CMSC 420 – 0201 – Fall 2019
Lecture 15

Memory Management

Memory Management

- When you do: `Node p = new Node()`, what is the operating system doing?
- Memory management:
  - Used by operating systems and run-time systems for programming languages
  - How are variables stored?
    - static: Fixed memory location
    - stack: Local variables and parameters for functions - Transient
      - Pushed when function is invoked / Popped when function returns
    - heap: Objects created via new (in Java, C++, …) and malloc (in C) – Persistent
      - C/C++ - Object exists until explicitly deleted (or freed)
      - Java/Python - Object exists until no longer referenced (and then subject to garbage collection)

Memory Management Approaches

Explicit Memory Allocation

- Memory is allocated via `new` (in object-oriented languages) or block allocation function like `malloc` (non object-oriented languages)
- …and released via `delete` (C++) or `free` (C).
- Issues:
  - Provides programmer with more control (good)
  - Memory leak: Forgetting to delete – Allocated memory block with no way of access (bad)
  - Dangling pointers: (bad)
    - A pointer that references a deleted block of memory
    - Often the result of aliasing (two pointers referring to the same object) and/or shallow copying (copying pointers, not contents)

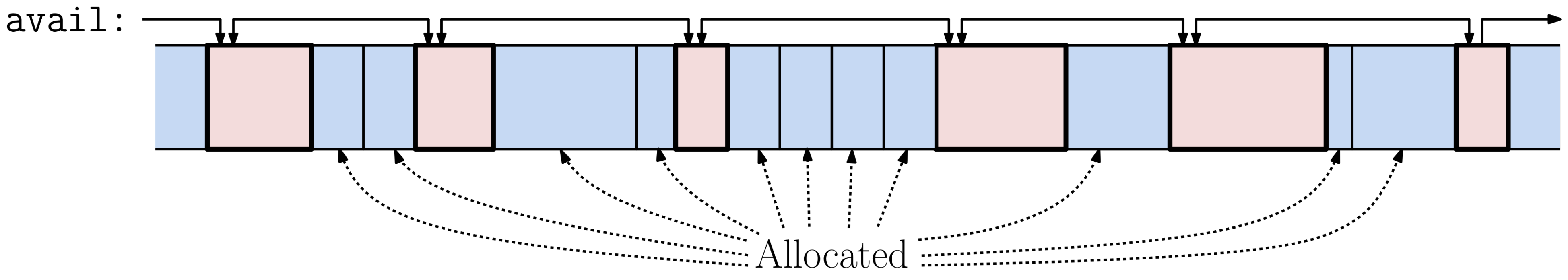
Memory Management Approaches

Implicit Memory Allocation

- Memory is allocated via `new` (as in Java) or just pops into existence (Python)
- When an object is unreachable (directly or indirectly), its space is reclaimed via garbage collection
- Issues:
  - No dangling pointers/memory leaks (good)
  - Compact memory to improve memory locality (good)
  - Less control for the programmer (may be bad)
  - Garbage collection takes time and occurs unpredictably
    - Problematic for real-time systems
    - Ameliorated by incremental garbage collection

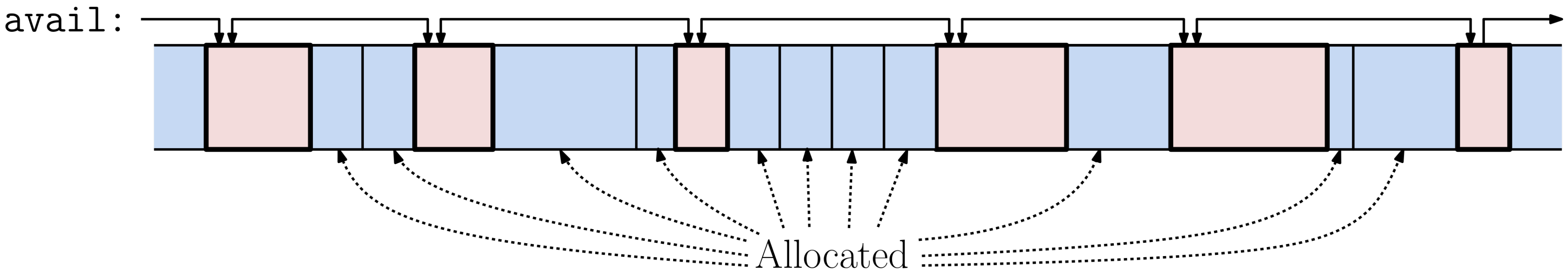
Explicit Memory Allocation - Overview

- Memory is divided into variable-sized blocks
- Blocks are marked as either available or in-use (allocated)
  - Initially there is one huge available block
  - As blocks are allocated/deallocated, memory becomes fragmented, like swiss cheese
  - Available blocks are maintained in a doubly linked list: `avail`



Explicit Memory Allocation - Overview

- Which available block to select?
  - First-fit: The first block on the available list that is large enough
  - Best-fit: The block that most closely fits the requested size (and is large enough)
- Which is better?
  - First-fit usually wins: Faster and tends to avoid small residual fragments (slivers)
  - Sliver avoidance: If block is just slightly larger than request, don't split it



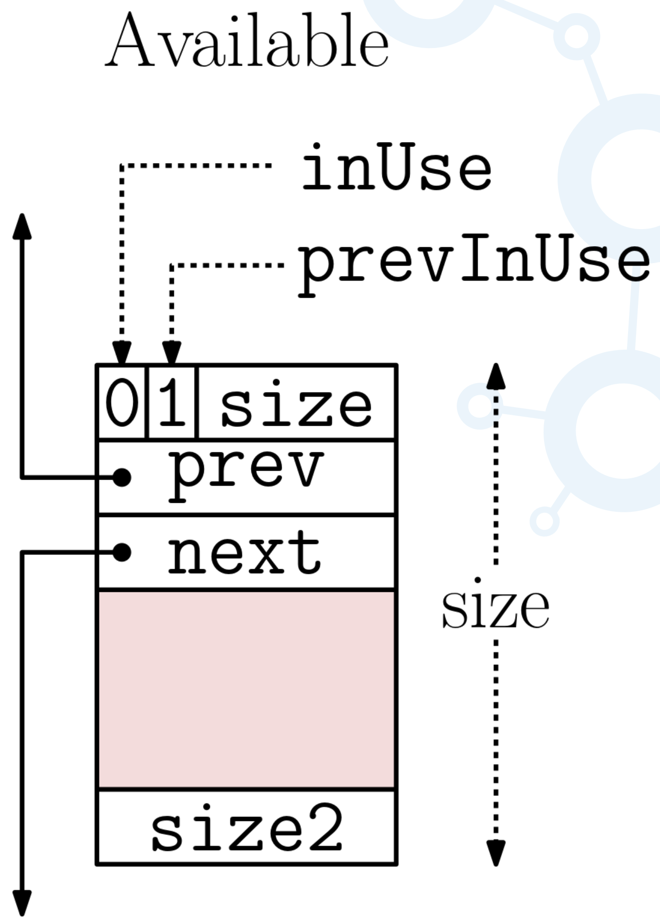
Notation and Assumptions

- Blocks are often aligned at word (32-bit) or double-word (64-bit) boundaries
  - Can be used for storing any type of data (byte, int, float, double)
- Pointers and pointer arithmetic:
  - A pointer to a generic word of memory of type: `void*`
  - Given pointer `p`:
    - `p+i`: is `i` words beyond `p`'s location
    - `*p`: is the value at this memory location

Block Structure

Available Block

- Each available block stores:
  - `size`: The size of the block, including these additional fields
  - `inUse`: A bit set to 0 (false)
  - `prevInUse`: A bit set to 1 (true) if the immediately preceding block in memory (not the same as `prev`) is in-use
  - `prev`: A pointer to the head of the previous available block
  - `next`: A pointer to the head of the next available block
  - `size2`: Stores the same value as `size`
- Notes:
  - `prev` and `next` need not be previous and next according to the physical memory layout
  - `p.size2` can be accessed as `*(p + p.size - 1)`



Block Structure

Allocated Block

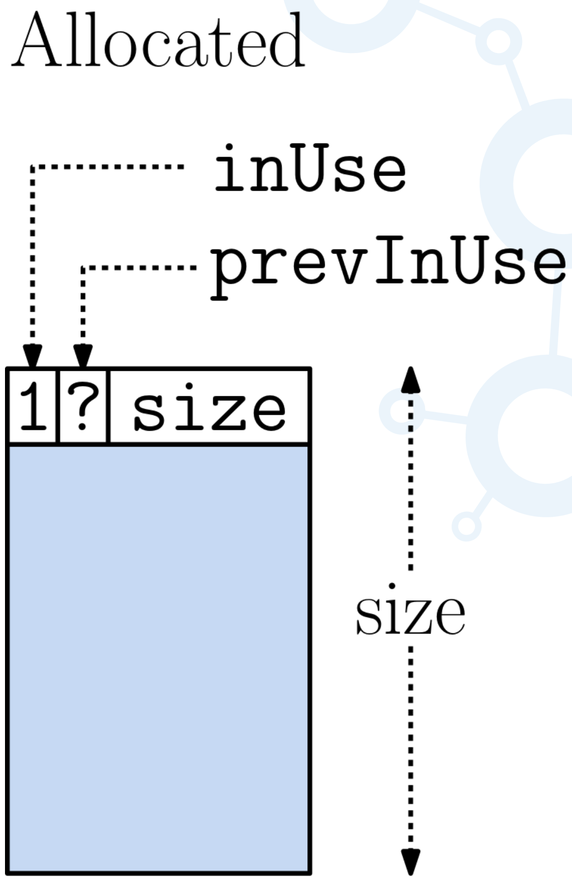
- Each allocated block stores:

  `size`: The size of the block, including these additional fields

  `inUse`: A bit set to 1 (true)

  `prevInUse`: A bit set to 1 (true) if the immediately preceding block in memory (not the same as `prev`) is in-use

- Note:
  - We incur an overhead of just one word for each allocated block
  - What's to keep the user from altering the header fields and undermining the system's integrity?
    - Usually nothing! - Segmentation fault soon follows
    - Buffer-overflow is a major security risk



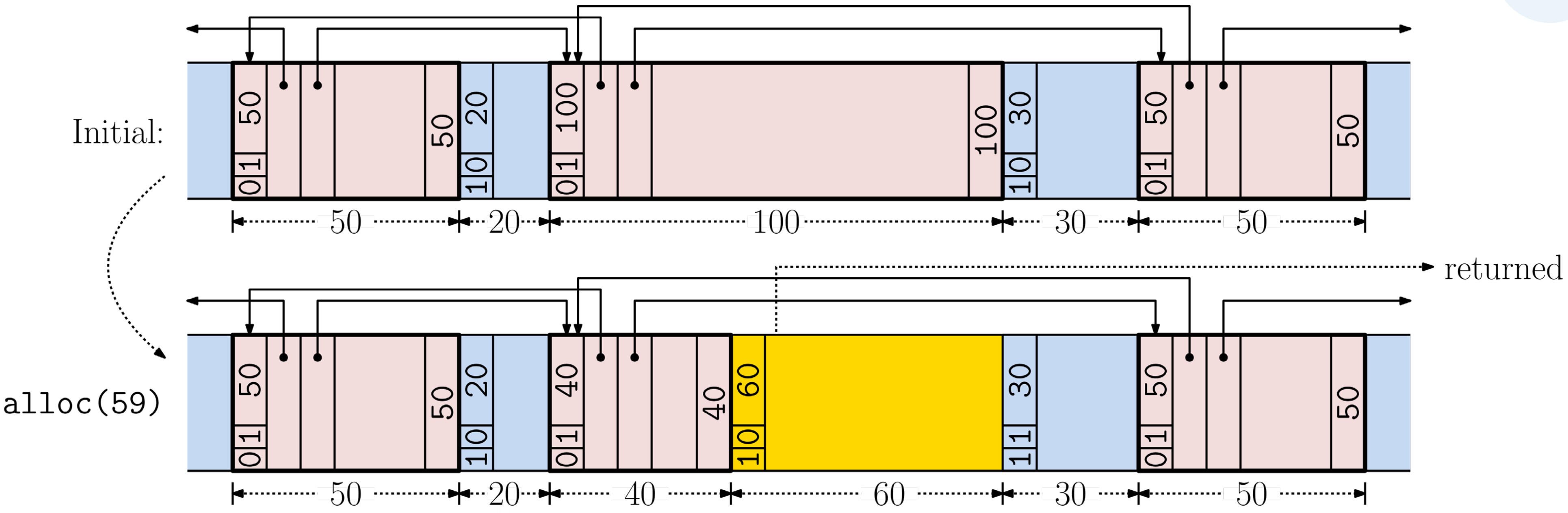
Allocation

- Allocate a block of size b:
    - Increase b by one to account for header
    - p ← Search avail list for appropriate block (by either First- or Best-fit)
    - If (p's size matches b (or is sufficiently close)):
        - Use entire block (unlink from available list)
    - Else:
        - Trim off a subblock of size b from the back of this block
        - Initialize its header
        - Adjust the size of the remaining block (and leave in available list)

Allocation

```
(void*) alloc(int b) {                      // allocate block with b words
    b += 1;                                 // extra space for system overhead
    p = search available space list for block of size at least b;
    if (p == null) { ...Error! Insufficient memory...}
    if (p.size - b < TOO_SMALL) {           // remaining fragment too small?
        avail.unlink(p);                    // remove entire block from avail list
        q = p;                              // this is block to return
    }
    else {                                  // split the block
        p.size -= b;                        // decrease size by b
        *(p + p.size - 1) = p.size;         // set new block's size2 field
        q = p + p.size;                     // offset of start of new block
        q.size = b;                         // size of new block
        q.prevInUse = 0;                    // previous block is unused
    }
    q.inUse = 1;                            // new block is used
    (q + q.size).prevInUse = 1;             // adjust prevInUse for following block
    return q + 1;                           // offset the link (to avoid header)
}
```

Allocation Example



Deallocation

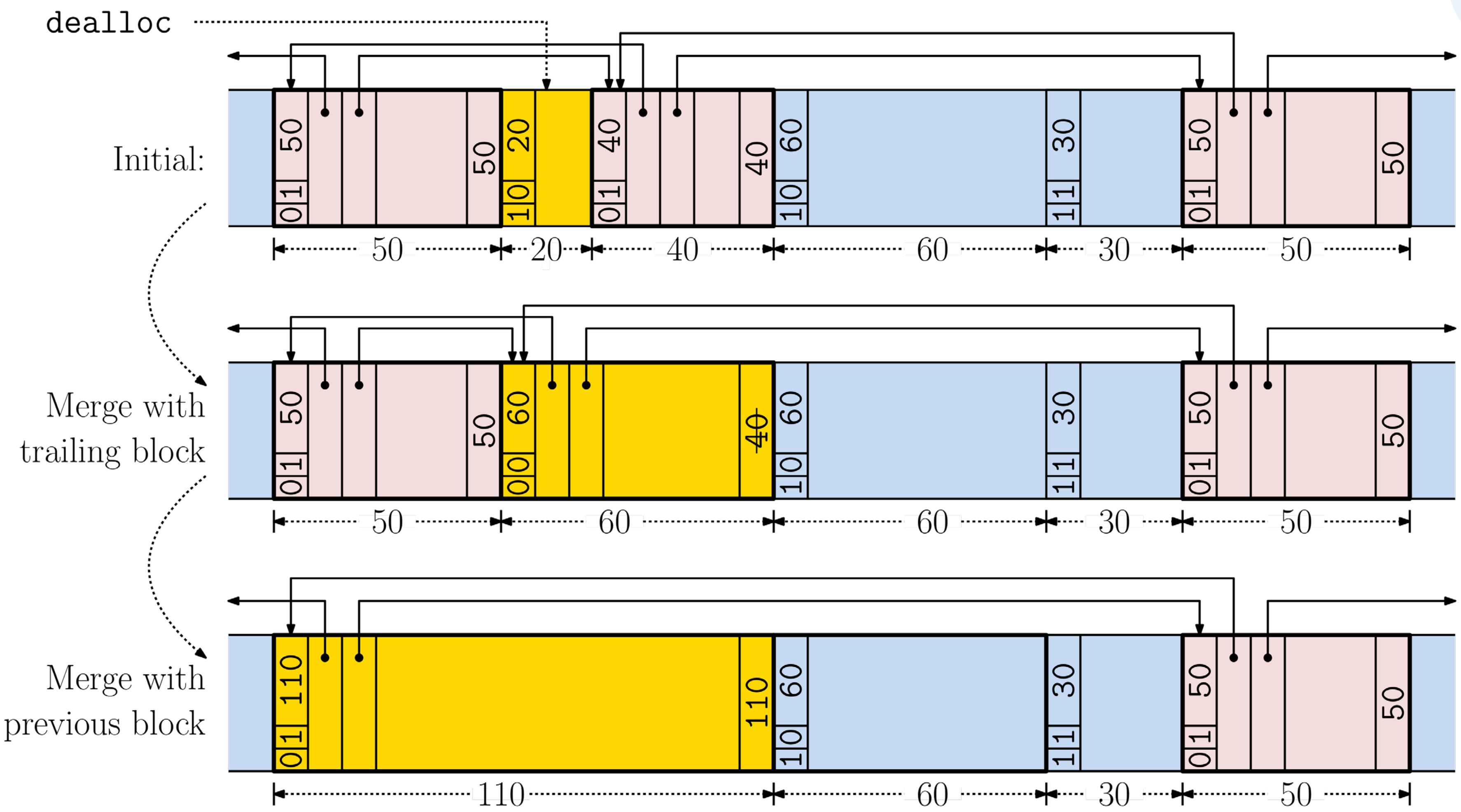
- Deallocate a block p:
  - Decrement p by one so it points to the header
  - If (immediately following block is not in-use):
    - Merge with this block (we are now in the available list)
  - Else:
    - Insert ourselves into the available list
  - If (immediately preceding block is not in-use):
    - Merge with this block, and adjust headers
    - Remove ourselves from the available list

Deallocation

```
delete(void* p) {                           // delete block at p
    p--;                                    // back up to the header
    q = p + p.size;                         // the immediately following block
    if (!q.inUse) {                         // is it available?
        p.size += q.size;                   // …merge q into p
        avail.move(q, p);                   // move q to p in avail space list
    }
    else avail.insert(p);                   // insert p into avail space list
    p.inUse = 0;                            // p is now available
    *(p + p.size - 1) = p.size;             // set our size2 value

    if (!p.prevInUse) {                     // previous is available?
         q = p - *(p-1);                    // get previous block using size2
         q.size += p.size;                  // merge p into q
         *(q + q.size - 1) = q.size;        // store new size2 value
         avail.unlink(p);                   // unlink p from avail space list
         (q + q.size).prevInUse = 0;        // notify next that we are avail
     }
}
```

Deallocation Example



Analysis

- No theoretical analysis of performance
- Empirical studies show:
  - First-fit usually outperforms best-fit (faster and less fragmentation)
  - User has ultimate control
    - You can allocate a huge chunk of memory and do your own memory allocation
- External Fragmentation:
  - Wastage between blocks due to memory being cut up like swiss cheese
  - Can ameliorate this by forcing blocks to be of uniform sizes that merge nicely (e.g., powers of 2), but this leads to…
- Internal Fragmentation:
  - Wastage within blocks due to forcing blocks to have uniform sizes

Buddy System

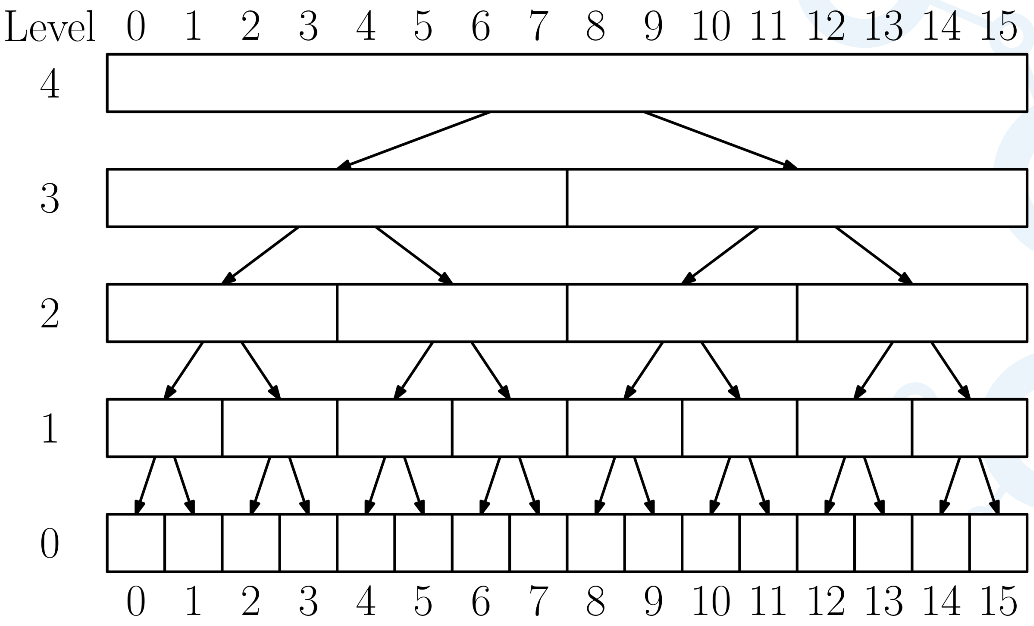
Coping with external fragmentation

- The memory-management system described above suffers from fragmentation:
  - Small residual blocks of available memory that are too small to fulfill requests
  - Scattered like holes in a block of swiss cheese
- Alternative:
  - Force blocks to be a given allowed set of sizes (e.g., powers of 2)
  - Now, blocks split and merge nicely (e.g., $8 \rightarrow 4+4$ and $4+4 \rightarrow 8$)
    - Reduces external fragmentation
  - If a request is not of this size, round it up to the next larger allowed size
    - Induces internal fragmentation

Buddy System

Coping with external fragmentation

- Start with a large block of size 2^m
- Blocks are formed by repeated bisection
- Blocks at level k have size 2^k
- A block of size 2^k starts at an address that is a multiple of 2^k



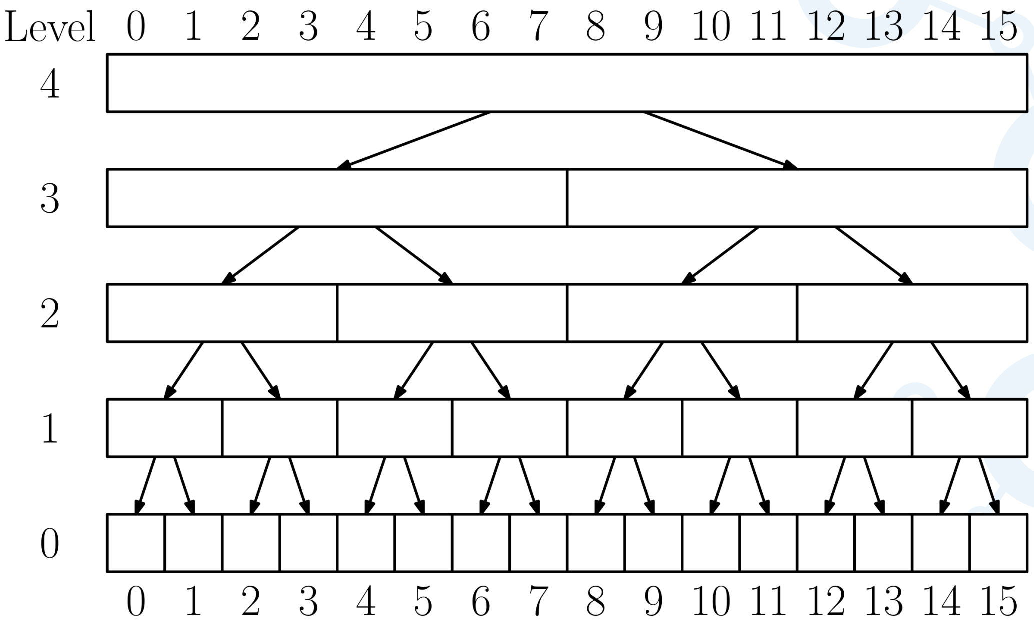
Buddy System

Coping with external fragmentation

- The sibling of a block is called its buddy
- Can be computed arithmetically

$$\text{buddy}_k(x) = \begin{cases} x + 2^k & \text{if } 2^{k+1} \text{divides } x \\ x - 2^k & \text{otherwise} \end{cases}$$

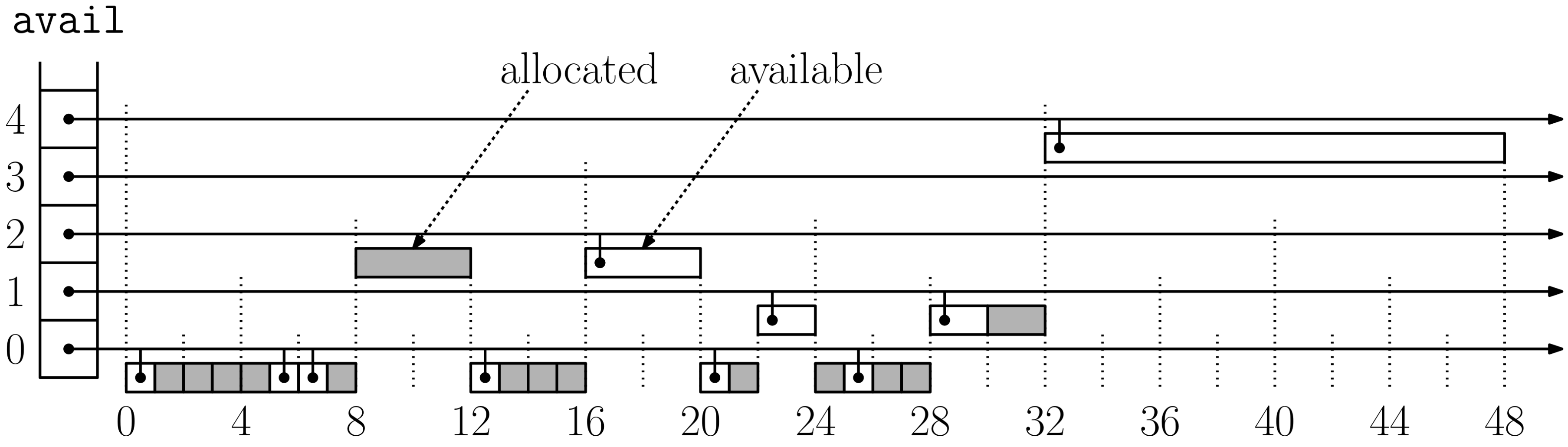
- Toggle the kth bit of x in binary:
  - $\text{buddy}_2(12) = \text{buddy}_2(001100) = 001000 = 8$
  - $\text{buddy}_3(80) = \text{buddy}_3(1010000) = 1011000 = 88$
  - Java: `buddy(k,x)=(1<<k)^x`



Buddy System

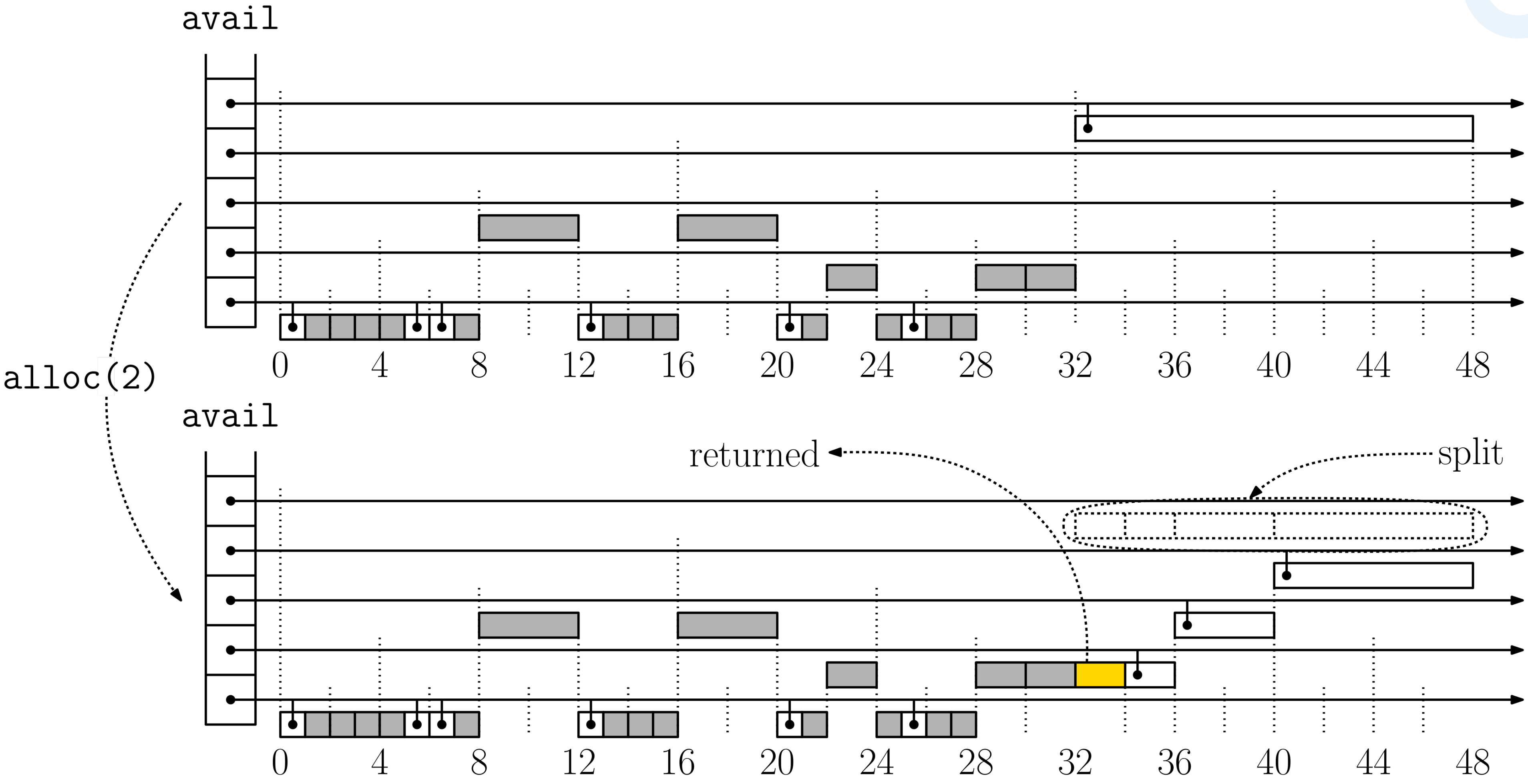
The Bigger Picture

- All allocation requests are rounded up to size 2^k
- Array of doubly linked lists of available blocks: avail[k] has blocks of size 2^k
- `p←alloc`(2^k): Find block of sufficiently large size. Subdivide if needed.
- dealloc(p): Make block available. Merge (repeatedly) with buddies.



Buddy System

Example of Allocation: `alloc(2)`



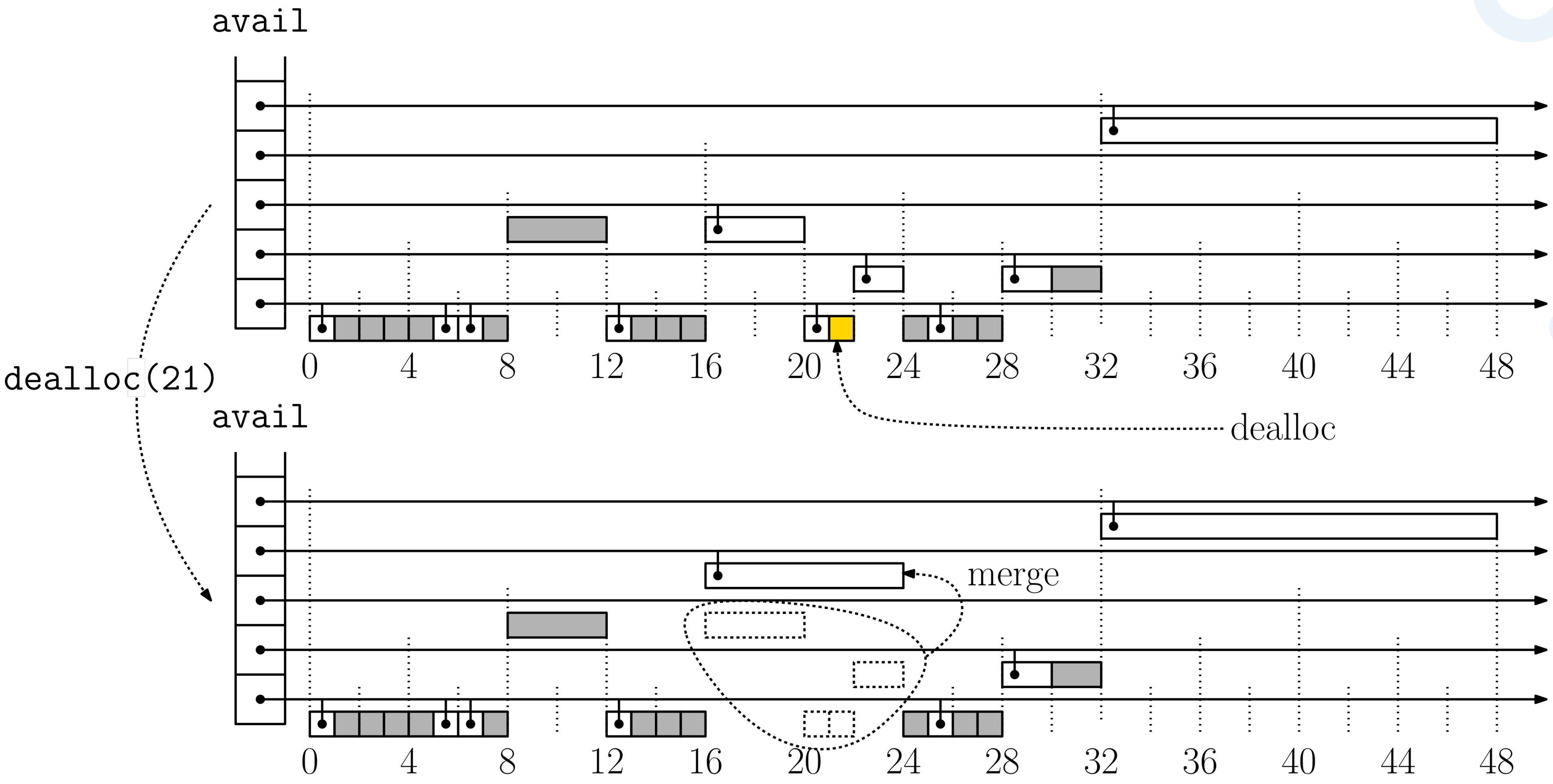
Buddy System

Allocation

- `alloc(b)`:
  - Let $k = \lceil \lg(b+1) \rceil$. Allow 1 word for header, and round to next higher power of 2.
  - Targe size: 2^k
  - Find smallest $j \geq k$ such that `avail[j]` is nonempty and remove any block: size 2^j
  - Repeatedly split until we have a block of size 2^k.
    - E.g., if $2^k = 2$ and $2^j = 16$, we split to sizes: $16 = 8 + 4 + 2 + 2$
  - Keep one block p of size 2^k and insert the others in the appropriate `avail` lists
  - Return a pointer to block p

Buddy System

Example of Deallocation



Buddy System

Deallocation

- `dealloc(p):`
    - Let $k = \lg(p.\mathrm{size})$, that is, $p.\mathrm{size} = 2^k$
    - Mark block p as available
    - Repeat:
        - Let $p' = \mathrm{buddy}_k(p)$
        - If block p' is allocated, break (merge is not possible)
        - Otherwise (merge is possible)
            - Remove p' from `avail[k]`
            - `Merge` p and p' into a new block of size 2^{k+1}
            - Let p point to this new block
    - Insert p into appropriate avail list

Summary

- Variant: Fibonacci Buddy System
  - Uses Fibonacci numbers, rather than powers of 2
  - $F(0) = 0,\ F(1) = 1,\ F(i) = F(i-1) + F(i-2)$
  - `avail[k]` stores available blocks of size $F(k)$
  - Round each request up to next larger Fibonacci number
  - If no available block of this size, find next larger available size $F(j)$
  - Split this block repeatedly:
    - E.g., Want a block of size $F(3) = 2$ but next available block is of size $F(9) = 34$. Split it into $34 = 2 + 3 + 8 + 21 = F(3) + F(4) + F(6) + F(8)$. Return block $F(3)$, and add others to `avail[4]`, `avail[6]`, and `avail[8]`, respectively.
  - Intuition: Less fragmentation because Fibonacci numbers are denser

Summary

- We have seen two common memory allocation systems
- Standard allocator
  - Uses blocks of arbitrary sizes
  - Maintains a linked list of available blocks
  - Small residual blocks can clog things up, causing external fragmentation
- Buddy system
  - Allocates blocks in a binary hierarchy, uses only blocks of size 2^k
  - Requests must be rounded up to next larger power of 2: Causes internal fragmentation
  - Reduces external fragmentation
  - Variant: Fibonacci Buddy