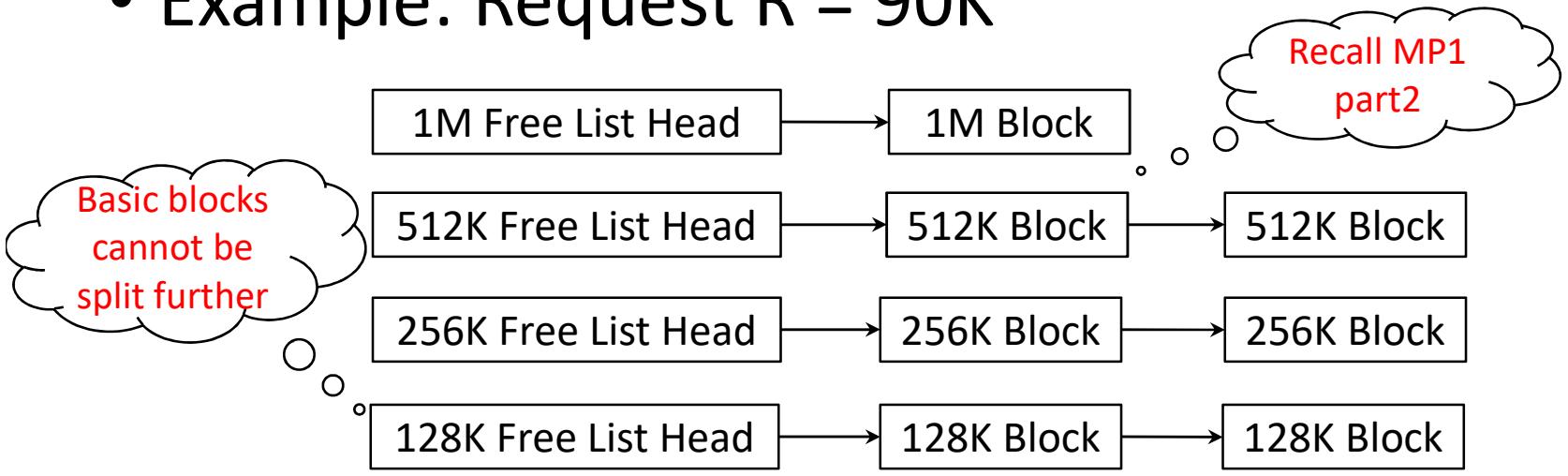
CSCE 313

MPX Introduction

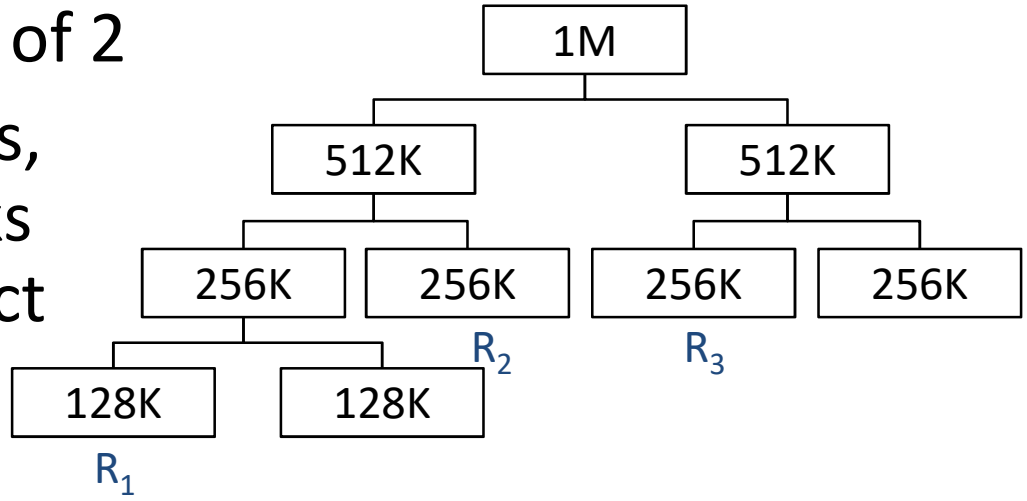
MPX Introduction

- Buddy system is specified by basic block size b and memory size M
  - E.g., b = 128K, M = 1M
  - Both should be power of 2
- Free blocks are organized into tiered free lists
- Example: Request R = 90K



Binary Buddy System

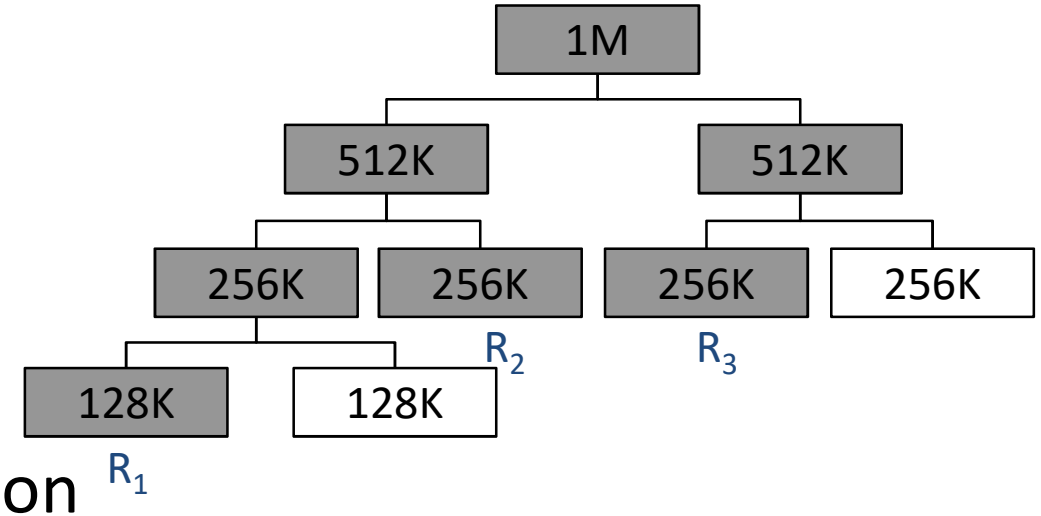
- Simple yet efficient memory management
- Organizes memory in blocks of size 2^k
- When request of size R arrives
  - Find a free block with size that's nearest power of 2
  - If no such block exists, split larger free blocks until a block of correct size is available



- Example
  - 1M memory, request R_1 = 90K
  - R_2 = 150K, R_3 = 200K

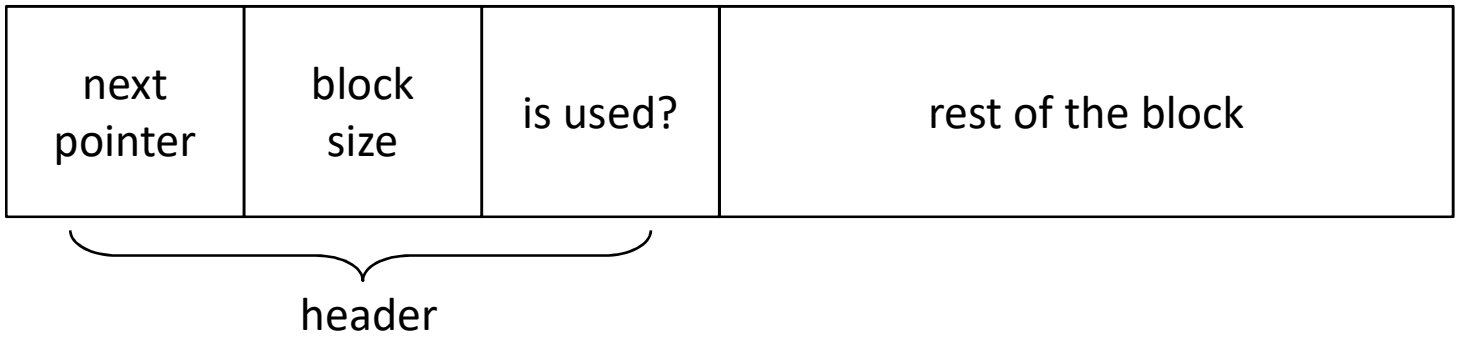
Binary Buddy System

- To free a block, check if its buddy is free
  - If so, merge into a larger free block
  - Merge process repeats until we can't go further
- Example
  - Free order R_2, R_1, R_3
- Drawbacks?
  - Internal and external fragmentation
  - Constant splitting and merging



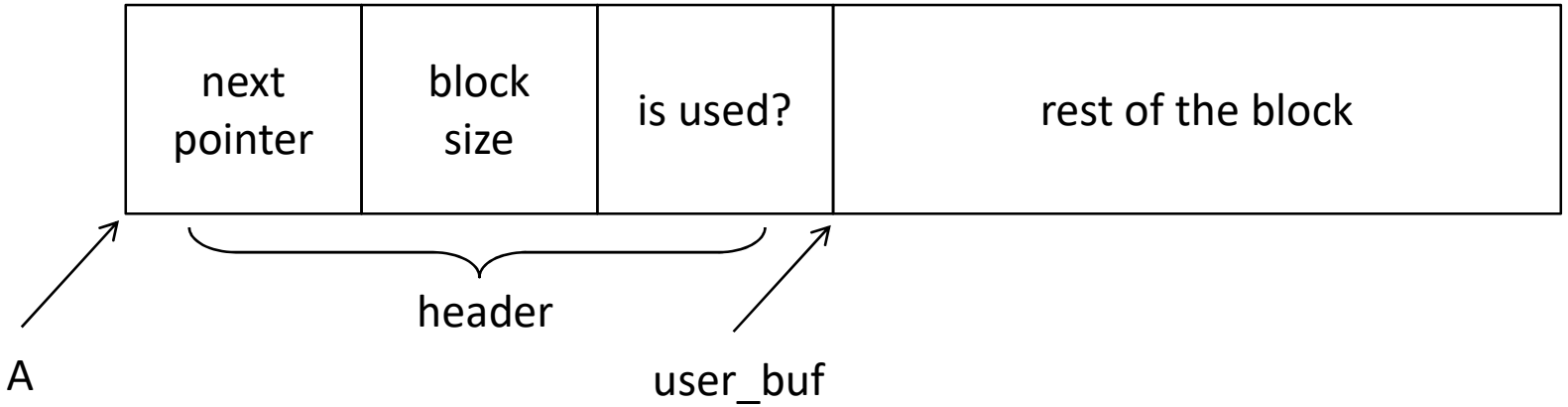
Header

- To organize free blocks into linked lists, we must maintain header information



- Available space for user: B - sizeof(header)
- Example: R = 120, sizeof(header) = 16
  - Size needed: R + sizeof(header) = 136
  - Round to power of 2: B = 256
  - Look for size 256 free blocks

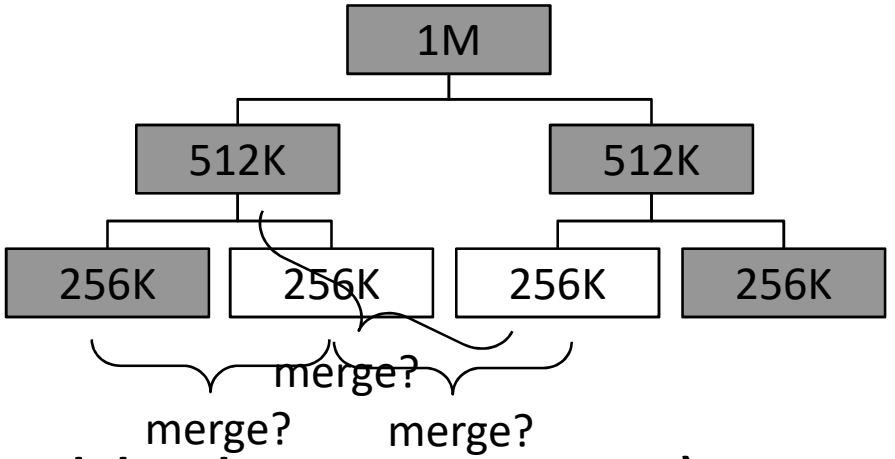
Header



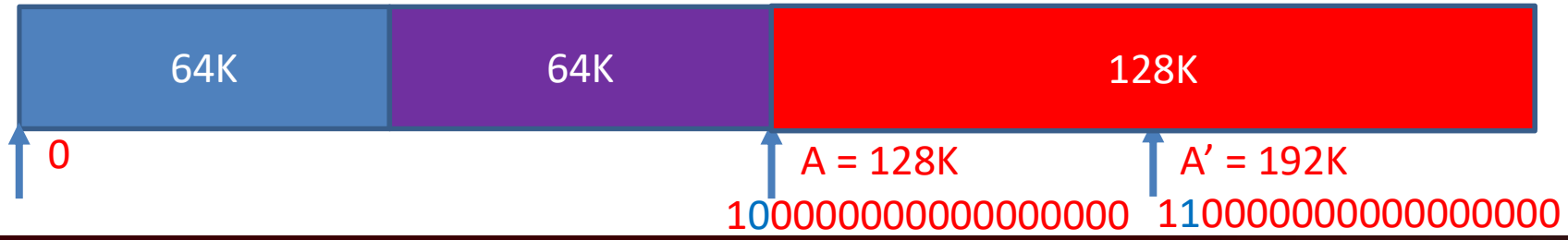
- void *my_malloc(int R)
  - Find a free block A that satisfies the requested R, may need to split larger blocks
  - Setup header info, return pointer user_buf
- bool my_free(void *user_buf)
  - Locate header A = user_buf - sizeof(header)
  - Free this block, may need to merge buddies

Merge Buddies

- Merge two blocks if
  - Both are free
  - They are buddies of the same block size



- Compute buddy addr (for block size B = 64K)
  - Approach #1: A + B = A', A' – B = A, need extra info in block header to distinguish left and right
  - Approach #2: A XOR B = A', A' XOR B = A, no need to modify header (only works for offset starts from 0)



Wrap up

- void *my_malloc(int R)
  - Needed size B = NearestPow2(R + sizeof(header))
  - Lookup free list of size B
    - If not empty, allocate the first block, remove it from list
    - If empty, try to find a larger free block and split it all the way down to B
    - Update free lists accordingly
- bool my_free(void *user_buffer)
  - Locate header A = user_buffer – sizeof(header)
  - Compute buddy addr A'
    - A_{off} = A – head_pointer, A_{off}' = A_{off} XOR A->block_size
      A' = head_pointer + A_{off}'
  - Merge until we can't go further

init_allocator

```
init_allocator(int b, int M) {
   if (b > M) error;
   b = higher_two(b);
   M = higher_two(M);

   head_pointer = (char*)malloc(M);

   header *h = (header*)head_pointer;
   h->block_size = M;
   h->in_use = false;
   h->next = NULL;

   map[M] = h;
   for (int i = M / 2; i >= b; i /= 2)
      map[i] = NULL;
}
```

validate input

allocate memory

setup free lists

my_malloc

```
my_malloc(int R) {
   int B = higher_two(R + sizeof(header));
   if (B < basic_block_size) B = basic_block_size;
   if (B > M) error;

   header *h = NULL; int S = B;
   while (h == NULL && S <= M) {
      h = map[S];
      S *= 2;
   }
   if (h == NULL) block not found;

   while (h->block_size > B) split_block(h);

   delete(h);
   h->in_use = true;
   return h + 1;
}
```

compute size needed

find a block that satisfies B

split block until getting size B

delete block from free list
give block to user

split_block

```
split_block(header *h) {
   delete(h);        delete from free list

   header *h2 = (header*)((char*)h + h->block_size / 2);
   h->block_size /= 2;                                          } split into two blocks
   h2->block_size = h->block_size;                              } setup two headers
   h->in_use = h2->in_use = false;

   add(h);          } add two new blocks to free list
   add(h2);         }
}
```

my_free

```
my_free(Addr a) {
   header *h = (header*)((char*)a – sizeof(header));
   h->in_use = false;
   add(h);

   while (h->block_size < M) {
      header *buddy = get_buddy(h);
      if (!buddy->in_use && buddy->block_size == h->block_size)
         h = combine_blocks(h, buddy);
      else
         break;    // cannot merge anymore
   }
}
```

add block back to free list

merge if possible

combine_blocks

```
combine_blocks(header *h1, header *h2) {
   delete(h1);
   delete(h2);

   header *h;
   if (h1 < h2) h = h1;
   else h = h2;
   h->block_size *= 2;

   add(h);
   return h;
}
```

delete two small blocks from free list

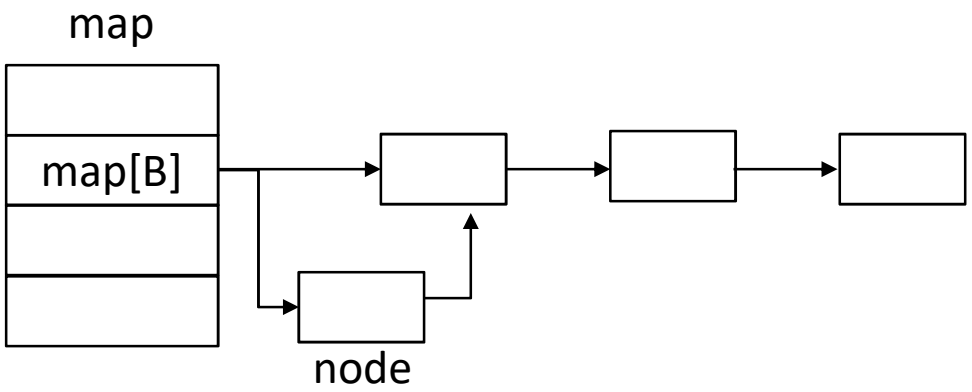
merge two blocks into a large block

add large block back to free list

add

```
add(header *node) {
   int B = node->block_size;
   node->next = map[B];
   map[B] = node;
}
```

insert node into the beginning of the tier with size B
map[B] points to the head of that tier



delete

```
delete(header *node) {
   int B = node->block_size;
   header *head = map[B];

   if (head == node) {
      map[B] = node->next;
   }
   else {
      while (head->next != node) {
         head = head->next;
      }
      head->next = node->next;
   }
}
```