CMSC 420: Lecture 19
Memory Management

Memory Management: One of the major systems issues that arises when dealing with data structures is how storage is allocated and deallocated as objects are created and destroyed. Although *memory management* is really a operating systems issue, we will discuss this topic because there are a number interesting data structures issues that arise. Sometimes for the sake of efficiency, it is desirable to design a special-purpose memory management system for your data structure application, rather than using the system's memory manager.

System-Level: We will not discuss the issue of how the runtime system maintains memory in great detail. Basically what you need to know is that there are two principal components of dynamic memory allocation. The first is the *stack.* When procedures are called, arguments and local variables are pushed onto the stack, and when a procedure returns these variables are popped. The stacks grows and shrinks in a very predictable way.

In contrast, whenever objects are allocated through the Java "`new`" operator, they are stored in a different section of memory called the *heap.* (In spite of the similarity of names, this heap has nothing to do with the binary heap data structure, which is used for priority queues.) As elements are allocated and deallocated, the heap storage becomes fragmented into pieces. How to maintain the heap efficiently is the main issue that we will consider.

Approaches: There are two basic aproaches of doing memory management. This has to do with whether storage deallocation is done *explicitly* or *implicitly.* Explicit deallocation is used in languages like C (resp., C++). Memory is allocated through `malloc` (resp., `new`), and is explicitly released to the system by invoking `free` (resp., `delete`). While these systems provide the user a high degree of control, they also impose a strong burden on the programmer. Blocks of memory may be *leaked* in the sense that the memory is inaccessible, and has not been deallocated. A more significant programming bug is the result of *aliasing*, where (unknown to the programmer) two pointers reference the same block of memory. (This often happens when a *shallow copy* results in a pointer being copied, rather than making a copy of the underlying object.) Now, when one of these pointers is used to release the block, the other pointer still references this chunk of released memory.

In contrast languages like Java uses implicit deallocation. It is up to the system to determine which objects are no longer accessible and reclaim their storage. This process is called *garbage collection.* In both cases there are a number of choices that can be made, and these choices can have significant impacts on the performance of the memory allocation system. Unfortunately, there is not one system that is best for all circumstances. We begin by discussing explicit deallocation systems.

Explicit Allocation/Deallocation: There is one case in which explicit deallocation is very easy to handle. This is when all the objects being allocated are of the same size. A large contiguous portion of memory is used for the heap, and we partition this into *blocks* of size b, where b is the size of each object. For each unallocated block, we use one word of the block to act as a *next* pointer, and simply link these blocks together in linked list, called the *available space list.* For each `alloc` request, we extract a block from the available space list and for each `dealloc` we return the block to the list.

If the records are of different sizes then things become much trickier. We will partition memory into blocks of varying sizes. Each block (allocated or available) contains information

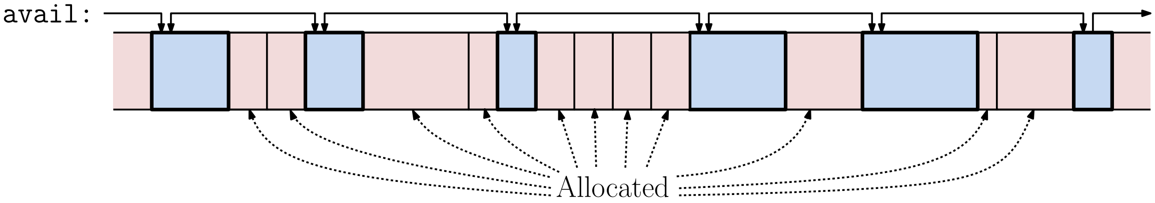

Fig. 1: Dynamic memory allocation block structure.

indicating how large it is. Available blocks are linked together to form an available space list. The main questions are: (1) what is the best way to allocate blocks for each `alloc` request, and (2) what is the fastest way to deallocate blocks for each `dealloc` request.

The biggest problem in such systems is the fact that after a series of allocation and deallocation requests the memory space tends to become *fragmented* into small blocks of memory. This is called *external fragmentation*, and is inherent to all dynamic memory allocators. Fragmentation increases the memory allocation system's running time by increasing the size of the available space list, and when a request comes for a large block, it may not be possible to satisfy this request, even though there is enough total memory available in these small blocks. Observe that it is not usually feasible to compact memory by moving fragments around. This is because there may be pointers stored in local variables that point into the heap. Moving blocks around would require finding these pointers and updating them, which is a very expensive proposition. We will consider it later in the context of garbage collection. A good memory manager is one that does a good job of controlling external fragmentation.

Overview: When allocating a block we must search through the list of available blocks of memory for one that is large enough to satisfy the request. The first question is, assuming that there does exist a block that is large enough to satisfy the request, what is the best block to select? There are two common but conflicting strategies:

First-fit: Search the available blocks sequentially until finding the first one that is large enough to satisfy the request.

Best-fit: Search all available blocks and select the smallest block that is large enough to fulfill the request.

Both methods work well in some instances and poorly in others. Remarkably, in spite of its name, Best-fit often performs worse in practice than First-fit. The reasons are twofold. First, First-fit is faster to execute, since the search can stop at the first block of sufficiently large size, rather than having to search the entire available list. Second, best-fit tends to produce a good deal of fragmentation by selecting blocks that are just barely larger than the requested size, but this results in a small residual block, or *sliver*, left over, and there may not be any future requests that are small enough that this small block can be used.

One method which is commonly used to reduce fragmentation is when a request is just barely filled by an available block that is slightly larger than the request, we allocate the entire block (more than the request) to avoid the creation of the sliver. This keeps the list of available blocks free of large numbers of tiny fragments which increase the search time. The additional waste of space that results because we allocate a larger block of memory than the user requested is called *internal fragmentation* (since the waste is inside the allocated block).

When deallocating a block, it is important that if there is available storage we should merge the newly deallocated block with any neighboring available blocks to create large blocks of free space. This process is called *merging*. Merging is trickier than one might first imagine. For example, we want to know whether the preceding or following block is available. How would we do this? We could walk along the available space list and see whether we find it, but this would be very slow. We might store a special bit pattern at the end and start of each block to indicate whether it is available or not, but what if the block is allocated that the data contents happen to match this bit pattern by accident? Let us consider the implementation of this scheme in greater detail.

Notation and Assumptions: Memory is usually allocated in bytes, but it is common to align all allocated blocks to a *word* (typically 32-bits) or a *double-word* (typically 64 bits) boundary. This way, we don't need to know what the data is being used for (bytes, ints, floats, doubles, etc.). In our examples, we will make the simplifying assumption that requests for memory allocation are given in terms of a number of words, and the block of memory that we return will be aligned at a word boundary. Given a pointer `p` to memory, we will use `*p` to refer to the contents of the word of memory at this address. Given a pointer `p` and integer i, we will use `p + i` to refer to the memory location that is i words beyond `p`. We will use the expression `void*` to indicate the "type" of a pointer to a location of physical memory.

Block Structure: The main issues are how to we store the available blocks so we can search them quickly (skipping over used blocks), and how do we determine whether we can merge with neighboring blocks or not. We want the operations to be efficient, but we do not want to use up excessive pointer space (especially in used blocks). Here is a sketch of a solution (one of many possible).

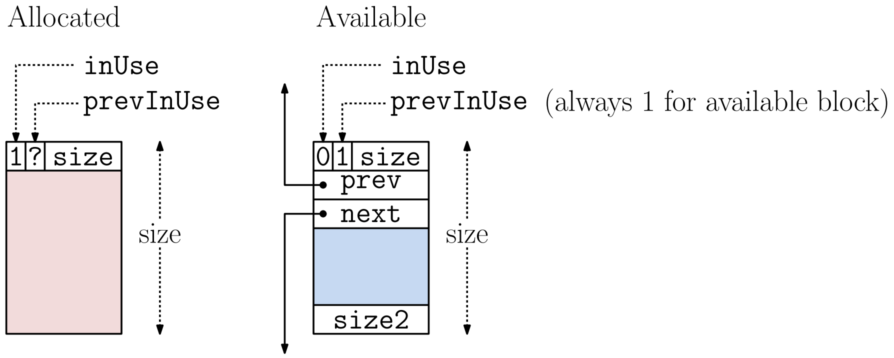

Fig. 2: Block structure for dynamic storage allocation.

Allocated blocks: For each block of used memory we record the following information. It can all fit in the first word of the block.

`size`: The size of the block of storage (including space for the following fields).

`inUse`: A single bit that is set to 1 (true) to indicate that this block is in use.

`prevInUse`: A single bit that is set to 1 (true) if the previous block is in use and 0 (false) otherwise. (Later we will see why this is useful.)

Available blocks: For an available block we store more information, which is okay because the user is not using this space. These blocks are stored in a doubly-linked circular list, called `avail`. Note that the available space list need not sorted by physical memory addresses. For example, it might be ordered by some other criteria, such as the sizes of the blocks.

`size`: The size of the block of storage (including space for the following fields).

`inUse`: A bit that is set to 0 (false) to indicate that this block is not in use.

`prevInUse`: A bit that is set to 1 (true) if the immediately preceding block (not the same as `prev`) is in use (which should always be true, since we should never have two consecutive unused blocks).

`prev`: A pointer to the previous block on the available space list. (It need not be the block immediately preceding in memory.)

`next`: A pointer to the next block on the available space list. (It need not be the block immediately following in memory.)

`size2`: Contains the same value as `size` at the head of the block. This field is stored in the *last* word of the block. For block p it can be accessed as `*(p + p.size - 1)`.

Note that available blocks require more space for all this extra information. The system will never allow the creation of a fragment that is so small that it cannot contain all this information.

You will observe that the run-time system trusts that the user's program will not overwrite any of the block header fields or previous/next pointers. What is to keep this from happening? The answer (at least with C and C++) is *nothing*! If the program accidentally overwrites a memory location outside of the allocated block, it can destroy the memory system's integrity, and very soon everything fails. Usually, the result is an abort with just a cryptic message, such as "Segmentation fault." Failure to check for these faults can result in undetected *buffer-overflow* writes, a common technique for hacking into software systems.

Allocation: To allocate a block we search through the linked list of available blocks until finding one of sufficient size (see Fig. 3). If the request is about the same size (or perhaps slightly smaller) as the block, we remove the block from the list of available blocks (performing the necessary relinkings) and return a pointer to it. We also may need to update the `prevInUse` bit of the next block since this block is no longer available. Otherwise we split the block into two smaller blocks, return one to the user, and leave the other on the available space list.

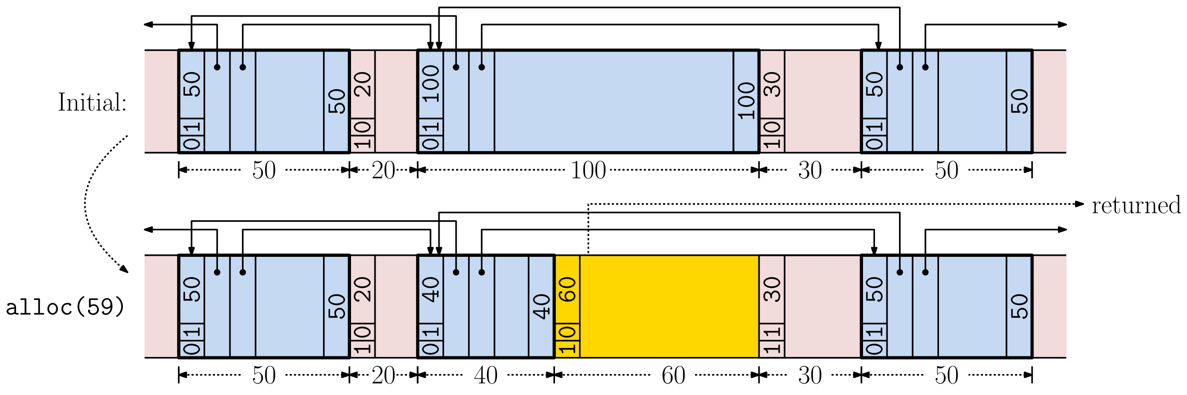

Fig. 3: An example of block allocation.

The following code block presents an algorithm for allocating a new block of memory. Note that we make use of pointer arithmetic here. The argument `b` is the desired size of the allocation. Because we reserve one word of storage for our own use we increment this value on entry to the procedure. We keep a constant TOO_SMALL, which indicates the smallest allowable fragment size. If the allocation would result in a fragment of size less than this value, we return the entire block. The procedure returns a generic pointer to the newly allocated block. The utility function `avail.unlink(p)` simply unlinks block p from the doubly-linked available space list. An example is provided in Fig. 3. Shaded blocks are available.

Allocate a block of storage

```
(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)
}
```

Deallocation: To deallocate a block, we check whether the next block or the preceding blocks are available. For the next block we can find its first word and check its `inUse` field. For the preceding block we use our own `prevInUse` field. (This is why this field is present). If the previous block is not in use, then we use the size value stored in the last word to find the block's header. If either of these blocks is available, we merge the two blocks and update the header values appropriately. If both the preceding and next blocks are available, then this result in one of these blocks being deleting from the available space list (since we do not want to have two consecutive available blocks). If both the preceding and next blocks are in-use, we simply link this block into the list of available blocks (e.g. at the head of the list).

The deletion function is shown in the following code block. Let `avail` denote the head of the available space list. There are four different cases depending on whether the blocks that immediate precede or follow p are allocated or available.

- If the following block q is available, then we merge it with p. (See Fig. 4, upper half.) To do this we update p's size, and move q's record in the available space list to p, using a utility function `move(q, p)`. This copies q's previous and next fields to p and appropriately update the entries in the available space list that point to q. (E.g., `q.prev.next = p`.) Otherwise we add p to the available space list, which we assume will set its previous and next pointers. Now we may assume that p is on the available space list.
- If the preceding block is not in use, we merge this block with p (see Fig. 4, lower half), and unlink p from the available space list. Note that we do need to alter `p.prevInUse`.

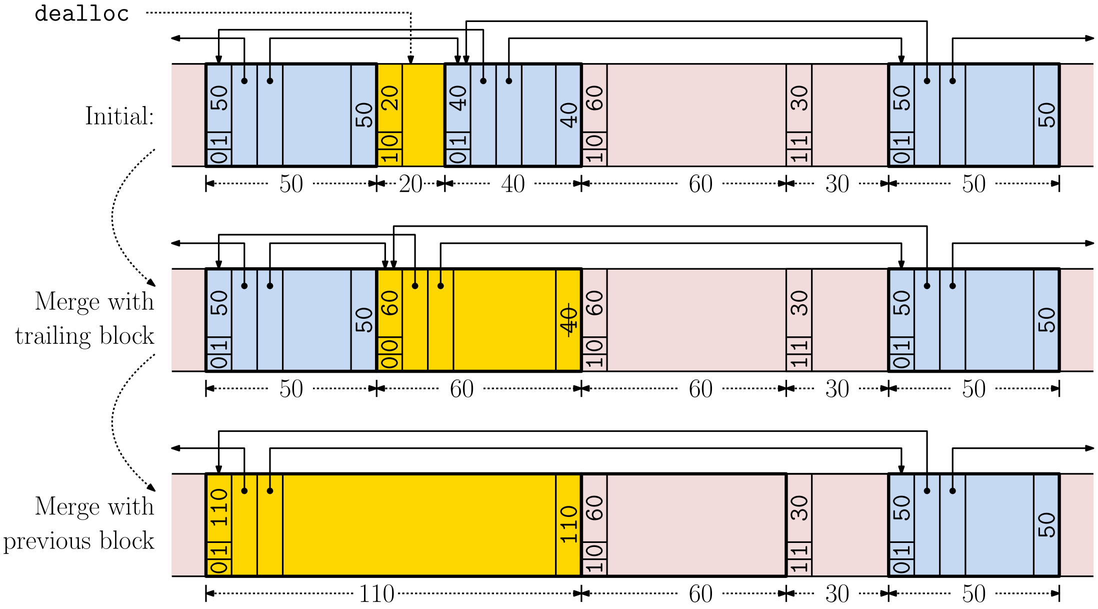

Fig. 4: An example of block deallocation and merging.

If the previous block was available, then we merged p with it and p has gone away, and otherwise p's original value was correct.

Analysis: There is not much theoretical analysis of this method of dynamic storage allocation. Because the system has no knowledge of the future sequence of allocation and deallocation requests, it is possible to contrive situations in which either first fit or best fit (or virtually any other method you can imagine) will perform poorly. Empirical studies based on simulations have shown that this method achieves utilizations of around 2/3 of the total available storage before failing to satisfy a request. Even higher utilizations can be achieved if the blocks are small on average and block sizes are similar (since this limits fragmentation). A rule of thumb is to allocate a heap that is at least 10 times larger than the largest block to be allocated.

Buddy System: The dynamic storage allocation method described last time suffers from the problem that long sequences of allocations and deallocations of objects of various sizes tends to result in a highly fragmented space. The *buddy system* is an alternative allocation system which limits the possible sizes of blocks and their positions, and so tends to produce a more well-structured allocation. Because it limits block sizes, *internal fragmentation* (the waste caused when an allocation request is mapped to a larger block size) becomes an issue.

The buddy system works by starting with a block of memory whose size is a power of 2 and then hierarchically subdivides each block into blocks of equal sizes (see Fig. 6). To make this intuition more formal, we introduce the two key elements of the buddy system:

(i) The sizes of all blocks (both allocated and available) are powers of 2. When a request comes for an allocation, the request (including the overhead space needed for storing block size information) is artificially rounded up to the next higher power of 2. Note that the allocated size is never more than twice the size of the request.

(ii) Blocks of size 2^k start at memory addresses that are multiples of 2^k. (We assume that addressing starts at 0, but it is easy to update this scheme to start at any arbitrary

Deallocating a block of storage

```
void dealloc(void* p) {                     // deallocate block at p
    p--;                                    // back up to the header
    q = p + p.size;                         // the following block
    if (!q.inUse) {                         // is following 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 is now in 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
     }
}
```

address by simply shifting addresses by an appropriate offset.)

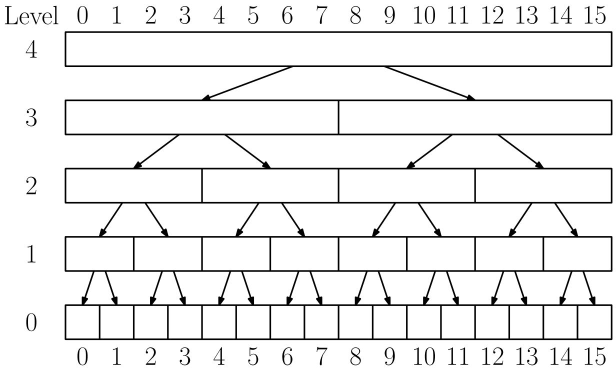

Fig. 5: Buddy system block structure.

Note that the above requirements limits the ways in which blocks may be merged. For example Fig. 6 below illustrates a buddy system allocation of blocks, where the blocks of size 2^k are shown at the same level. Available blocks shaded blue and allocated blocks are shaded red. The two available blocks at addresses 5 and 6 (the two white blocks between 4 and 8) cannot be merged because the result would be a block of length 2, starting at address 5, which is not a multiple of 2. For each size group, there is a separate available space list.

For every block there is exactly one other block with which this block can be merged with. This is called its *buddy*. In general, if a block b is of size 2^k, and is located at address x, then its buddy is the block of size 2^k located at address

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

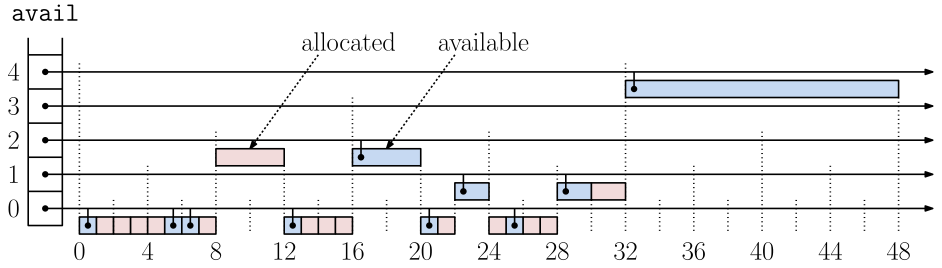

Fig. 6: Buddy system example. Allocated blocks are shaded red and available blocks are shaded blue. Available blocks are linked into the available-block list of the appropriate size.

This is easy to compute via bit manipulation. Basically the buddy's address is formed by complementing bit k in the binary representation of x (where the lowest order bit is bit 0). In Java, this can be expressed as `buddy(k,x) = (1<<k)^x`. For example, for $k = 3$ the blocks of length 8 at addresses 208 and 216 are buddies. If we look at their binary representations we see that they differ in bit position 3 (four bits from the right). Because they must be multiples of 8, bits 0–2 are zero.

$$\begin{aligned} 80 &= \texttt{1010000}_2 \\ 88 &= \texttt{1011000}_2. \end{aligned}$$

So, $\text{buddy}_3(80) = 88$ and vice versa.

Putting the Pieces Together: As we mentioned earlier, one principle advantage of the buddy system is that we can exploit the regular sizes of blocks to search efficiently for available blocks. We maintain an array of linked lists, one for the available block list for each size group. In particular, `avail[k]` is the header pointer to a doubly linked list of available blocks of size 2^k.

Here is how the basic operations work. We assume that each block has the same structure as described in the dynamic storage allocation example from last time. The `prevInUse` bit and the size field at the end of each available block are not needed given the extra structure provided in the buddy system. Each block stores its size (actually the log base 2 of its size is sufficient) and a bit indicating whether it is allocated or not. Also each available block has links to the previous and next entries on the available space list. There is not just one available space, but rather there are multiple lists, one for each level of the hierarchy (something like a skip list). This makes it possible to search quickly for a block of a given size.

Buddy System Allocation: We will give a high level description of allocation and dealloction. To allocate a block of size b, let $k = \lceil \lg(b+1) \rceil$. (Recall that we need to include one extra word for the block size. However, to simplify our figures we will ignore this extra word.) We will allocate a block of size 2^k. In general there may not be a block of exactly this size available, so find the smallest $j \geq k$ such that there is an available block of size 2^j. If $j > k$, repeatedly split this block until we create a block of size 2^k. In the process we will create one or more new blocks, which are added to the appropriate available space lists.

For example, in Fig. 7 we request a block of length 2. There are no available blocks of this size, so we use the smallest available block of the next larger size, that is, the block of length 16 at address 32. We remove it from its available space list. We recursively split it into subblocks of sizes 8, 4, 2, 2, until we get to a block of the desired size. We return one of the blocks of size 2, and we insert the remaining blocks (of sizes 2, 4, and 8) into the available space lists of the appropriate sizes.

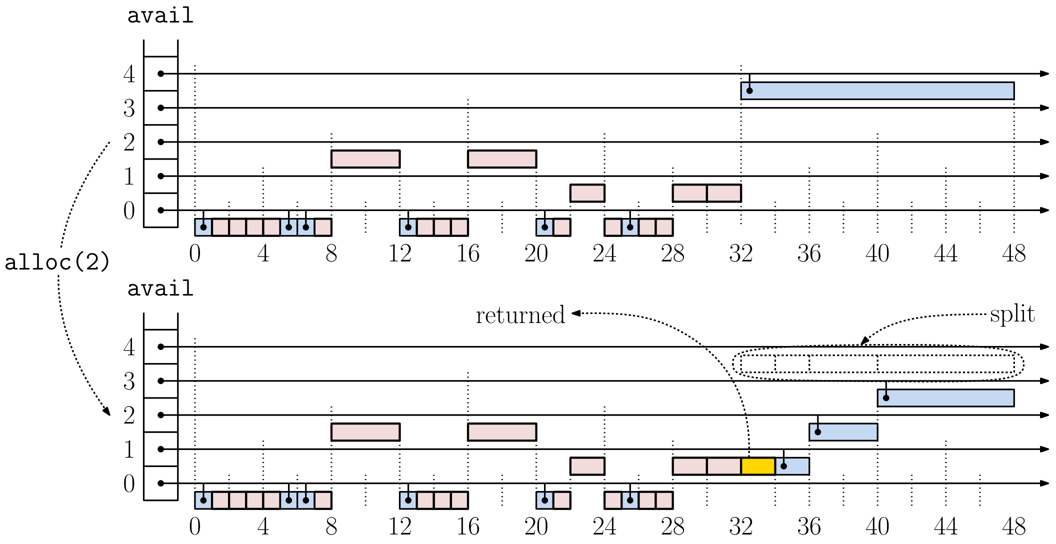

Fig. 7: Example of allocating a block in the buddy system.

Deallocation: To deallocate a block, we first mark this block as being available. We then check to see whether its buddy is available. This can be done in constant time. If so, we remove the buddy from its available space list, and merge them together into a single free block of twice the size. This process is repeated until we find that the buddy is allocated.

Fig. 8 shows the deallocation of the block of size 1 at address 21. It is merged with its buddy of size 1 at address 20, thus forming a block of size 2 at 20. This is then merged with its size-2 buddy at 22, forming a block of size 4 at 20. Finally, this is merged with its size-4 buddy at 16, forming a block of size 8 at 16. This block's buddy (the block of size 8 starting at 0) is not available, so the merging process stops. We insert this final block into the appropriate level of the available space list.

Fibonacci Buddy System: An interesting variant of the buddy system is designed to reduce fragmentation by using a hierarchy that is "denser" with respect to the sizes available. The *Fibonacci Buddy System* uses Fibonacci numbers, rather than powers of 2.

Recall that $F(0) = 0$, $F(1) = 1$, and generally $F(i) = F(i-1) + F(i-2)$. The first few Fibonacci numbers are: $0, 1, 1, 2, 3, 5, 8, 13, 21, 34, \ldots$. In this system `avail[k]` stores available blocks of size $F(k)$. Each allocation request (after adding +1 for the header) is rounded up to the next larger Fibonacci number, say $F(k)$. If no block of size $F(k)$ is available, we find next larger available size, say $F(j)$. We then repeatedly split $F(j)$ using Fibonacci numbers to obtain a block of the desired size. For example, if $F(k) = F(3) = 2$ is the desired size and $F(j) = F(9) = 34$ is the smallest available, we split the $F(j)$ block up as:

$$F(9) = 34 = F(7)+F(8) = F(5)+F(6)+F(8) = F(3)+F(4)+F(6)+F(8) = 2+3+8+21.$$

We remove the block of size 34 from `avail[9]`, and split it up as described above. We return the block of size 2, and insert the blocks of sizes 3, 8, and 21 into `avail[4]`, `avail[6]`, `avail[8]`, respectively.

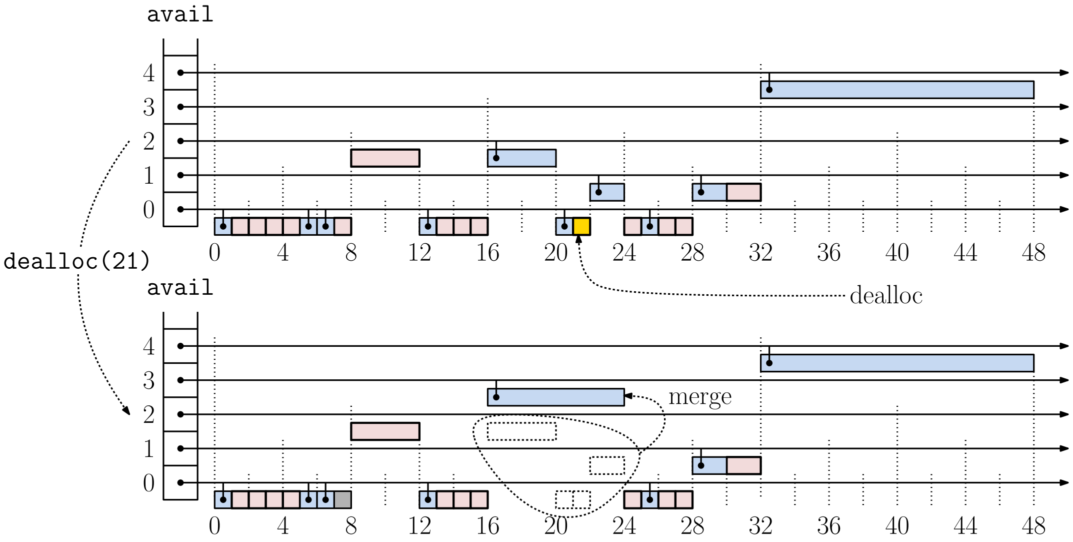

Fig. 8: Example of deallocating a block in the buddy system.