
- because now p is header *,
 - the pointer arithmetic -1 will -header_size
 - and the type is header *
- `(header *)p - 1;`

Q3.3 header2footer example

- Suppose pointer variable h **points to the beginning of a chunk** and has value 0x7890123456a0. If the total size of the chunk is 1KB (including header and footer fields), then what is the memory address for the footer of this chunk?
- **0x789012345a90**
 - total size of chunk = 1KB
 - 1024-byte -> 0x400
 - $0x7890123456a0 + 0x400 = 0x789012345aa0$
 - end of the chunk
 - $0x789012345aa0 - 0x10 = 0x789012345a90$
 - beginning of the footer

Q3.4 header2footer

```
header* header2footer(header *h)
{
    header *f;
    _____??_;
    return f;
}
```

`header2footer` takes as argument a pointer to the start of the chunk, and returns a pointer to the same chunk's footer.

Which of the following C statement to use for the missing line? Note that `get_size` is a helper function that returns the chunk size encoded in the header/footer field `size_n_status`.

- A. `f = h + 1;` go to next/prev 16 bytes
- B. `f = h - 1;`
- C. `f = h + get_size(h);`
- D. `f = (header *)((char *)h + get_size(h));`
- E. `f = h - get_size(h);`
- F. `f = (header *)((char *)h - get_size(h));`
- G. `f = (header *)((char *)h + get_size(h) - sizeof(header));`
- H. `f = (header *)((char *)h - get_size(h) + sizeof(header));`
- I. None of the above.

header2footer

- the addr of `h + size of total chunk in byte - header size in byte`
 - need first casting `h` to `char *`
 - if don't, actually go to `addr(h) + size(chunk) * sizeof(header) - sizeof(header) * sizeof(header)`
- finally casting it to `header *`

Q3.5 footer2header example

- Suppose pointer variable `f` points to the beginning of a chunk's footer and has value `0x789012345a90`. If the total size of the chunk is 1KB (including header and footer fields), then what is the memory address for the header of this chunk?
- `0x7890123456a0`
 - total size of chunk = 1KB
 - 1204-byte -> `0x400`
 - $0x789012345a90 + 0x10 = 0x789012345aa0$
 - end of the footer
 - $0x789012345aa0 - 0x400 = 0x7890123456a0$
 - beginning of the chunk (header)

Q3.6 footer2header

```
header* footer2header(header *f)
{
    header *h;
    _____;
    return h;
}
```

`footer2header` takes as argument a pointer to the footer of the chunk, and returns a pointer to the same chunk's header.

Which of the following C statement to use for the missing line? Note that `get_size` is a helper function that returns the chunk size encoded in the header/footer field size_n_status.

- A. `h = f + 1;`
- B. `h = f - 1;`
- C. `h = f + get_size(f);`
- D. `h = (header *)((char *)f + get_size(f));`
- E. `h = f - get_size(f);`
- F. `h = (header *)((char *)f - get_size(f));`
- G. `h = (header *)((char *)f + sizeof(header) - get_size(f));`
- H. `h = (header *)((char *)f - sizeof(header) + get_size(f));`
- I. None of the above.

`footer2header`

- the addr of f + size of footer in byte – size of total chunk in byte

Q3.7 curr2prev example

- Suppose pointer variable h points to the beginning of some chunk and has value 0x789012345aa0. Suppose this chunk has size 4KB and its previous chunk has size 1KB. What is the memory address for the beginning of its previous chunk?
- 0x7890123456a0
 - high level: $\text{addr} - \text{size of the previous chunk}$
 - $0x789012345aa0 - 1\text{KB}$
 - $0x789012345aa0 - 0x400 = 0x7890123456a0$

Q3.8 curr2prev example

```
header* curr2prev(header *curr)
{
    header *prev_footer;
    _____;
    return footer2header(prev_footer);
}
```

`curr2prev` takes as argument a **pointer to the current chunk's header**, and returns a **pointer to the previous chunk's header**.

Which of the following C statement to use for the missing line? Note that **footer2header** is the helper function that returns a pointer to the chunk's header given a pointer to the same chunk's footer.

- A. `prev_footer = curr - 1 ;`
- B. `prev_footer = curr - sizeof(header);`
- C. `prev_footer = (header *)((char *)curr - sizeof(header));`
- D. `prev_footer = curr - 2;`
- E. `prev_footer = curr - 2*sizeof(header);`
- F. `prev_footer = (header *)((char *)curr - 2*sizeof(header));`
- G. None of the above.

curr2prev

- `curr_header2prev_footer`
- `prev_footer2prev_header -> footer2header`

Lab-4

Lab-4

- If you don't do bonus
 - only need to write code in file `mm-implicit.c`
 - a very simple design -> malloc using implicit list without footer
 - Complete helper function `next_chunk` and use it to complete the heap checker function `mm_heapcheck`
 - Complete helper functions `ask_os_for_chunk`, `split`, `first_fit` and use them to implement `mm_malloc`
 - Complete helper functions `payload2header`, `coalesce` and use them to implement `mm_free`
 - Complete `mm_realloc`
- Tips
 - Please follow the instruction & comment code! – the lab shouldn't be hard
 - Review helper functions, pseudocode, steps summary
- Due after Thanksgiving, encourage to do bonus part

realloc

- In the C, the **realloc** function is used to **resize** a block of memory that was **previously allocated**
- The realloc function allocates a block of memory (which be can make it larger or smaller in size than the original) and copies the contents of the old block to the new block of memory, if necessary
- **void *realloc(void *ptr, size_t size);**
 - if ptr is NULL, the call is equivalent to malloc(size)
 - if size is equal to zero, the call is equivalent to free(ptr)
 - if ptr is not NULL, it must have been returned by an earlier call to malloc or realloc

realloc

- Copies the contents of the old block to the new block of memory, if necessary
 - If the new block is larger, everything else is uninitialized
 - How to copy data?
 - See a C library function, `memcpy()`
- `void *memcpy(void *dest, void * src, size_t n)`
 - copies `n` characters from memory area `src` to memory area `dest`.

Combinational logic

Building Blocks

How we get there..

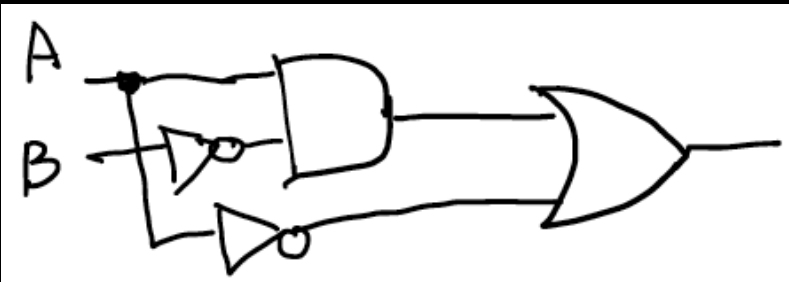
- Instruction set (e.g. X86) is the interface between software and hardware
- Logic design
 - assumes the existence of a collection of standard logic components
 - made from logic gates -> made from small groups of transistors
 - studies the organization of primitive logic components into subsystems that perform calculation and data transfer
 - e.g. An arithmetic-logic unit (**ALU**) is the part of a CPU that carries out arithmetic and logic operations on the operands in computer instruction words

Combinational Logic

- There is no memory
 - That is, the outputs are a function ONLY of the current inputs, not of anything in the past
- Values are either true or false (but not both, or anything else)
 - We commonly represent true with 1 and false with 0
- There is at least one input and at least one output
 - But there can be more
 - Each input is either true or false
 - Each output should be defined for all possible values for the inputs!

Combinational Logic

- There are three main ways to represent combinational circuits
 1. As a **circuit diagram**
 2. As a set of equations/**expressions**
 3. As a **truth table**



$$A\bar{B} + \bar{A}$$

A	B	C
0	0	1
0	1	1
1	0	1
1	1	0

Combinational Logic

- Logical expressions are build from a number of “gates”
- You are already familiar with the most important ones!
 - AND, OR, NOT
 - All boolean functions can be written with these three building blocks!
 - There are others, like XOR and NAND
 - All boolean functions can be written with just NAND!!!
 - e.g. $\text{NOT}(A) == \text{NAND}(A,A)$; $\text{AND}(A,B) == (A \text{ NAND } B) \text{ NAND } (A \text{ NAND } B)$
- Letters/words are used to represent the inputs

Combinational Logic

- Basic logic design
 - Logic circuits == Boolean expressions
- How to build a combinational logic circuit
 - Step1: Specify the truth table
 - Step2: Output is the sum of products

Filling in a truth table

- First, list out all of the possible inputs
 - For N inputs, there are 2^N possibilities
 - It helps to split things in two, that is, have space for the 2^N possibilities, and for the first variable put 0 for the first half and 1 for the second half
 - For the next variable, put 0 for the first half of the previous variable's 0 group and 1 for the second half, then do the same for its 1 group
- Then, evaluate the expression for each possible inputs
 - This is tedious, so there are some shortcuts, especially if the expression is written nicely

Filling in a truth table

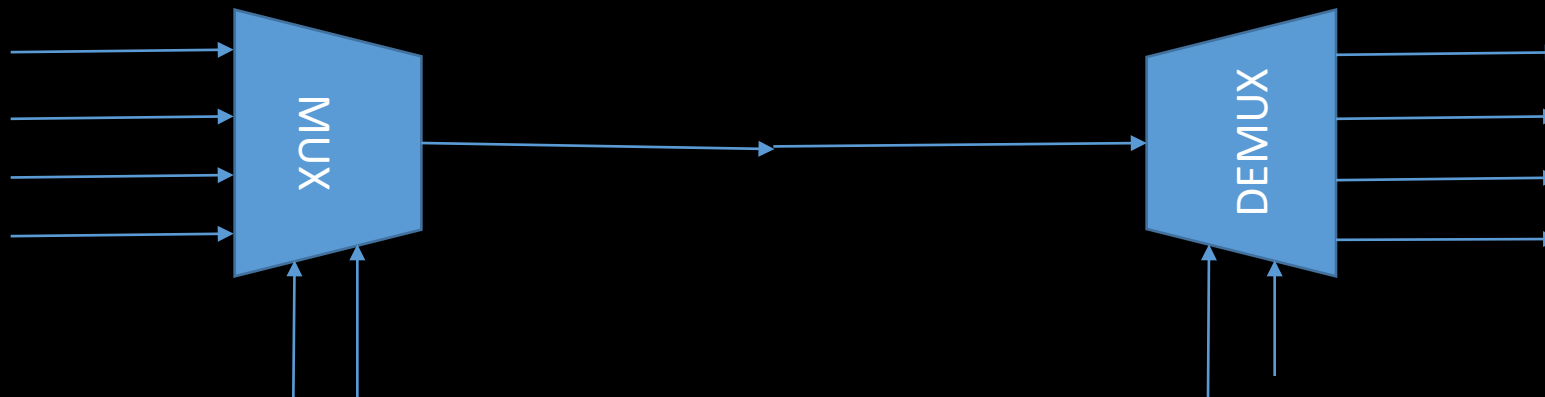
- Groups of things ANDed together are typically called a **clause**
 - That is, clauses are things separated by ORs
- For each clause, see what the variables present are
 - For any variable that is negated, keep in mind that that variable is 0
 - For any other variable in the clause, keep in mind it is 1
 - Then, look through the truth table and wherever you see a row that has all of the variables in the clause with the right value, put a 1 for the output
- Then when you are out of clauses, fill in 0 for any output left

Getting a circuit from a truth table

- Find the sum of product
- For each output, look for where that output is 1 in the truth table
 - Look at the list of inputs
 - anywhere an input a is 1, write a in the clause
 - Anywhere an input a is 0, write $\sim a$ in the clause
 - Say the output equals all of the clauses ORed together
 - Typically you can simplify, but that's a topic for a different day

Multiplexor (MUX)

- A multiplexor is a device which takes in multiple signals and outputs a single signal
- The purpose of using a multiplexor is to make full use of the capacity of the communication channel and greatly reduce the cost of the system
 - On the receiving side, a demultiplexer splits the single data stream into the original multiple signals



Multiplexor (MUX)

- In digital circuit design, the selector wires are of digital value
- 4-to-1 Multiplexor
- It can be noted that 2^N non-select signals require N select signals
 - 8-to-1 MUX, 16-to-1 MUX...

2^N input
signals

