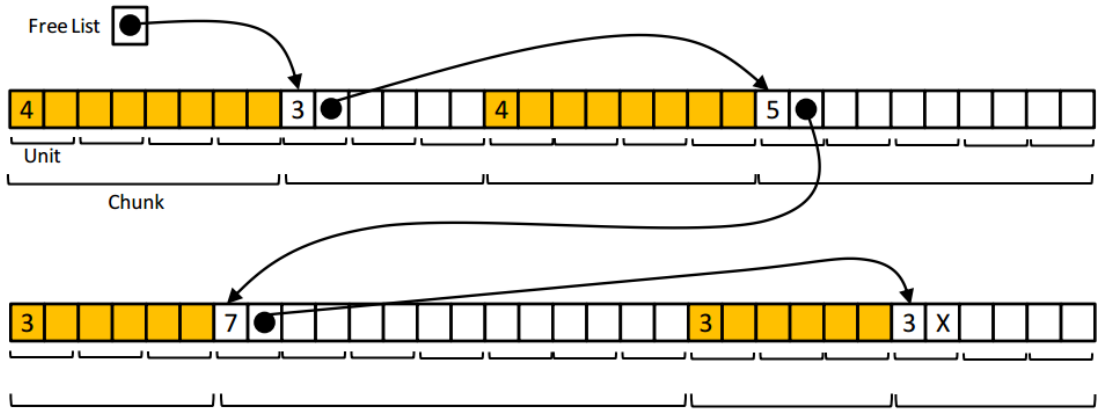
Precept:
Heap Manager Assignment

Insu Yun

Baseline Implementation

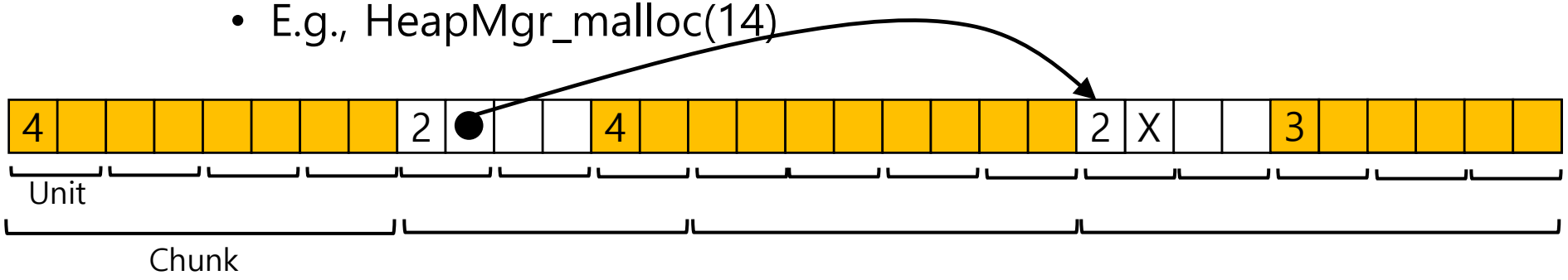
- Data structure
  - Each box consists of 4 bytes
  - Each Chunk's header Unit contains a length and, if the Chunk is free, a pointer to the next Chunk in the Free List.
  - Chunks in the Free List are in increasing order of memory address.
  - A global variable points to the first Chunk in the Free List.



Baseline Implementation

- void *HeapMgr_malloc(size_t uiBytes)
  - If this is the first call → initialize the heap manager.
  - Determine the number of units the new chunk should contain.
  - For each chunk in the free list:

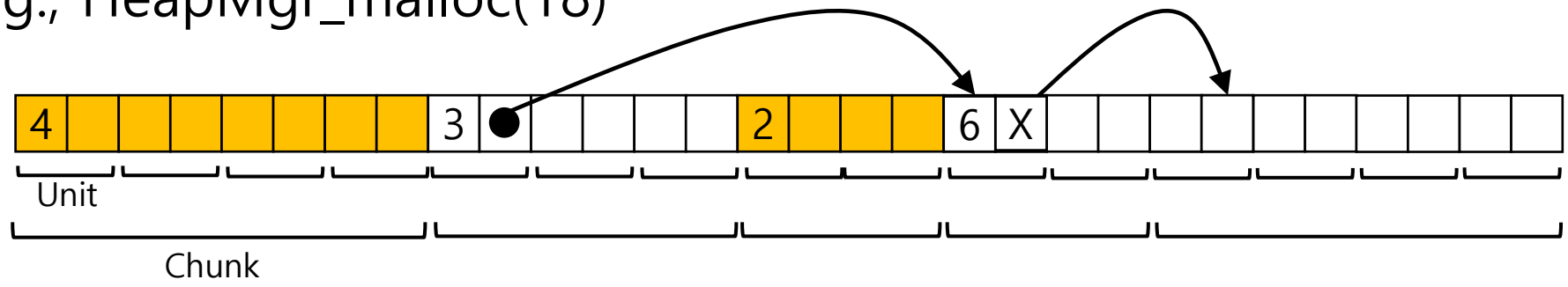
    If the current free list chunk is big enough:
    - If the current free list chunk is close to the requested size, then remove it from the free list and return it. If the current free list chunk is too big, then split the chunk and return the tail end of it. The free list need not be altered.
    - E.g., HeapMgr_malloc(14)



Baseline Implementation

- Ask the OS for more memory enough for the new chunk. Return NULL if the OS refuses. Create a new chunk using that memory. Insert the new chunk at the end of the free list. If appropriate, coalesce the new chunk and the previous one. Let the current free list chunk be the last one.

- E.g., HeapMgr_malloc(18)



Baseline Implementation

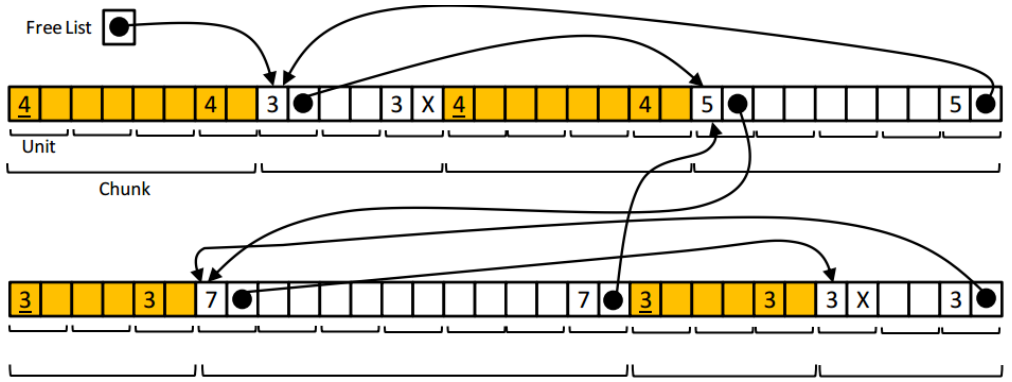
- If the current free list chunk is close to the requested size, then remove it from the free list and return it. If the current free list chunk is too big, then split the chunk and return the tail end of it. In the latter case, the free list need not be altered.

Baseline Implementation

- void HeapMgr_free(void *pvBytes)
  - Traverse the free list to find the correct spot for the given chunk.
  - Insert the given chunk into the free list at the correct spot.
  - If appropriate, coalesce the given chunk and the previous one.
  - If appropriate, coalesce the given chunk and the next one.

First Implementation

- Data structure
  - Each Chunk's header Unit contains a status (INUSE or FREE), a length, and, if the Chunk is free, a pointer to the next Chunk in the Free List.
  - Each Chunk's footer Unit contains a length and, if the Chunk is free, a pointer to the previous Chunk in the Free List.
  - The Chunks in the Free List are in no particular order.
  - '_' means INUSE; absence of '_' means FREE.



First Implementation

- void *HeapMgr_malloc(size_t uiBytes)
  - If this is the first call → initialize the heap manager.
  - Determine the number of units the new chunk should contain.
  - For each chunk in the free list:

    If the current free list chunk is big enough:
    - If the current free list chunk is close to the requested size, then remove it from the free list, set its status to INUSE, and return it. If the current free list chunk is too big, then remove it from the free list, split the chunk, insert the tail end of it into the free list, set the status of the front end of it to INUSE, set the status of the tail end of it to FREE, and return the front end of it.

First Implementation

- Ask the OS for more memory — enough for the new chunk. Return NULL if the OS refuses. Create a new chunk using that memory. Insert the new chunk into the free list. If appropriate, coalesce the new chunk and the previous one in memory. To do so, remove both chunks from the free list, coalesce them to form a larger chunk, and insert the larger chunk into the free list. Let the current free list chunk be the new chunk.
- If the current free list chunk is close to the requested size, then remove it from the free list, set its status to INUSE, and return it. If the current free list chunk is too big, then remove it from the free list, split the chunk, insert the tail end of it into the free list, set the status of the front end of it to INUSE, set the status of the tail end of it to FREE, and return the front end of it.

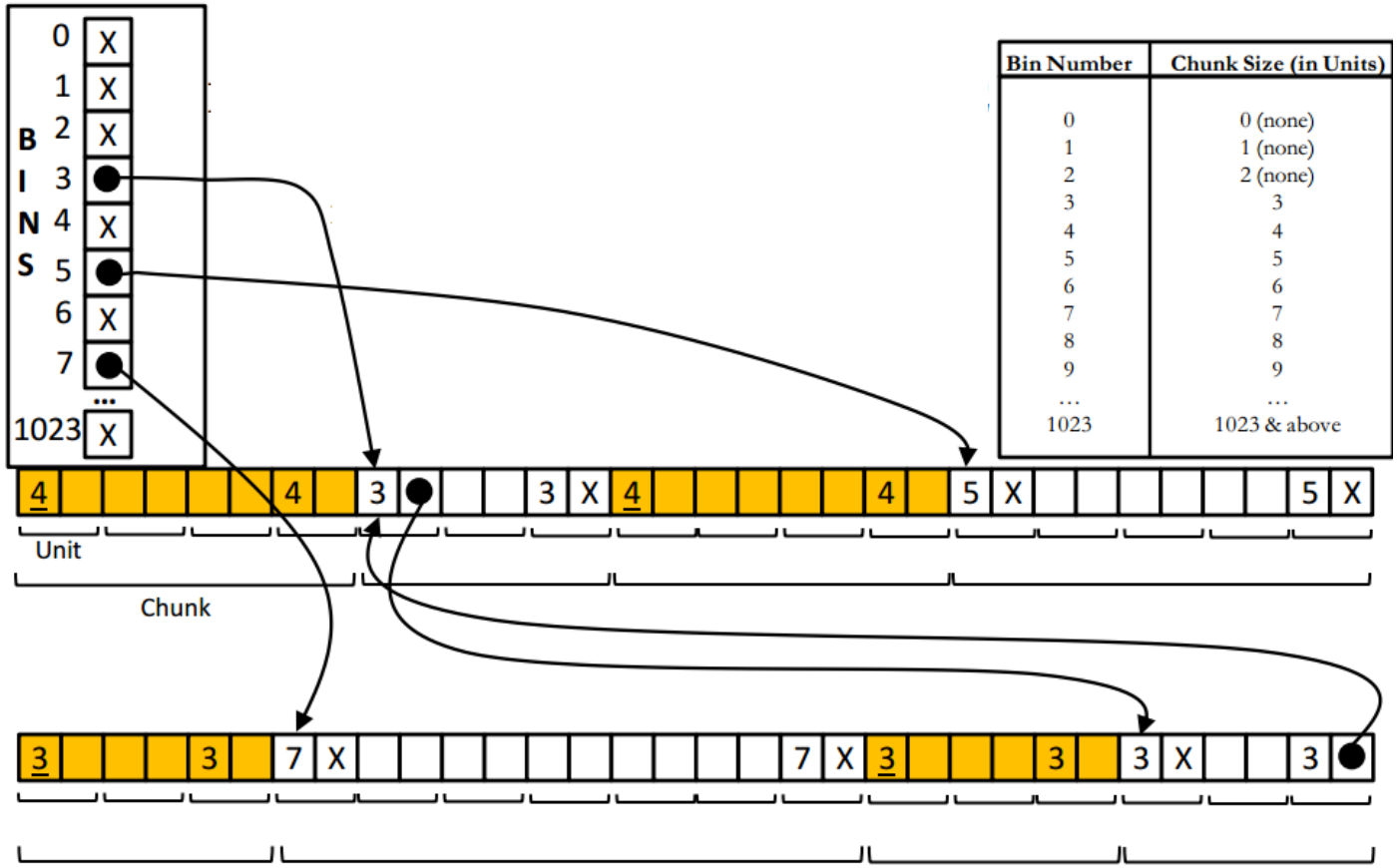
First Implementation

- void HeapMgr_free(void *pvBytes)
  - Set the status of the given chunk to FREE.
  - Insert the given chunk into the free list.
  - If appropriate, coalesce the given chunk and the previous one in memory. To do so, remove both chunks from the free list, coalesce them to form a larger chunk, and insert the larger chunk into the free list.
  - If appropriate, coalesce the given chunk and the next one in memory. To do so, remove both chunks from the free list, coalesce them to form a larger chunk, and insert the larger chunk into the free list.
- Note: Do not decrease the program counter (using extra sbrk call) even after free of the last located chunk. – It will be used for calculating total memory usage.

Second Implementation

- Data structure
  - Each Chunk's header Unit contains a status (INUSE or FREE), a length, and, if the Chunk is free, a pointer to the next Chunk in its Bin.
  - Each Chunk's footer Unit contains a length and, if the Chunk is free, a pointer to the previous Chunk in its Bin.
  - The Chunks in the Bins are in no particular order.
  - '_' means INUSE; absence of '_' means FREE.

Second Implementation



Second Implementation

- void *HeapMgr_malloc(size_t uiBytes)
  - If this is the first call → initialize the heap manager.
  - Determine the number of units the new chunk should contain.
  - For each bin from the start bin to the last bin...

    For each chunk in the current bin...

    If the current chunk is big enough...
    - If the current chunk is close to the requested size, then remove it from its bin, set its status to INUSE, and return it. If the current chunk is too big, then remove it from its bin, split the chunk, insert the tail end of it into the proper bin, set the status of the front end of it to INUSE, set the status of the tail end of it to FREE, and return the front end of it.

Second Implementation

- Ask the OS for more memory — enough for the new chunk. Return NULL if the OS refuses. Create a new chunk using that memory. Insert the new chunk into the proper bin. If appropriate, coalesce the new chunk and the previous one in memory. To do so, remove both chunks from their bins, coalesce them to form a larger chunk, and insert the larger chunk into the proper bin. Let the current chunk be the new chunk.
- If the current chunk is close to the requested size, then remove it from its bin, set its status to INUSE, and return it. If the current chunk is too big, then remove it from its bin, split the chunk, insert the tail end of it into the proper bin, set the status of the front end of it to INUSE, set the status of the tail end of it to FREE, and return the front end of it.

Second Implementation

- void HeapMgr_free(void *pvBytes)
  - Set the status of the given chunk to FREE.
  - Insert the given chunk into the proper bin.
  - If appropriate, coalesce the given chunk and the previous one in memory. To do so, remove both chunks from their bins, coalesce them to form a larger chunk, and insert the larger chunk into the proper bin.
  - If appropriate, coalesce the given chunk and the next one in memory. To do so, remove both chunks from their bins, coalesce them to form a larger chunk, and insert the larger chunk into the proper bin.