Memory Management

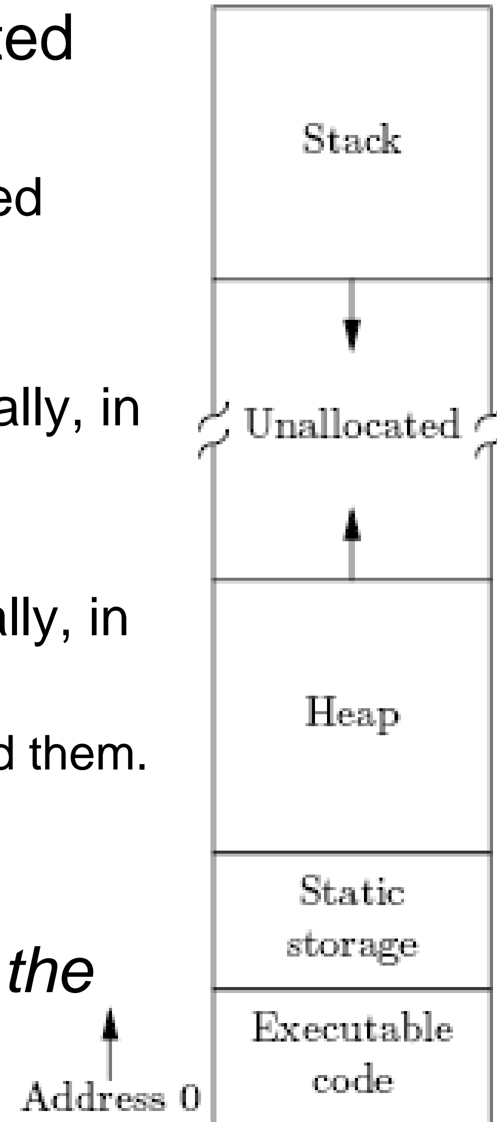
- Memory Areas and their use

- Memory Manager Tasks:
  - acquire
  - release

- Free List Implementations
  - Singly Linked List
  - Doubly Linked List
  - Buddy Systems

Memory Management

- **Memory areas**: In languages like C or Java, the memory used by a program can be allocated from three different areas:
  - **Static**: laid out at compilation time, and allocated when the program starts.
    - Used for Global variables and constants
  - **Stack**: memory is allocated and freed dynamically, in LIFO order.
    - Used for Local variables and parameters
  - **Heap**: memory is allocated and freed dynamically, in any order.
    - Used for data outliving the method which created them. In Java all objects are stored in the heap
- *The memory management techniques we discuss in this lecture apply exclusively to the management of the heap.*



Memory Manager

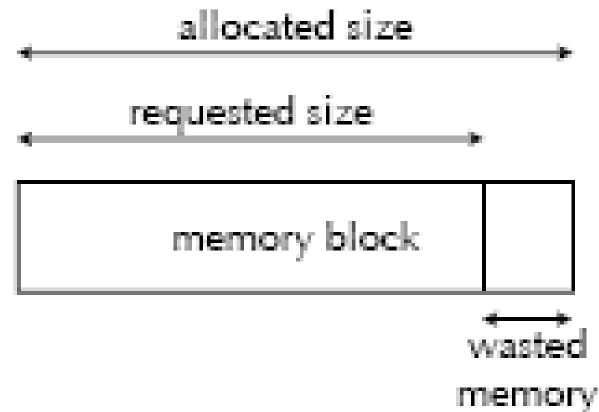
- The **memory manager** is part of the Operating System.
- It must keep track of which parts of the heap are free, and which are allocated.

- A memory manager supports the following operations:
  - **acquire**: allocates memory needed by programs
  - **release**: deallocates memory no longer needed by programs

- It also defragments memory when needed

Problems faced in memory allocation

- Memory fragmentation:
  - **External fragmentation**: Memory wasted outside allocated blocks
  - **Internal fragmentation**: Memory wasted inside allocated block. Results when memory allocated is larger than memory requested.



- Overhead: Additional memory that must be allocated, above and beyond that requested by programs, in order to provide for the management of the heap.

Free List

- Memory manager uses a **free list** data structure that keeps track of free memory blocks in a scheme for dynamic memory allocation.
- Common implementations for free list:
  - Singly-linked list
  - Doubly-linked list
  - Buddy systems: an array of doubly-linked lists

- Allocation Policies:
  - **First fit** chooses the first block in the free list big enough to satisfy the request, and split it.
  - **Next fit** is like first fit, except that the search for a fitting block will start where the last one stopped, instead of at the beginning of the free list.
  - **Best fit** chooses the smallest block bigger than the requested one.
  - **Worst fit** chooses the biggest, with the aim of avoiding the creation of too many small fragments – but doesn’t work well in practice.

Singly-linked list implementation of free-list

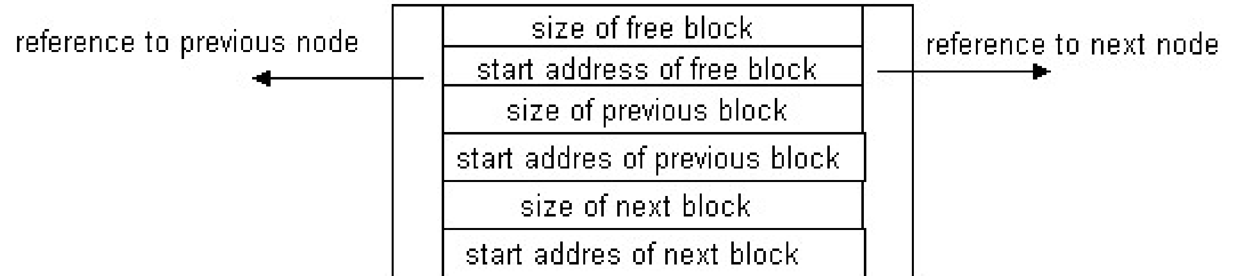
- Each node represents a free block of memory
- Nodes must be sorted according to start addresses of free blocks so that adjacent free memory blocks can be combined.
- acquire( ) and release( ) operations are O(n); where n is the number of blocks in the heap.
- In order to acquire a block, a node is searched following one of the allocation policy. If the block is bigger than requested, it is divided into two. One part is allocated and one remains in the list.
- In order to release a block,
  - a new node must be inserted (if the adjacent block is not on the free list)
  - or a node, which contains the adjacent free block, must be modified.
  - Searching for the place of the new or existing node has complexity O(n).

Doubly-linked list implementation of free-list

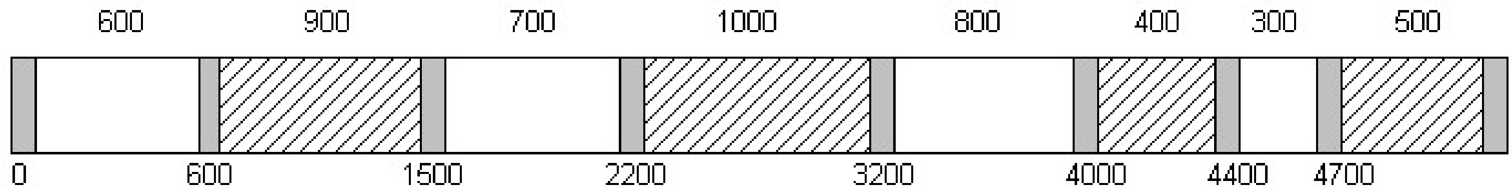
- In this implementation
  - Nodes are not sorted according to start addresses of free blocks.
  - All memory blocks have boundary tags between them. The tag has information about the size and status (allocated/free)
  - Each node in the doubly linked list represents a free block. It keeps size & start address of the free block and start addresses & sizes of the previous and next memory blocks. The adjacent blocks may be or may not be free
- The release operation does not combine adjacent free blocks. It simply prepends a node corresponding to a released block at the front of the free list. This operation is thus O(1). Adjacent free blocks are combined by acquire().
- The acquire operation traverses the free list in order to find a free area of a suitable size. As it does so it also combines adjacent free blocks.

Doubly Linked List Example

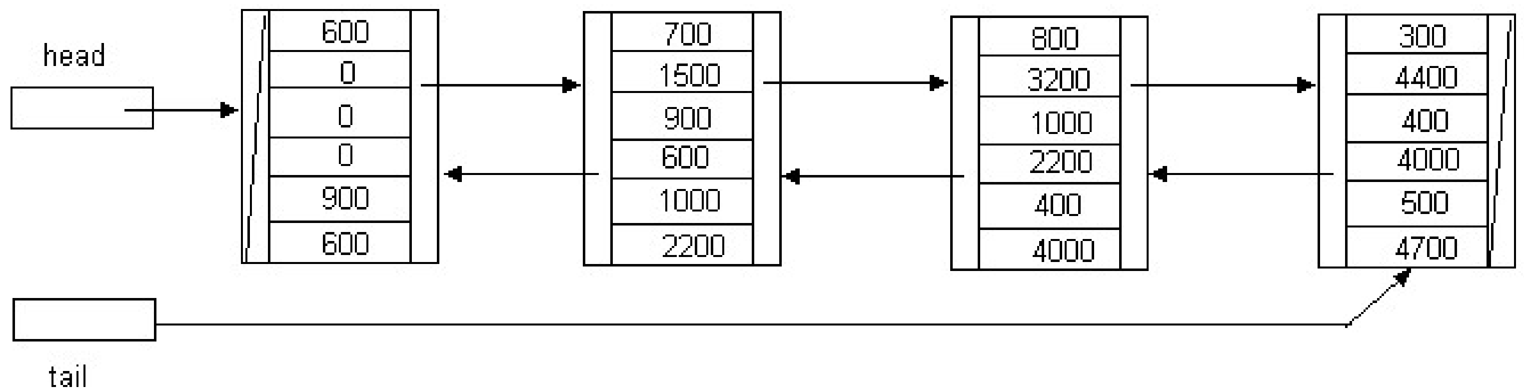
- Node structure:



- Initial state of memory (shaded=allocated, grayed=boundary tags)

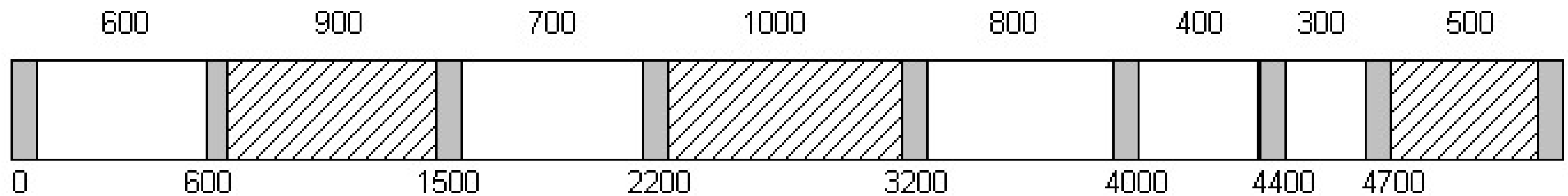


- The corresponding free list

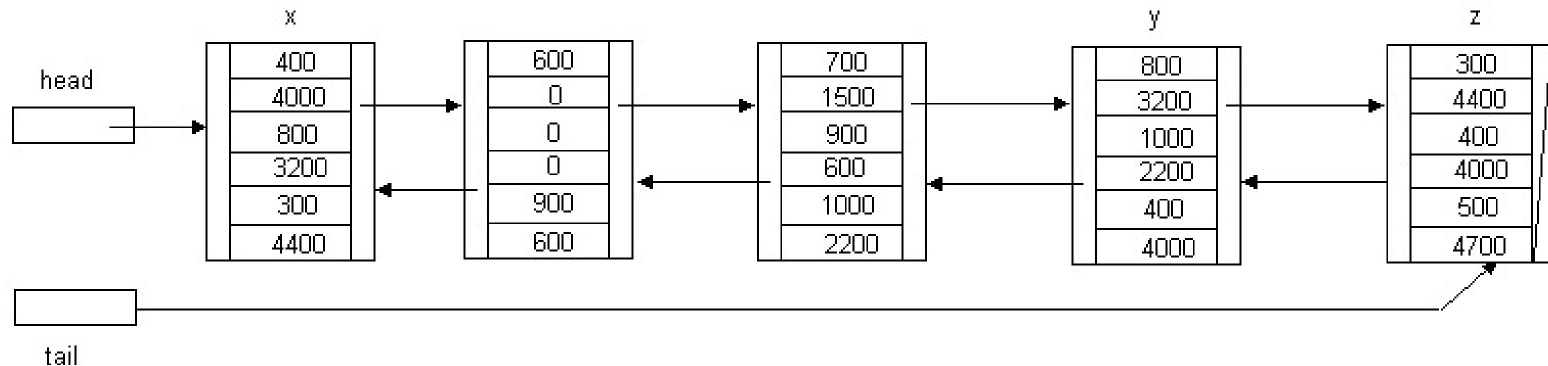


Doubly Linked List Example (Cont.)

- The operation **release(400, 4000)** will result in:

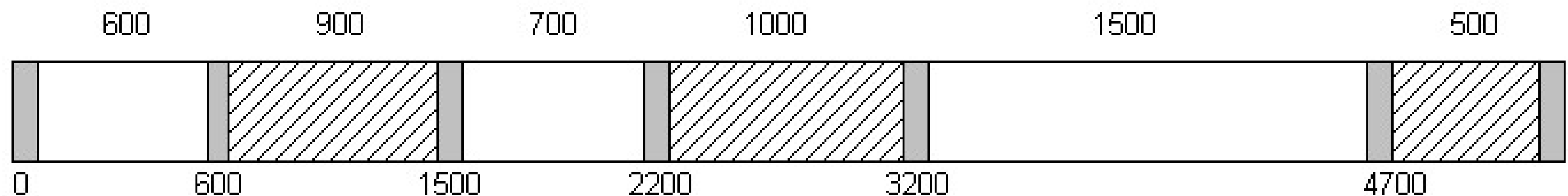


- The node corresponding to the freed block is appended at the front of the **free-list**. The nodes **x**, **y**, and **z** correspond to the three free blocks that have not yet been combined.

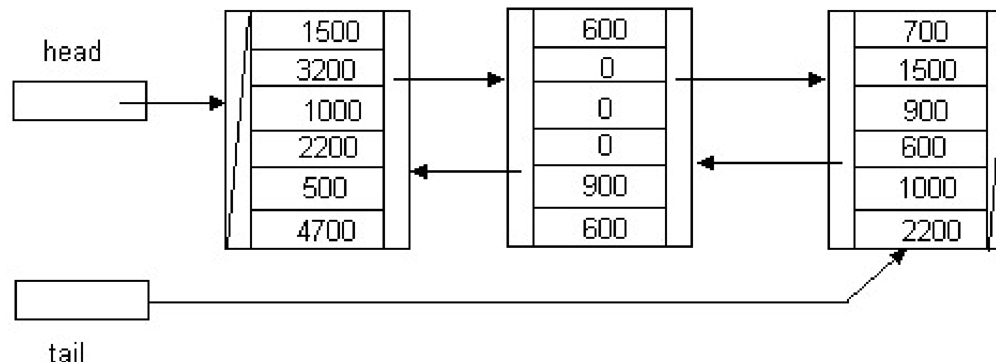


Doubly Linked List Example (Cont.)

- The operation **acquire(600)** using the **first-fit** allocation policy will first result in the combination of the three adjacent free blocks:

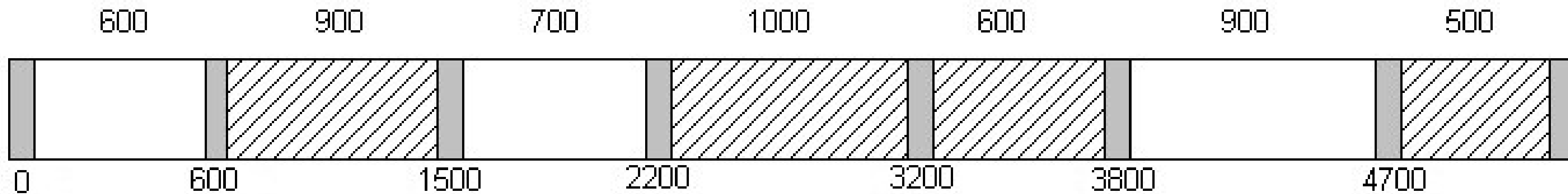


- At this point the corresponding **free list** is:

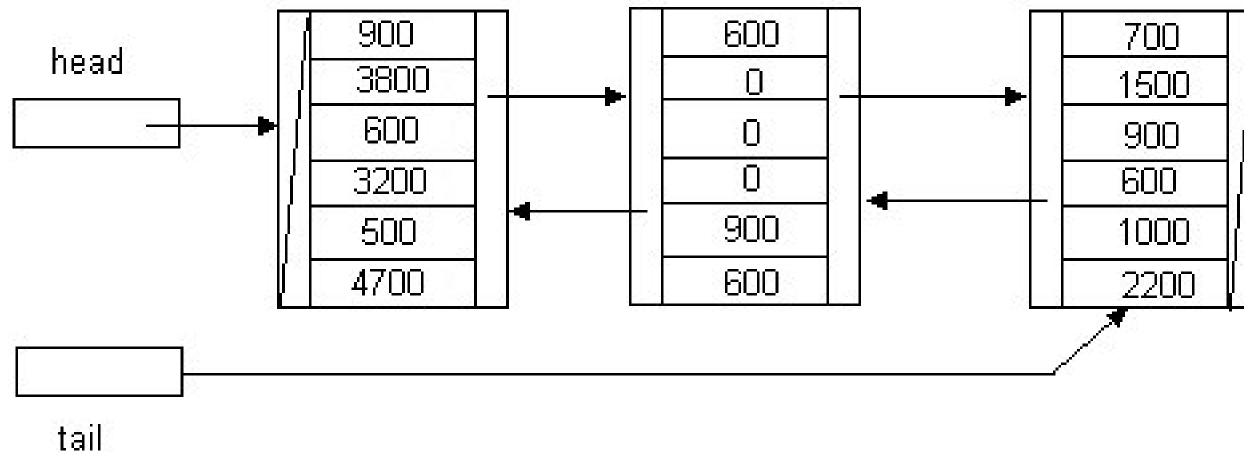


Doubly Linked List Example (Cont.)

- The required **600** bytes are then allocated, resulting in:



- The corresponding **free list** is:



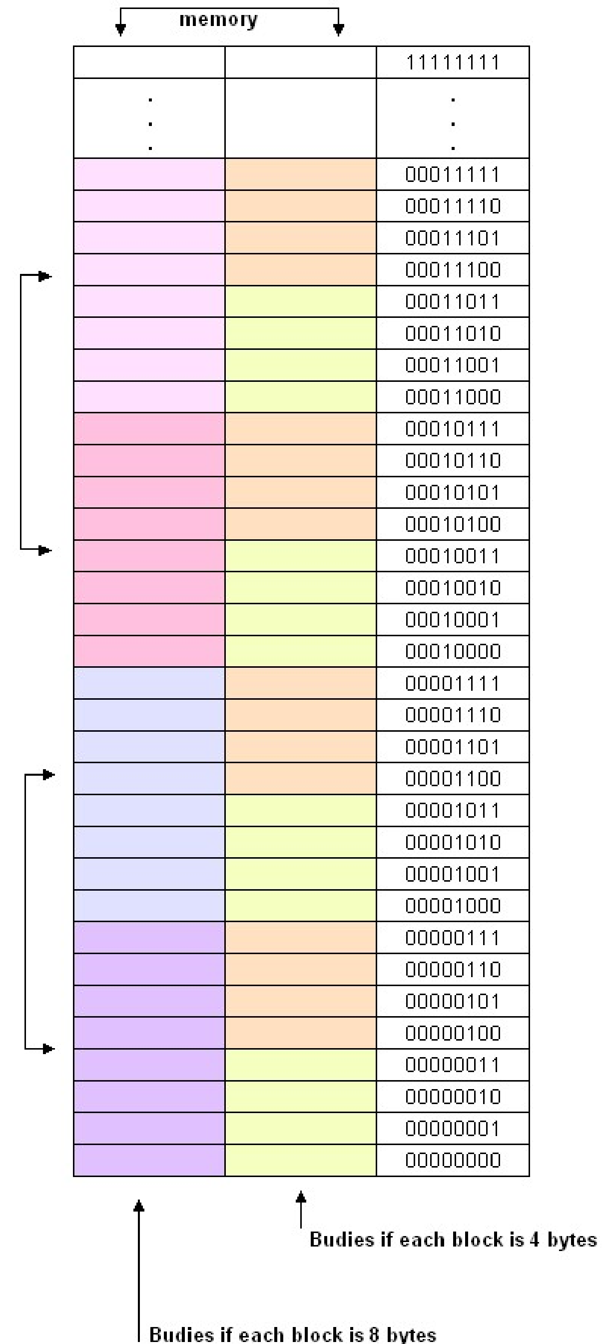
Buddy Systems implementation of free-list

- Instead of having a single free list, it has an array of free lists; each element of the array holding blocks of the same size. One type of buddy systems is the **binary buddy system**.
  - For a memory of size **m**, there are free-lists of size 2^0, 2^1, 2^2, . . . , 2^k, where $m \leq \lceil 2^k \rceil$
  - The heap is viewed as one large block which can be split into two equal smaller blocks, called buddies. Each of these smaller blocks can again be split into two equal smaller buddies, and so on. Each memory block has its “buddy”. The “buddy” of a block of size $\mathbf{2^k}$ that starts at address **x** is the block of size $\mathbf{2^k}$ that start at address **y = complementBit_k(x),** where the address bits are numbered from right to left starting with 0.

Buddies

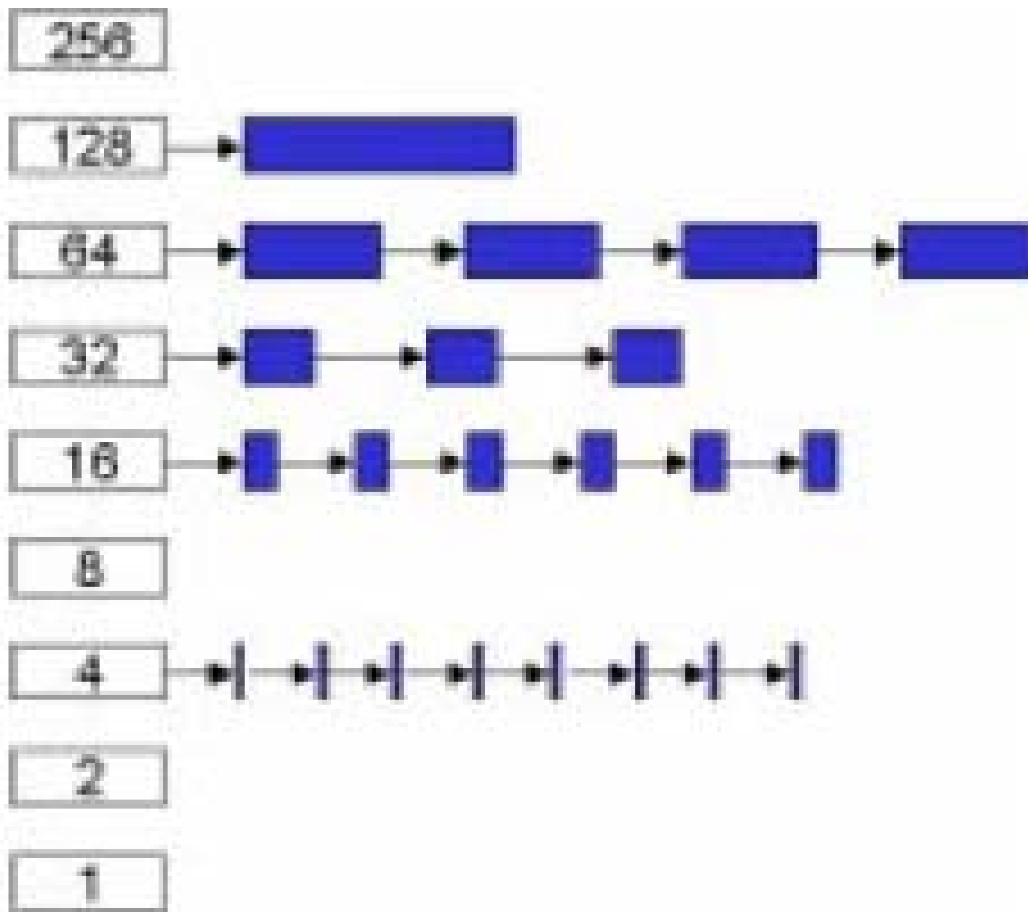
- If each block is of size 8 bytes (i.e., 2^3 bytes); then the buddy of a block is obtained by complementing bit 3 of its starting address. If each block is of size 4 bytes (i.e., 2^2 bytes); then the buddy of a block is obtained by complementing bit 2 of its starting address.

- **Example:** What is the starting address of the buddy of a block that starts at address **1100101010101101** if each block is 16 bytes?
- **Solution**: $16 = 2^4$; the starting address of the buddy is obtained by complementing bit 4: **1100101010111101**



Binary Buddy System implementation of free-list

- Each array element is a list of free blocks of same size. The size of each block is a power of 2.



Binary Buddy System Algorithms

- **acquire(x):** x <= 2^k**,** the corresponding free list is searched
  - If there is a block in this list, it is allocated;
  - otherwise a block of size $\mathbf{2^{k+1}}$, $\mathbf{2^{k+2}}$, and so on is searched and taken off the free list. The block is divided into two buddies. One buddy is put on the free list for the next lower size and the other is either allocated or further splinted if needed.
- **release(x):** The block is placed back in the free list of its size, and
  - if its buddy is also free they are combined to form a free block of size $\mathbf{2^{k+1}}$. This block is then moved to the corresponding free list.
  - If its buddy is free they are combined to form a free block of size $\mathbf{2^{k+2}}$, which is then moved to the appropriate free list and so on.

Buddy Systems Advantages/Disadvantages

- Advantage:
  - Both **acquire( )** and **release( )** operations are fast.
- Disadvantages:
  - Only memory of size that is a power of 2 can be allocated $\Rightarrow$ internal fragmentation if memory request is not a power of 2.
  - When a block is released, its buddy may not be free, resulting in external fragmentation.