

## Compilation

- The compiler generates an executable (e.g., a.out) that contains CPU instructions
- The OS loads the instructions into RAM at runtime

## Compilation

```
void foo() {  
    bar();  
}
```

```
void bar(){  
}
```

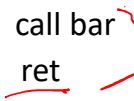
How does the compiler know the address of **bar** at compile time?

The actual RAM address of **bar** is known only at runtime

## Compilation

- The compiler does not need to know the actual address of **bar**
- **call** instruction's operand is the offset of **bar** relative to the address of the next instruction (i.e., subsequent instruction to call instruction)

```
foo:
e8 1 0 0 0    call bar
c3            ret
bar:
c3            ret
```



The operand of a call instruction is a relative address corresponding to the instruction pointer (address of subsequent instruction to call). Because the compiler statically knows the relative offset of all instructions, as long as the loader loads them at the same offsets at runtime, the compiler doesn't need to know the absolute addresses.

## Compilation

- As long as the relative offset of **foo** and **bar** is the same during compile and load time, the compiler doesn't need to know to actual addresses of **call** and **jmp** targets

## Compiling, linking and loading

- Preprocessor takes C source code, expands #include, etc., produce C source code
- Compiler takes C source code, produce assembly language
- Assembler takes assembly language, produces .o file

**main.c**

```
int foo(int, int);  
int main() {  
    return foo(10, 20);  
}  
gcc -c main.c  
generates main.o
```

**foo.c**

```
int foo(int x, int y) {  
    return x + y;  
}  
gcc -c foo.c  
generates foo.o
```

# Linker

- Linker takes multiple '.o' file, produces a single program image

a.out

```
foo:
mov 4(%esp), %eax
add 8(%esp), %eax
ret
```

```
main:
push $20
push $10
call foo
add $8, %esp
ret
```

# Linker

- Linker takes multiple '.o' file, produces a single program image
  - adjust relative addresses of call targets who are not local to a single object file

a.out

```
foo:
mov 4(%esp), %eax
add 8(%esp), %eax
ret
```

```
main:
push $20
push $10
call foo
add $8, %esp
ret
```



## Loader

- Loader loads the program image into the memory at runtime and jumps to main routine

## Dynamic memory allocation

- Dynamic: happens during execution
- Static: happens during compilation
- Why do we need dynamic memory allocation?

## Dynamic memory allocation

```
main() {  
    int arr[10000];  
    int n, i;  
    printf("enter the number of elements\n");  
    scanf("%d", &n);  
    for (i = 0; i < n; i++)  
        arr[i] = rand();  
}
```

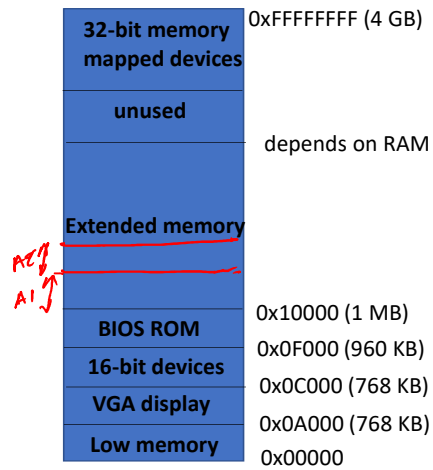
## Dynamic memory allocation

- Dynamic memory allocation is needed because
  - Sometimes, we don't know the total memory requirement during the compile time
- If we only want to use static memory allocation, then we may have to overapproximate the memory use
  - e.g., in the previous example, we have assumed that the number of elements cannot be more than 10000
    - If this assumption fails during runtime, then the program may not behave correctly
    - During runtime, we are always allocating 40000 bytes even though the actual number of elements is small (say 10)

## Dynamic memory allocation

- Local variables are allocated on the stack that requires a contiguous memory area
  - If the static memory allocation is too high, then the memory allocation is more likely to fail because contiguous memory may not always be available

## Physical address space



If multiple applications can use RAM at the same time, an application may find contiguous memory, even though RAM space is available.

## Dynamic memory allocation

- C standard library provides APIs for dynamic memory allocation
  - `void *malloc(size_t size);`
  - `void free(void *ptr);`
- Unlike local variables, where the allocation/deallocation happens automatically, dynamic memory allocation is entirely manual and done by the application

## malloc implementation

- Using `alloc_from_ram` API, we can only allocate memory of size multiples of 4096 bytes
  - Not suitable for small allocation sizes
- The RAM area used for dynamic memory allocation and deallocation is called the `heap`



## Buddy allocator

- Buddy allocator maintains free lists of memory objects of different allocation sizes ( $2^k$ )
- On allocation, the allocation size is rounded to the nearest  $2^k$  that is greater than or equal to the current allocation size
- If an object exists in the free list corresponding to  $2^k$  size, then the allocator removes the object from the list and returns to the caller
  - Otherwise, the allocator allocates an object of size  $2^{k+1}$ , splits the object into two halves, adds one to the free list and returns the another to the caller

## Buddy allocator

8
16
32
64
128
256
512
1024
2048
4096

Initially, all lists are empty.

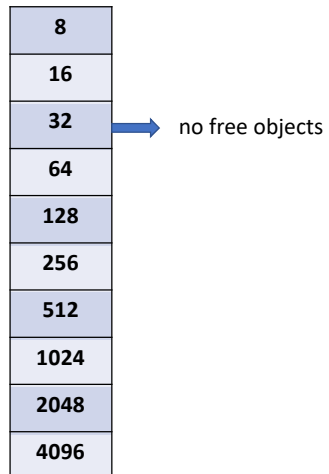
The buddy allocator maintains different lists of free memory objects. These lists are called buckets. Each bucket contains free (available for allocation) objects of size  $2^k$  (called bucket size). On allocation, the allocation size is rounded to the nearest bucket size that is greater than or equal to the current allocation size. If the corresponding bucket in the buddy allocator contains free objects, then the allocator removes an object from the bucket and return to the caller. Otherwise, the buddy allocator recursively calls itself to allocate an object that is twice the size of the bucket size, split them into two halves (objects), add one to the bucket and return another object to its caller. Notice that, in the worst case, an allocation may trigger multiple allocations up to the highest bucket size (4096). If the bucket corresponding to bucket size 4096 is empty, then the allocator calls `alloc_from_ram` to allocate 4096 bytes and return to its caller. During free, the object is added to the corresponding bucket and checks if the adjacent object is also free. If the adjacent object is free, then the allocator removes both of them from the bucket, merges them, and recursively calls itself to free the merged object. Similar to allocation, the merging and deallocation to buckets of larger sizes can go all the way up to the bucket of size 4096. Whenever an object is promoted to the bucket of size 4096, the allocator immediately calls `free_ram` to return the RAM space to the OS.

## Buddy allocator

8
16
32
64
128
256
512
1024
2048
4096

malloc(24)  
rounded size : 32

## Buddy allocator



malloc(24)  
rounded size : 32

## Buddy allocator

8
16
32
64
128
256
512
1024
2048
4096



malloc(24)  
rounded size : 32

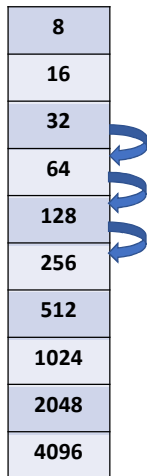
## Buddy allocator

8
16
32
64
128
256
512
1024
2048
4096



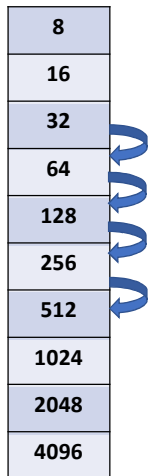
malloc(24)  
rounded size : 32

## Buddy allocator



malloc(24)  
rounded size : 32

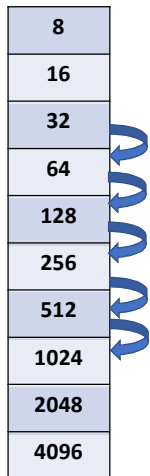
## Buddy allocator



malloc(24)  
rounded size : 32

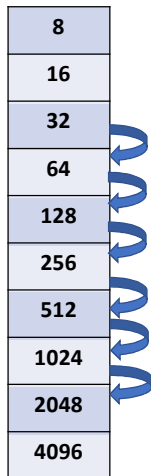


## Buddy allocator



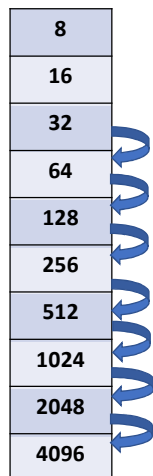
malloc(24)  
rounded size : 32

## Buddy allocator



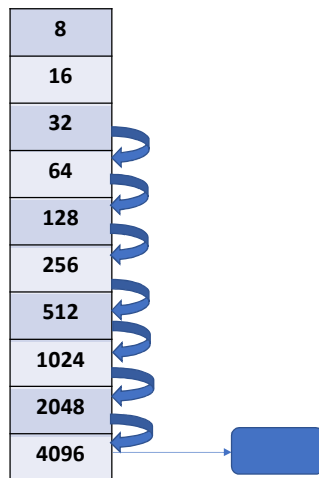
malloc(24)  
rounded size : 32

## Buddy allocator



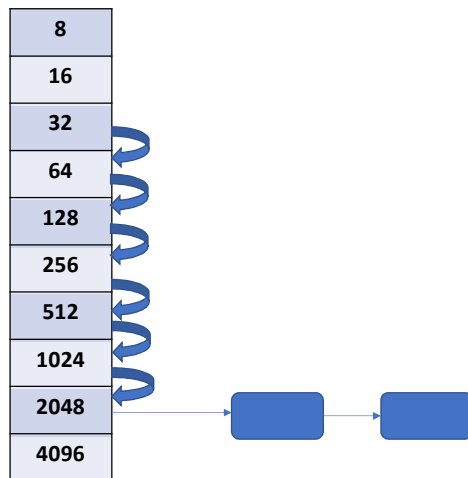
malloc(24)  
rounded size : 32

## Buddy allocator



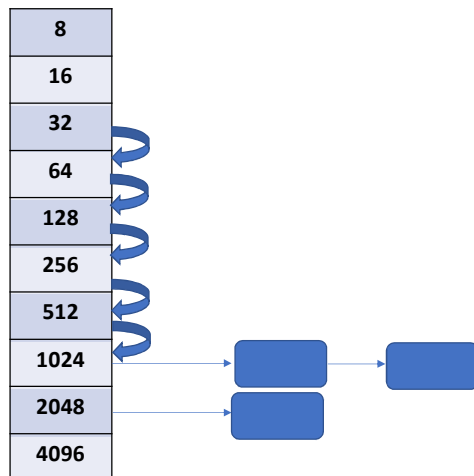
malloc(24)  
rounded size : 32

## Buddy allocator



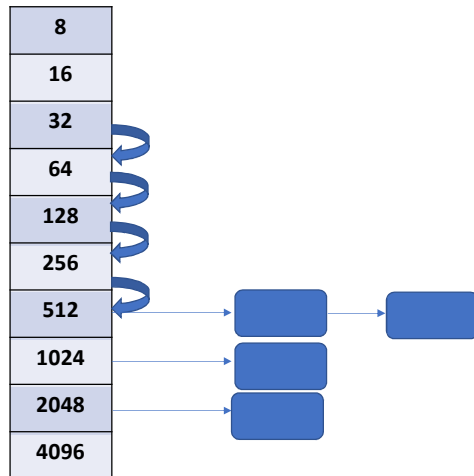
malloc(24)  
rounded size : 32

## Buddy allocator



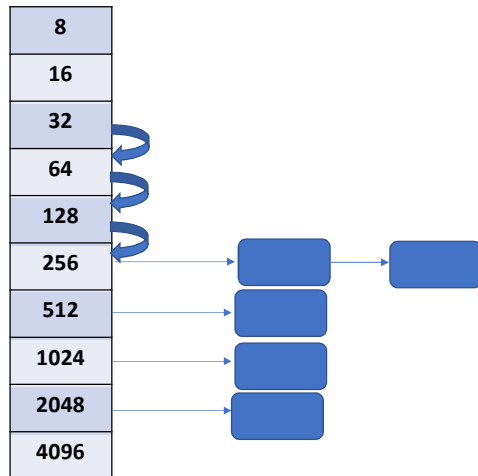
malloc(24)  
rounded size : 32

## Buddy allocator



malloc(24)  
rounded size : 32

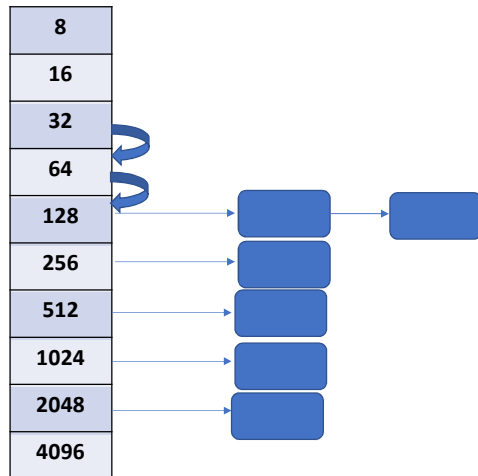
## Buddy allocator



malloc(24)  
rounded size : 32

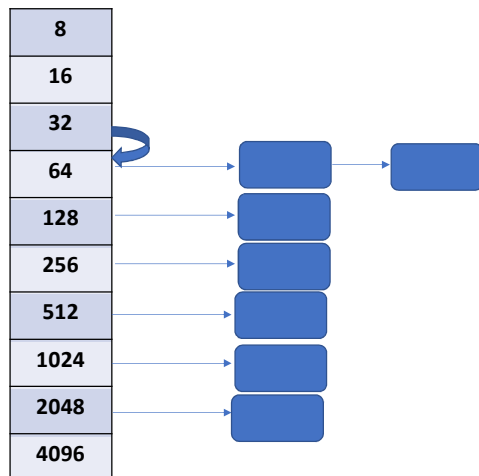


## Buddy allocator



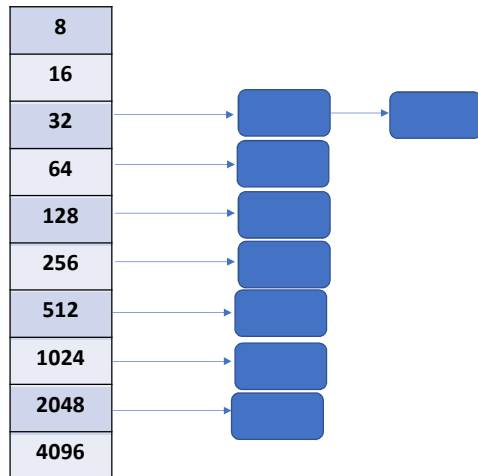
malloc(24)  
rounded size : 32

## Buddy allocator



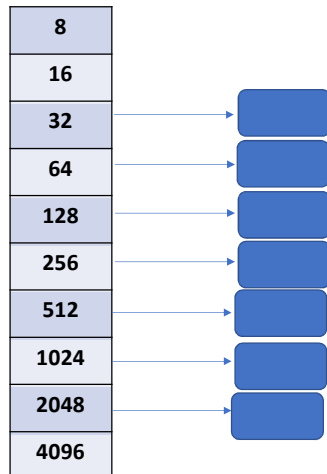
malloc(24)  
rounded size : 32

## Buddy allocator



malloc(24)  
rounded size : 32

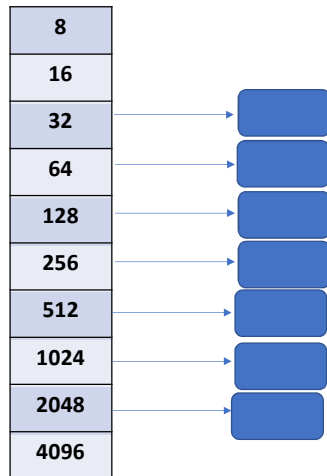
## Buddy allocator



A = 32

malloc(24)  
rounded size : 32

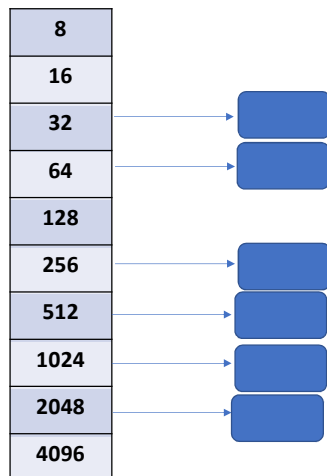
# Buddy allocator



A = 32

malloc(24)  
rounded size : 32  
malloc(128)

## Buddy allocator

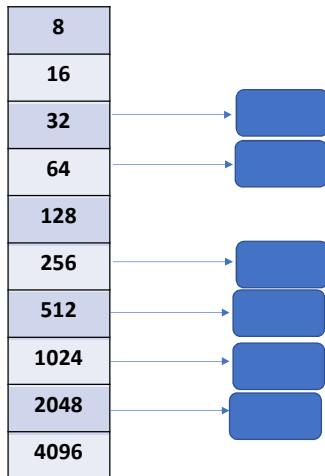


A = 32

B=128

malloc(24)  
rounded size : 32  
malloc(128)

## Buddy allocator

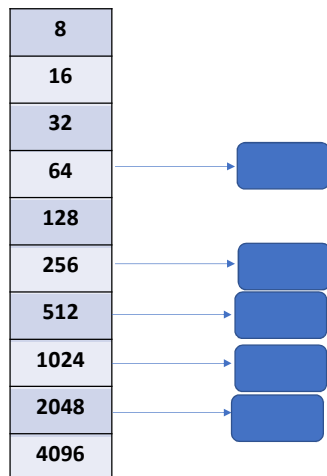


A=32

B=128

malloc(24)  
rounded size : 32  
malloc(128)  
malloc(32)

## Buddy allocator



A=32

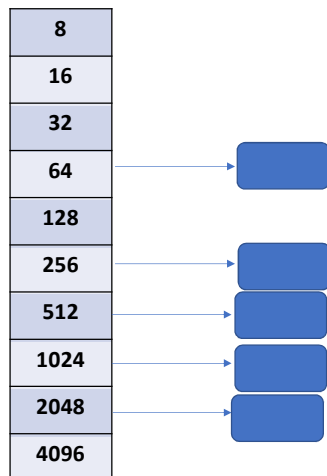
B=128

C=32

malloc(24)  
rounded size : 32  
malloc(128)  
malloc(32)



## Buddy allocator



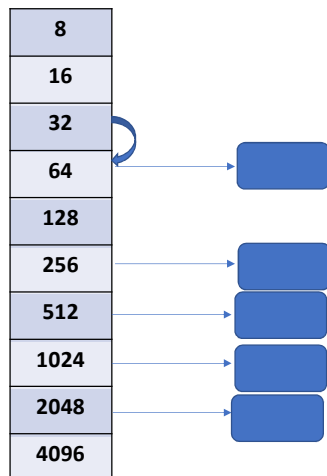
A=32

B=128

C=32

malloc(24)  
rounded size : 32  
malloc(128)  
malloc(32)  
malloc(32)

## Buddy allocator



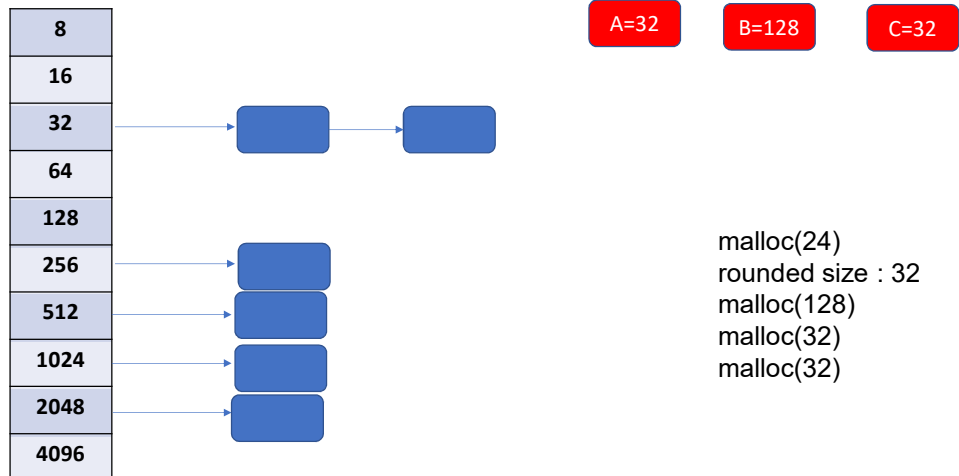
A=32

B=128

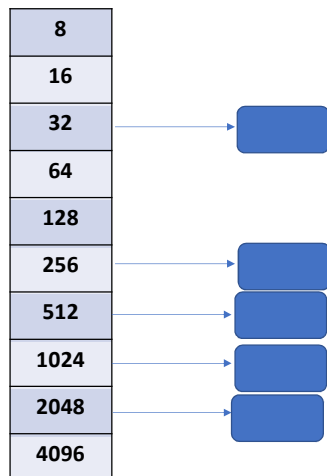
C=32

malloc(24)  
rounded size : 32  
malloc(128)  
malloc(32)  
malloc(32)

## Buddy allocator



## Buddy allocator



A=32

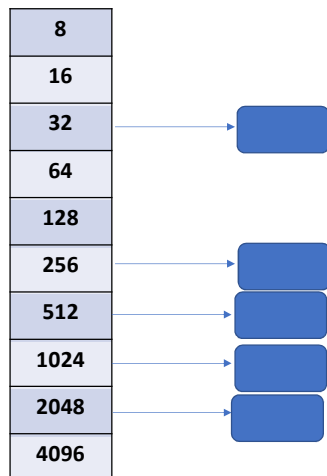
B=128

C=32

D=32

malloc(24)  
rounded size : 32  
malloc(128)  
malloc(32)  
malloc(32)

## Buddy allocator



A=32

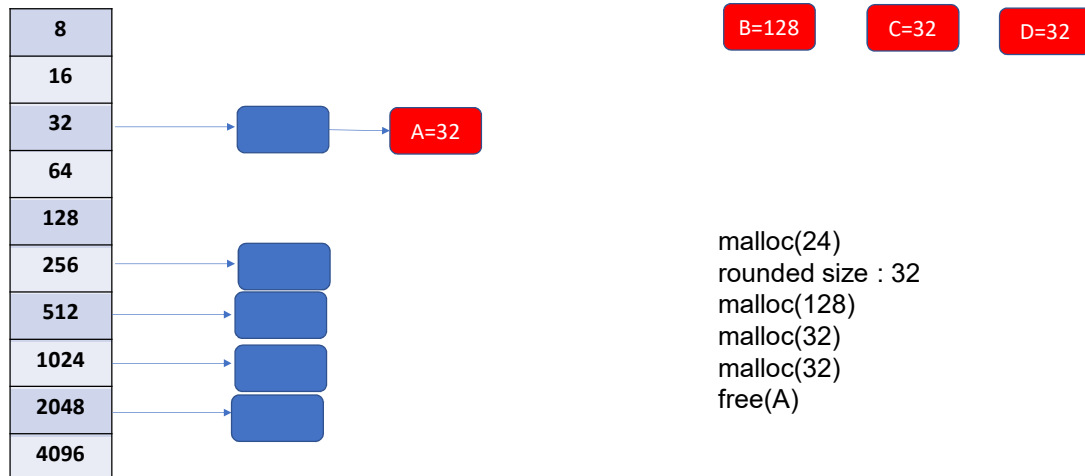
B=128

C=32

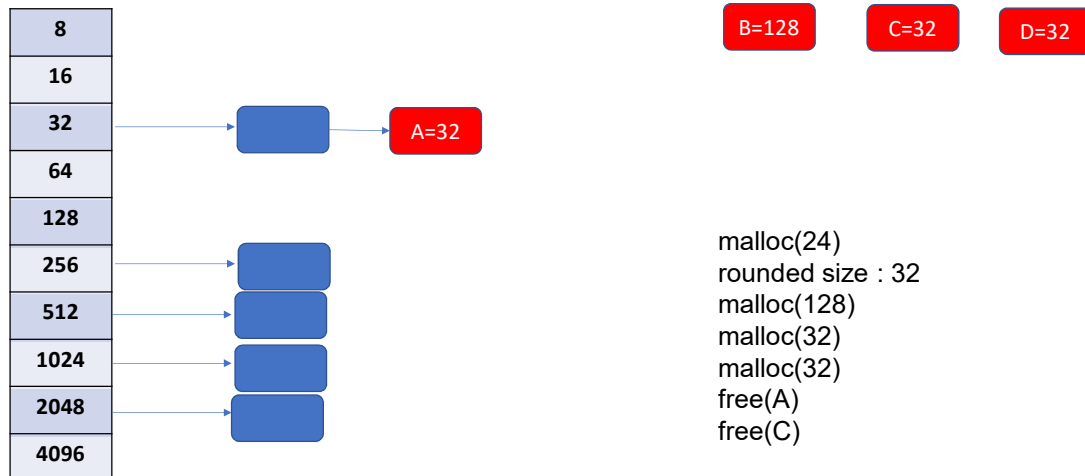
D=32

malloc(24)  
rounded size : 32  
malloc(128)  
malloc(32)  
malloc(32)  
free(A)

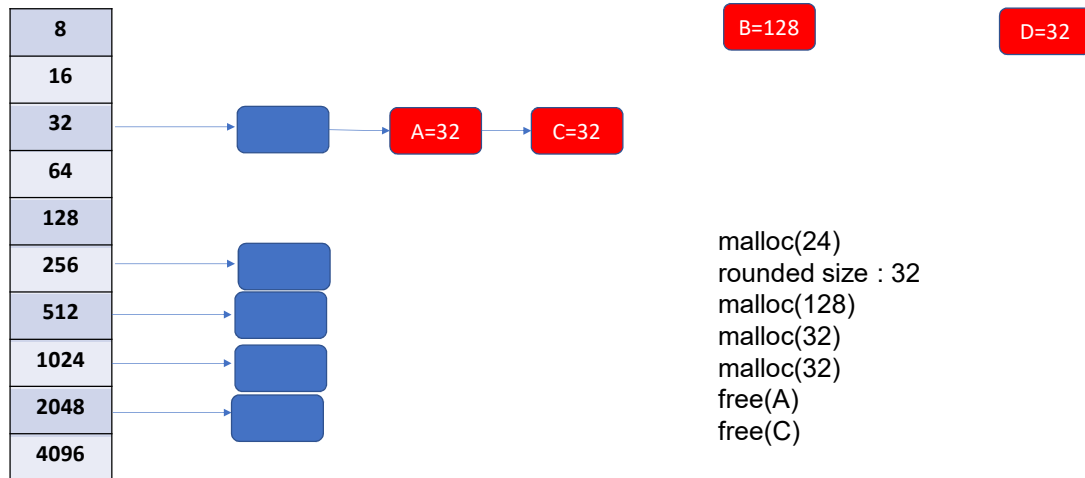
## Buddy allocator



## Buddy allocator

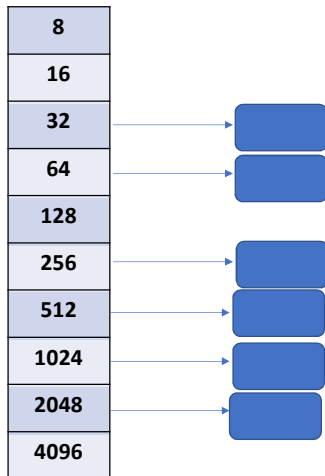


## Buddy allocator





## Buddy allocator

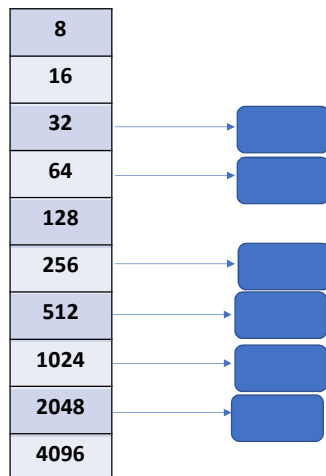


B=128

D=32

malloc(24)  
rounded size : 32  
malloc(128)  
malloc(32)  
malloc(32)  
free(A)  
free(C)

## Buddy allocator

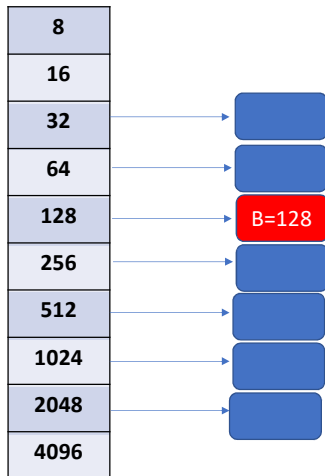


B=128

D=32

```
malloc(24)
rounded size : 32
malloc(128)
malloc(32)
malloc(32)
free(A)
free(C)
free(B)
```

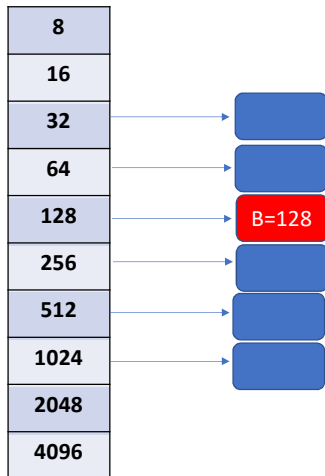
## Buddy allocator



D=32

```
malloc(24)
rounded size : 32
malloc(128)
malloc(32)
malloc(32)
free(A)
free(C)
free(B)
malloc(2048)
```

## Buddy allocator

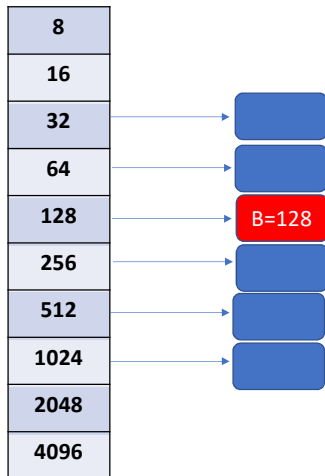


E=2048

D=32

```
malloc(24)
rounded size : 32
malloc(128)
malloc(32)
malloc(32)
free(A)
free(C)
free(B)
malloc(2048)
```

## Buddy allocator

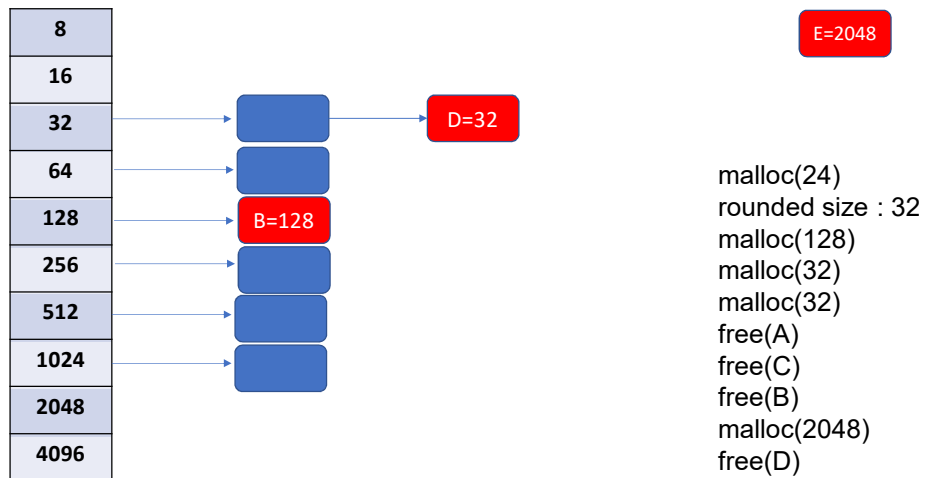


E=2048

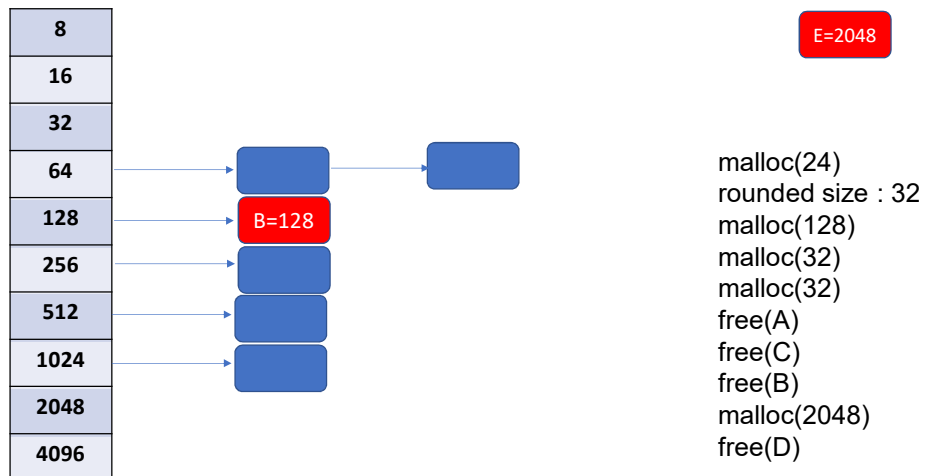
D=32

```
malloc(24)
rounded size : 32
malloc(128)
malloc(32)
malloc(32)
free(A)
free(C)
free(B)
malloc(2048)
free(D)
```

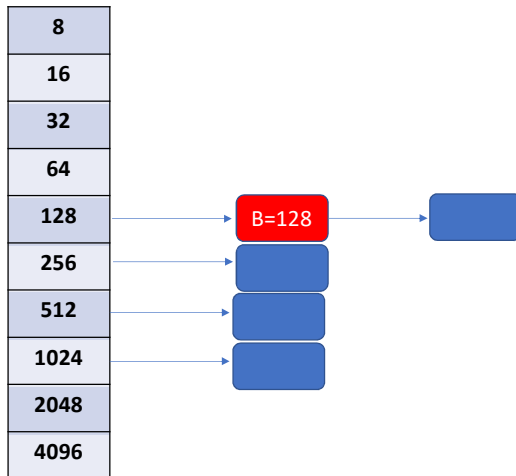
## Buddy allocator



## Buddy allocator



# Buddy allocator



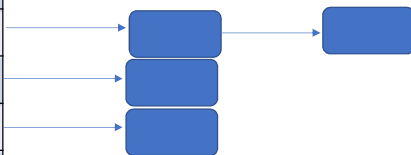
E=2048

```
malloc(24)
rounded size : 32
malloc(128)
malloc(32)
malloc(32)
free(A)
free(C)
free(B)
malloc(2048)
free(D)
```



# Buddy allocator

8
16
32
64
128
256
512
1024
2048
4096



E=2048

```
malloc(24)
rounded size : 32
malloc(128)
malloc(32)
malloc(32)
free(A)
free(C)
free(B)
malloc(2048)
free(D)
```

## Buddy allocator

8
16
32
64
128
256
512
1024
2048
4096



E=2048

```
malloc(24)
rounded size : 32
malloc(128)
malloc(32)
malloc(32)
free(A)
free(C)
free(B)
malloc(2048)
free(D)
```

## Buddy allocator

8
16
32
64
128
256
512
1024
2048
4096



E=2048

```
malloc(24)
rounded size : 32
malloc(128)
malloc(32)
malloc(32)
free(A)
free(C)
free(B)
malloc(2048)
free(D)
```

## Buddy allocator

8
16
32
64
128
256
512
1024
2048
4096




E=2048

```
malloc(24)
rounded size : 32
malloc(128)
malloc(32)
malloc(32)
free(A)
free(C)
free(B)
malloc(2048)
free(D)
```

## Buddy allocator

8
16
32
64
128
256
512
1024
2048
4096



E=2048

```
malloc(24)
rounded size : 32
malloc(128)
malloc(32)
malloc(32)
free(A)
free(C)
free(B)
malloc(2048)
free(D)
malloc(2048)
```

## Buddy allocator

8
16
32
64
128
256
512
1024
2048
4096

E=2048

F=2048

```
malloc(24)
rounded size : 32
malloc(128)
malloc(32)
malloc(32)
free(A)
free(C)
free(B)
malloc(2048)
free(D)
malloc(2048)
```

## Buddy allocator

8
16
32
64
128
256
512
1024
2048
4096

E=2048

F=2048

malloc(24)  
rounded size : 32  
malloc(128)  
malloc(32)  
malloc(32)  
free(A)  
free(C)  
free(B)  
malloc(2048)  
free(D)  
malloc(2048)  
free(F)

## Buddy allocator

8
16
32
64
128
256
512
1024
2048
4096

F=2048

E=2048

```
malloc(24)
rounded size : 32
malloc(128)
malloc(32)
malloc(32)
free(A)
free(C)
free(B)
malloc(2048)
free(D)
malloc(2048)
free(F)
```



## Buddy allocator

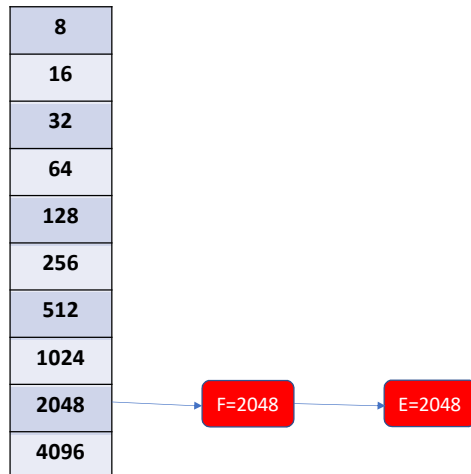
8
16
32
64
128
256
512
1024
2048
4096

F=2048

E=2048

```
malloc(24)
rounded size : 32
malloc(128)
malloc(32)
malloc(32)
free(A)
free(C)
free(B)
malloc(2048)
free(D)
malloc(2048)
free(F)
free(E)
```

## Buddy allocator



```
malloc(24)
rounded size : 32
malloc(128)
malloc(32)
malloc(32)
free(A)
free(C)
free(B)
malloc(2048)
free(D)
malloc(2048)
free(F)
free(E)
```

# Buddy allocator

8
16
32
64
128
256
512
1024
2048
4096

free\_ram(A, 4096)

A

```
malloc(24)
rounded size : 32
malloc(128)
malloc(32)
malloc(32)
free(A)
free(C)
free(B)
malloc(2048)
free(D)
malloc(2048)
free(F)
free(E)
```

## Buddy allocator

8
16
32
64
128
256
512
1024
2048
4096

malloc(24)  
rounded size : 32  
malloc(128)  
malloc(32)  
malloc(32)  
free(A)  
free(C)  
free(B)  
malloc(2048)  
free(D)  
malloc(2048)  
free(F)  
free(E)

## Why free\_ram is required?

- `void free_ram(void *ptr, size_t size)` is the deallocator corresponding to `void* alloc_from_ram(size_t size)`
- If the allocator doesn't free memory allocated using `alloc_from_ram` when it is not needed, the OS will not be able to use it for other tasks

## >4096 allocations

- For allocations of a size larger than 4096, the size is rounded up to the nearest multiples of 4096 and `alloc_from_ram` is used directly
  - When the application frees the memory `free_ram` API is immediately called to free the memory

## Why promote objects during free?

- If we don't merge and promote objects during free, then a large allocation may cause an unnecessary call to `alloc_from_ram`
  - e.g., in the absence of promotion, the last allocation of size 2048 in the previous example would have caused another call to `alloc_from_ram`
- Call to `alloc_from_ram` is very expensive
  - will be discussed later in this course

## Simple allocator (Assignment-1)

16
32
64
128
256
512
1024
2048
MAX

Simple allocator also maintains free lists corresponding to different allocation sizes ( $2^k$ ), but it doesn't move objects between lists of different sizes



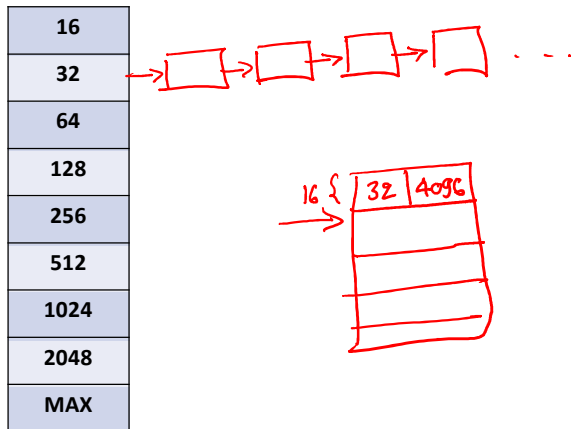
# malloc

16
32
64
128
256
512
1024
2048
MAX

malloc(24)  
rounded size : 32

# malloc

A=32



malloc(24)  
rounded size : 32  
page = alloc\_from\_ram(4096)  
divide page to chunks of 32 bytes  
add them to the free\_list(32)  
pop a memory object from  
free\_list(32) and return to caller

free

*free(A)*

16
32
64
128
256
512
1024
2048
MAX



free(ptr)  
add ptr to free list corresponding  
to the size of ptr  
*How to find out the size of ptr?*

## Simple allocator

15    1111  
      1000

- `alloc_from_ram` can allocate a memory object of size multiples of 4096
- The address returned by the `alloc_from_ram` is 4096 bytes aligned
  - i.e., the address is divisible by 4096
  - the lower 12-bits of the address are zero
- Let us call the memory area returned by the `alloc_from_ram` a page

## Simple allocator

- Simple allocator stores some metadata at the start of every page, and the rest of the page is divided into chunks of allocation size ( $2^k$ ) and added to free list

```
struct page_metadata {  
    int allocation_size;  
    long long free_bytes_available;  
};
```

## free

16
32
64
128
256
512
1024
2048
MAX

`free(ptr)`

Add ptr to free list corresponding to the size of ptr

**How to find out the size of ptr?**

fetch the allocation size from page metadata

update the `free_bytes_available`

if `free_bytes_available == 4096`

remove all objects on this page from the free list corresponding to the size of ptr

`free_ram(page, 4096)`

## malloc

16
32
64
128
256
512
1024
2048
MAX

```
malloc(128)
page = alloc_from_ram(4096)
initialize page metadata
divide [page+size_meta, page+4096]
to chunks of 128 bytes
add these chunks to free_list(128)
pop a memory object from
free_list(128) and return to caller
```

## Simple allocator

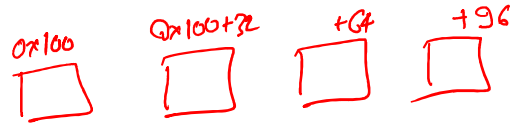
- You will be provided `alloc_from_ram` and `free_ram` APIs
- You are not supposed to use the standard library's `malloc` and `free` routines
- You are not supposed to use a third party linked list library
  - You have to implement your own linked list implementation for free list
- You are to implement `mymalloc` and `myfree` APIs of the simple allocator



## Simple allocator

- How are we supposed to implement a free list without using malloc?

```
struct node {  
    struct node *prev;  
    struct node *next;  
    char *addr;  
};
```



## Simple allocator

- How are we supposed to implement a free list without using malloc?
  - Instead of allocating new nodes for the free list, which store the addresses of free objects, we can use the free object itself as a free list node
  - as long as the size of free object is greater than or equal to a linked list node, we are fine
    - the above condition is true because the minimum allocation size in the simple allocator is 16 that is enough to store a linked list node

## Global variables

- The global variables remain alive throughout the program execution
- The compiler allocates space for global variable statically (at compile time)
  - global variables never get deallocated
- The actual RAM addresses of global variables are only known at runtime
  - how does the compiler know the addresses of global variables statically

Similar to instructions, global variables also reside in the a.out file. The compiler knows the relative offset of a global variable with respect to an instruction pointer. However, generating code for global variables is not that easy, because, in 32-bit mode, we can not directly read/write data to an address relative to EIP.

## Static addresses of global variables

- Compiler knows the address of global variables relative to current EIP

```
int a = 0;  
└─┬─┘  
main:  
→ mov $100, a  
ret
```

## RIP-Relative addressing

- Section 2.2.1.6 in Intel manual -2
- In 64-bit mode a new addressing mode, RIP-relative is introduced
- Using RIP-relative addressing, we can directly access RIP-relative addresses
  - e.g., `movl $100, 4(%rip)` → `*(unsigned*)(%rip+4) = 100`
  - where `%rip` contains the address of the next instruction (i.e., subsequent instruction of `movl $100, 4(%rip)`)

## 32-bit mode

*movl (%eax), %eax  
ret*

- In 32-bit mode, RIP-relative addressing mode is not present
- However, we can compute EIP-relative address using the following trick

*movl \$100, 0x10(%eip)      /\* not allowed in 32-bit \*/*

-----  
*call get\_eip              // return %eip corresponding to movl in %eax  
movl \$100, off(%eax)      // off = 0x10 + length(movl \$100, off(%eax))*

To compute an EIP relative address, we can call the `get_eip` routine that returns the address of the next instruction in the `%eax` register. After returning from `get_eip`, `%eax` can be used to access memory relative to `%eip`. The definition of `get_eip` is shown on the next slide.

## get\_eip

```
get_eip:  
    mov (%esp), %eax  
    ret
```

# Application

- CPU instructions
  - e.g., x86 instructions
  - loader (OS) loads the instructions anywhere in the RAM and set the %eip to the starting instruction (e.g., the first instruction of main)
  - branch instructions operands are offset relative to current %eip that is known at compile time



## Application

- Local variables are automatically allocated and deallocated on the stack
- The application allocates space for RAM at the start of the main routine and sets the stack pointer (%esp)

# Application

- Dynamic memory allocation
  - An application can use malloc/free routines for dynamic memory allocation

# Application

- Global variables
  - Global variables are allocated once and never get deallocated
  - Compiler allocates space for global variable similar to instructions
    - all the addresses (global variables, call and jmp targets, etc.) are relative to the current instruction pointer (%eip)