CS 33

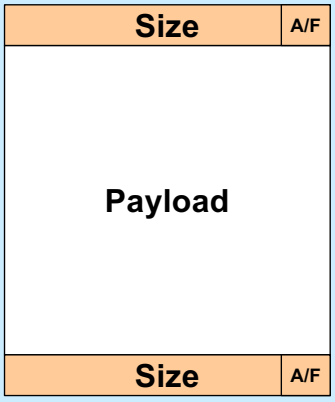
Storage Allocation

Data Structure Requirements

- All blocks
  - we need to know how big they are
    - » when free is called, it must be known how much to free
    - » when looking at a free block in malloc, we need to know its size
  - we need to know which they are: free or allocated
    - » needed for coalescing
- Free blocks
  - they need to be linked into the free list

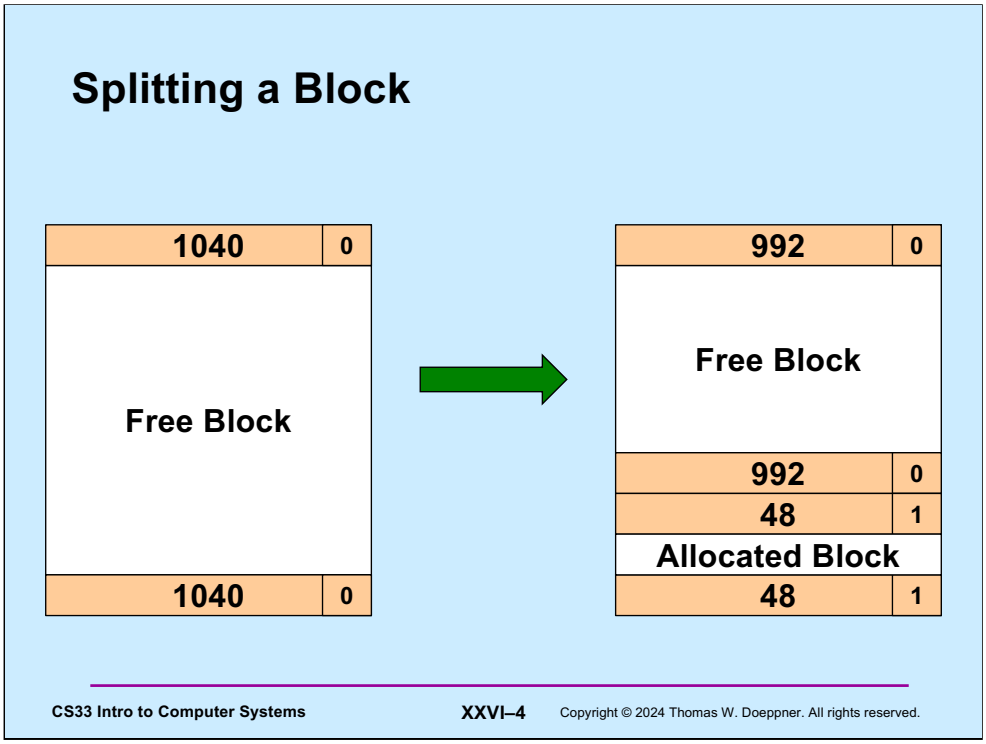
It's now time to design the data structures we need to represent our "heap" – the dynamic memory region.

Solution: Boundary Tags



One solution (which we use in the malloc assignment) is the boundary tags approach. Here we have a fixed overhead for each block of memory (whether free or allocated) that indicates its size and whether it's free. So that we can determine if adjacent blocks are free (and what their sizes are), we put this information at each end of the block. The non-overhead portion of the block (which is available to hold data) is called the **payload**.

One could set the **size** to be the size of the entire block, or the size of just the payload – either way can be made to work. We find it more convenient for the **size** to be that of the entire block. Thus the size of the payload is **size** minus the amount of memory required to hold the boundary tags (in our implementation, each boundary tag (containing size and the allocated-or-free bit) is a **long**; thus the total amount of memory used for the boundary tags is 16 bytes).

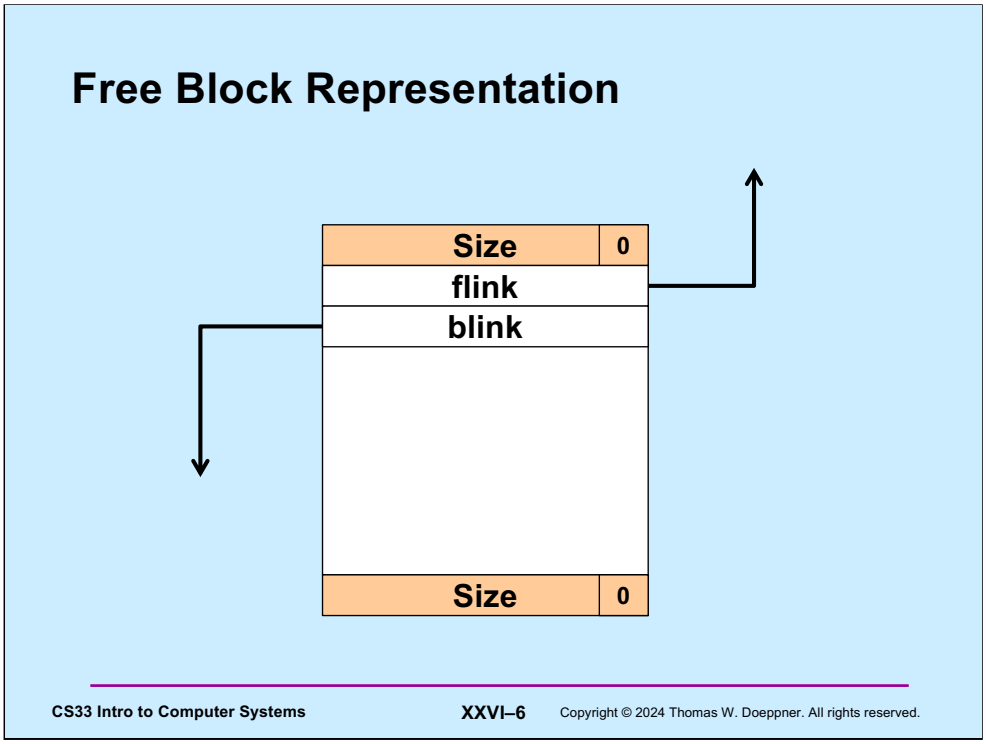


Splitting a block is straightforward. We take a block that was previously free and divide it into two blocks – an allocated block that's big enough to hold the storage request, and the remainder represented as a free block.

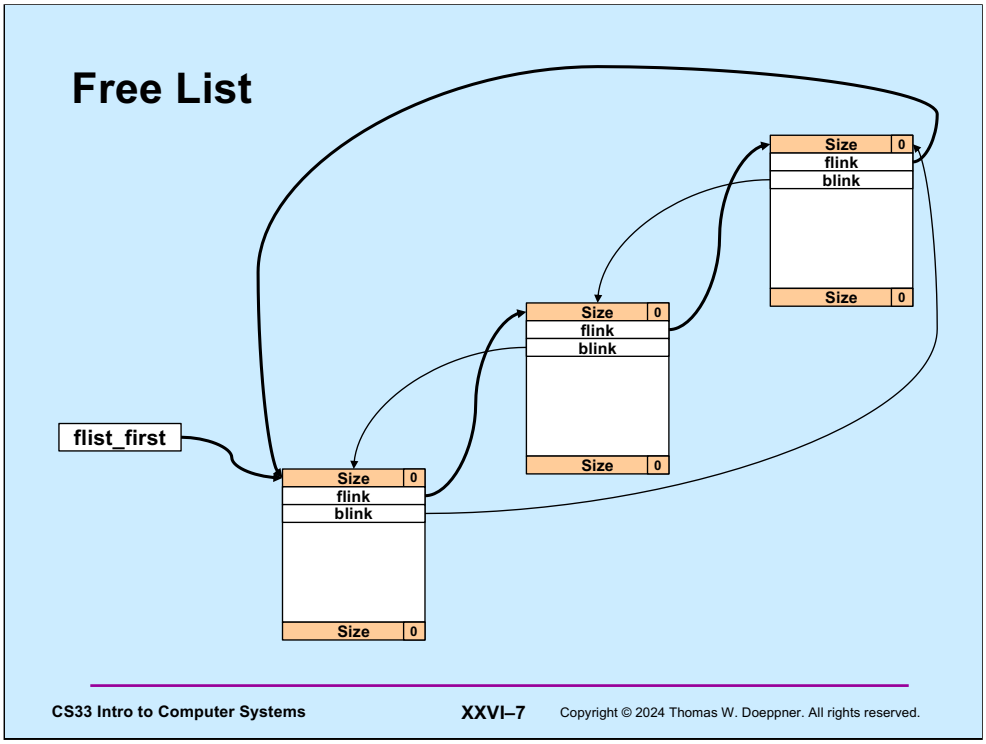
Representing the Free List

- We need a pointer to the first element
  - *flist_first*
- We need to traverse the list from beginning to end
  - required by malloc
- We need to merge adjacent blocks
  - this may require removing a block from the free list, then reinserting it (as part of a coalesced block)
- Links may be put in the free block's payload area
  - not needed for allocated blocks!

The global variable **flist_first** is a pointer to the first item in the free list (and is null if the free list is empty).



Here's our representation of a free block. Note that it has both a forward link (**flink**) and a backwards link (**blink**) – thus the free list is doubly linked.



The free list is a circular, doubly linked list.

Quiz 1

Why is the free list doubly linked?

a) we don't really need it to be doubly linked for malloc and free, but it may be necessary for some future operations
b) so that, given a pointer to an arbitrary free block, we can easily remove the block from the list
c) to facilitate sorting the free list
d) so we can traverse it in both directions

If the course had a final exam, this question would definitely be on it. Make sure you understand the answer. It will come up again in the course (and count towards your grade!).

Quiz 2

Why is the free list circular?

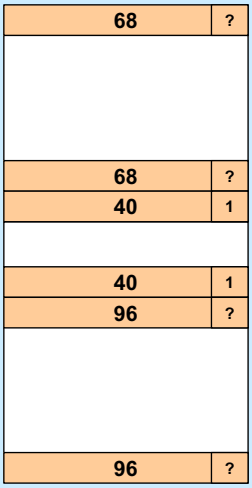
a) so that we don't have to special-case the the handling of the first and last list elements
b) to facilitate implementing the next-fit search strategy
c) both of the above
d) none of the above

Heap ≠ Free List

- Heap
  - collection of all memory usable as dynamic storage: the dynamic portion of the address space
    - » both allocated and free
- Free list
  - those blocks of the heap that are free
    - » linked together (circular, doubly)

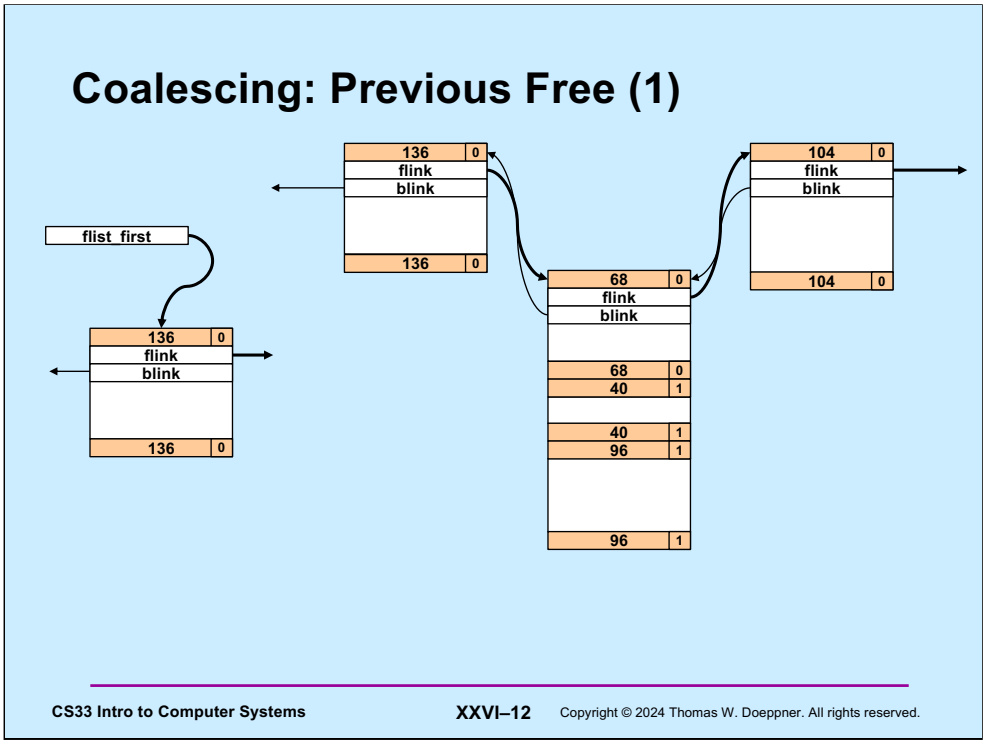
- Both important, but different
- Confusion: what does *next block* mean?
  - next adjacent block (next in heap)
  - next free block (next in free list)

Coalescing Revisited

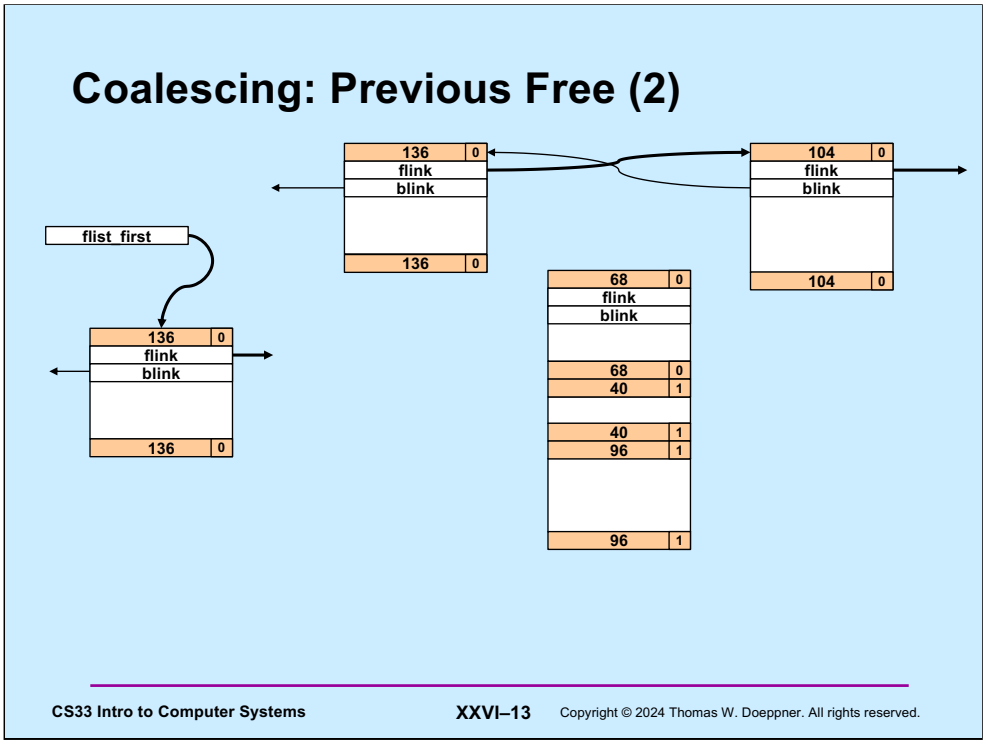


- We are freeing a block
  - is the previous block free?
  - is the next block free?
  - are both free?

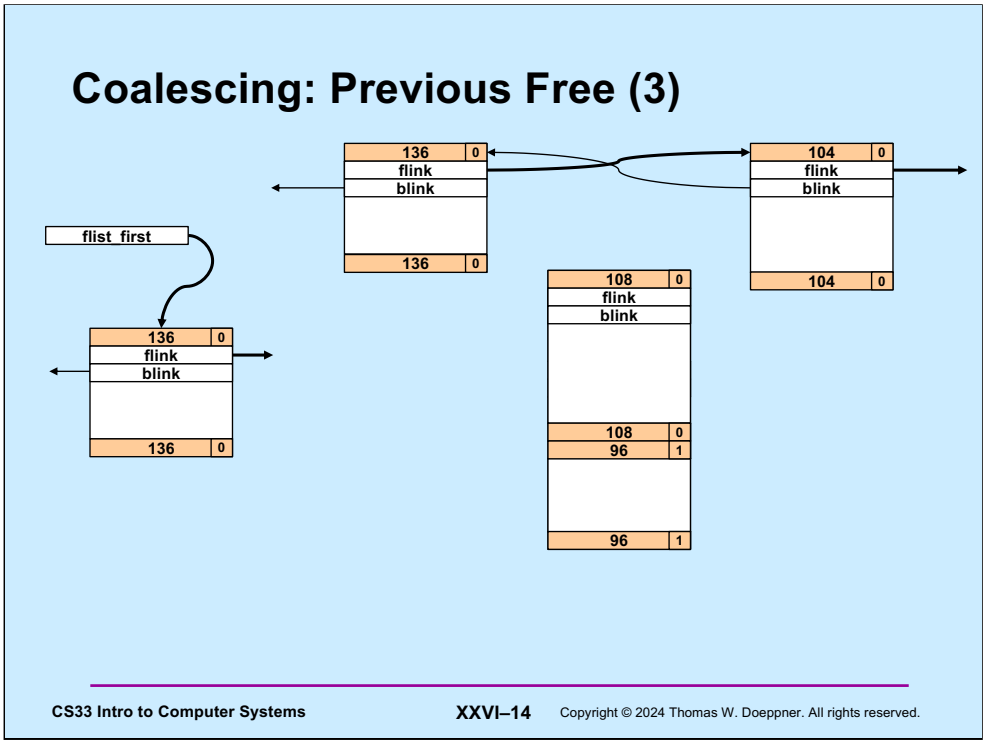
We now look at implementing the coalesce operation, given our data structures. Let's assume that we're about to free the middle block, of size 40. To handle coalescing, we need to know whether the previous block and the next block are free.



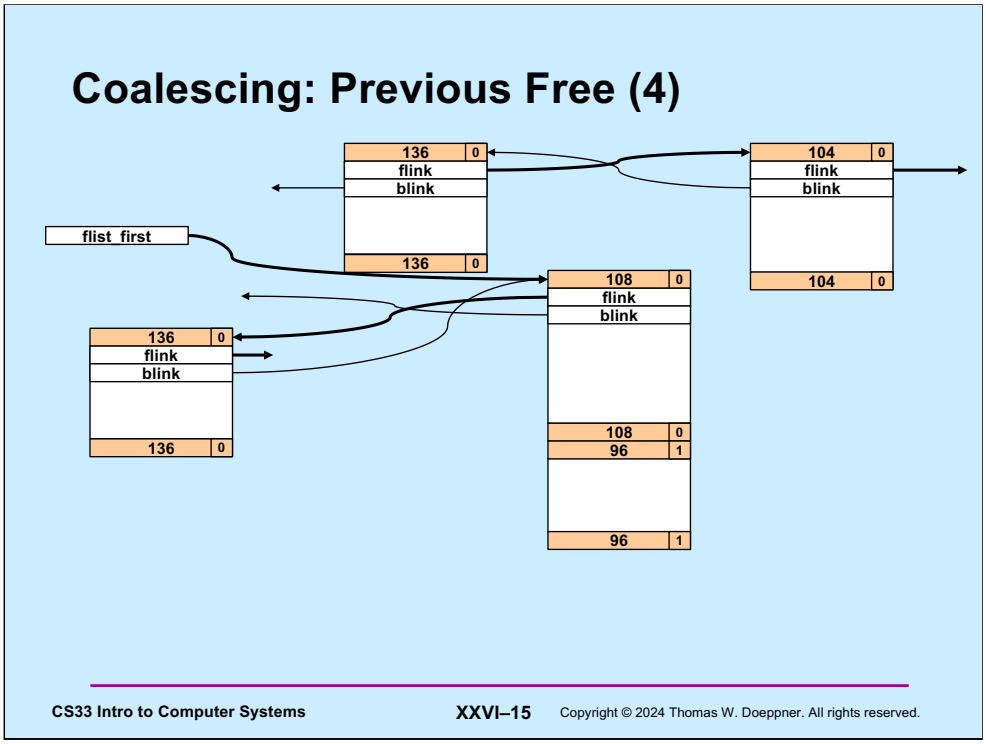
Suppose the previous block is free, but the next block is allocated. Thus, the previous block is in the free list. We'll assume it's not the first element of the free list, which is pointed to by **flist_first**.



We first pull the previous block from the free list.



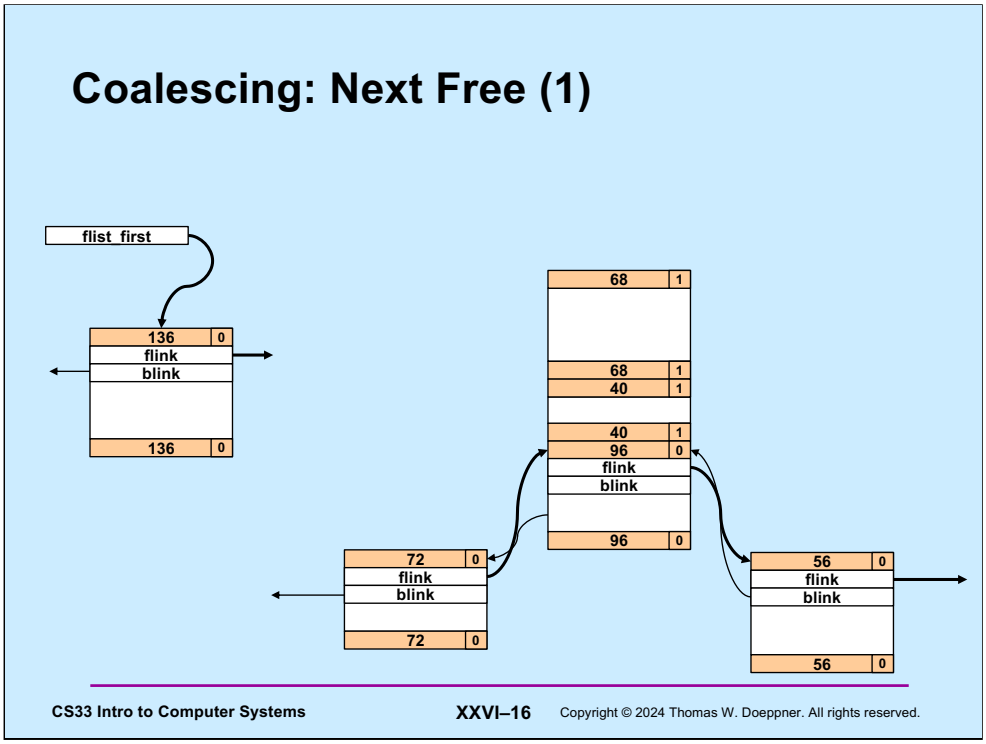
We the merge the newly freed block with the previous block.



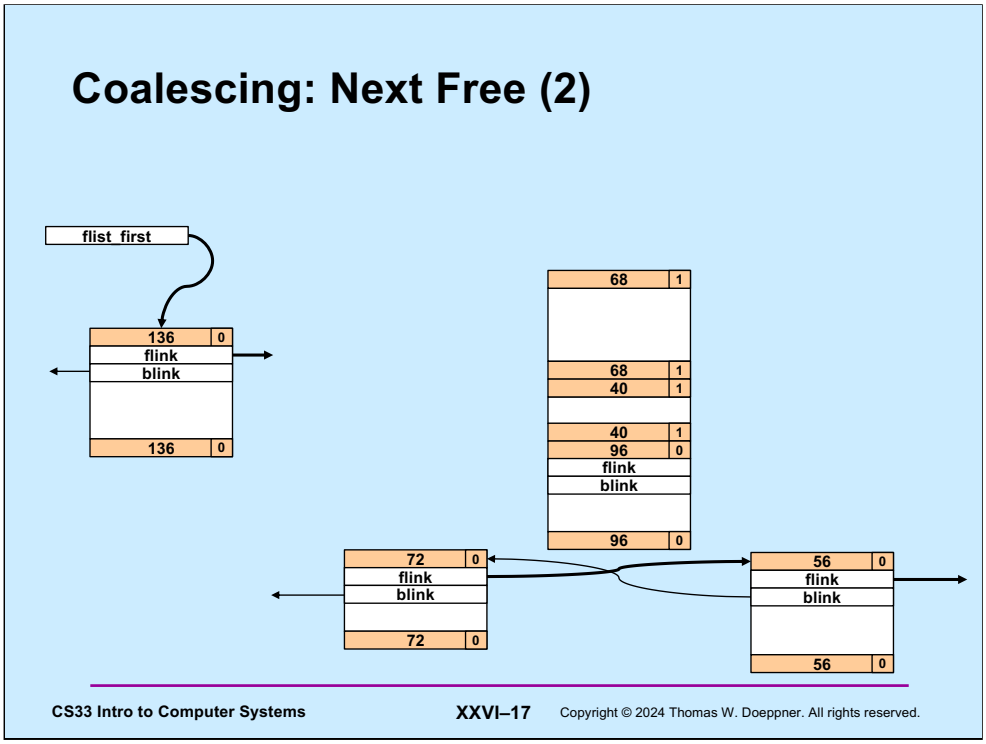
Finally, we add the merged free block to the beginning of the free list.

This, of course, is not the only way to do this. We could simply leave the previous block in the free list at its current position and increase its size so as to absorb the block being freed. This perhaps could be more efficient than what's shown in the slide, but it leads to some slight complications in the code. Feel free to do it either way in your own code.

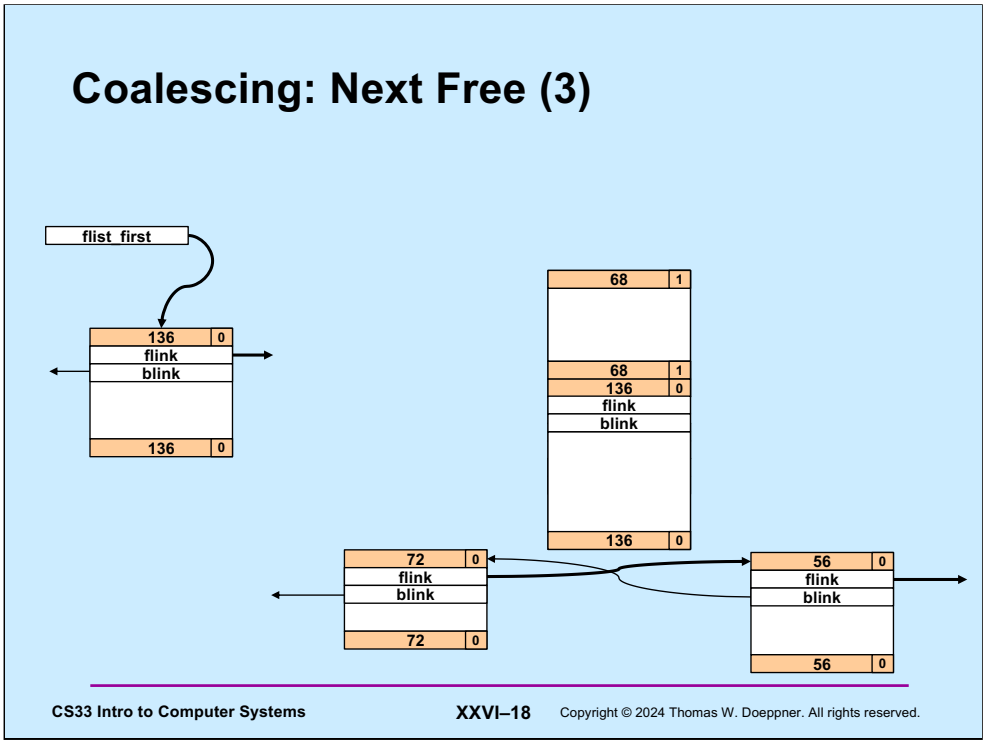
A potential advantage of implementing coalesce as done here is that it puts a potentially larger block at the beginning of the free list, possibly improving the performance of first fit.



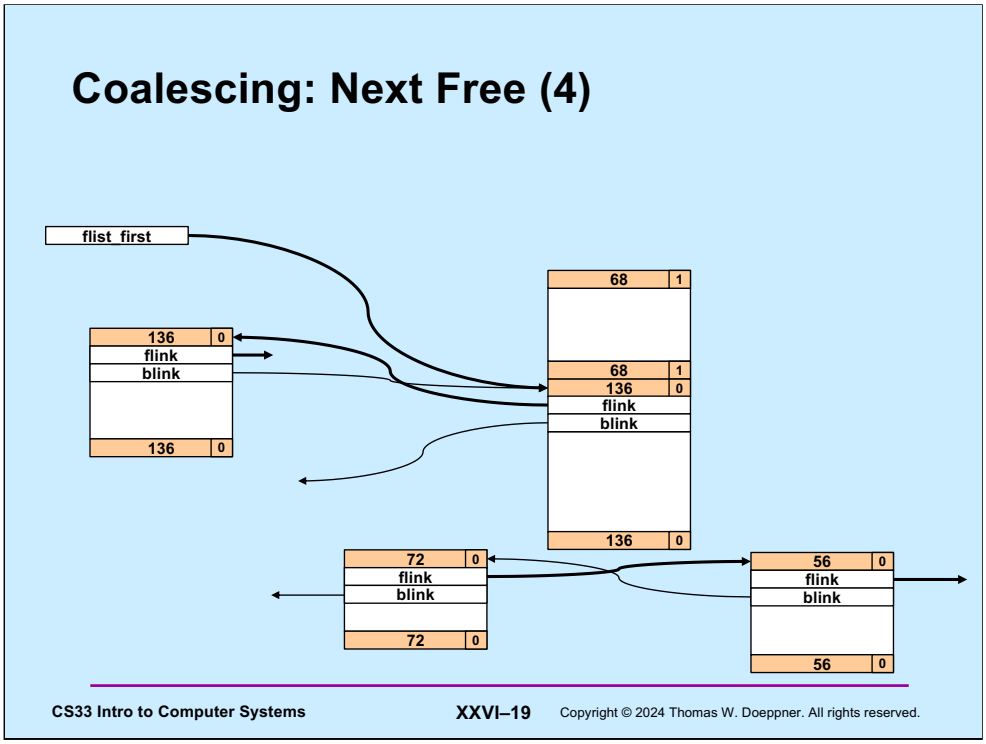
Here the previous block is allocated but the next block is free.



We first pull the next block from the free list.

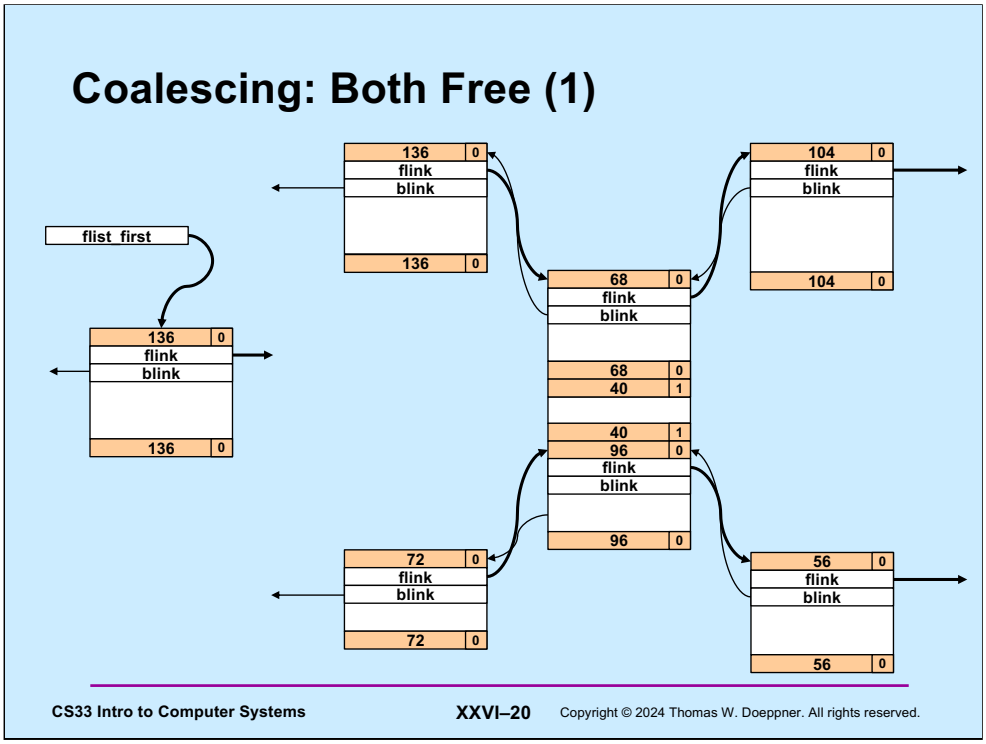


We then merge the block we're freeing with the next block.

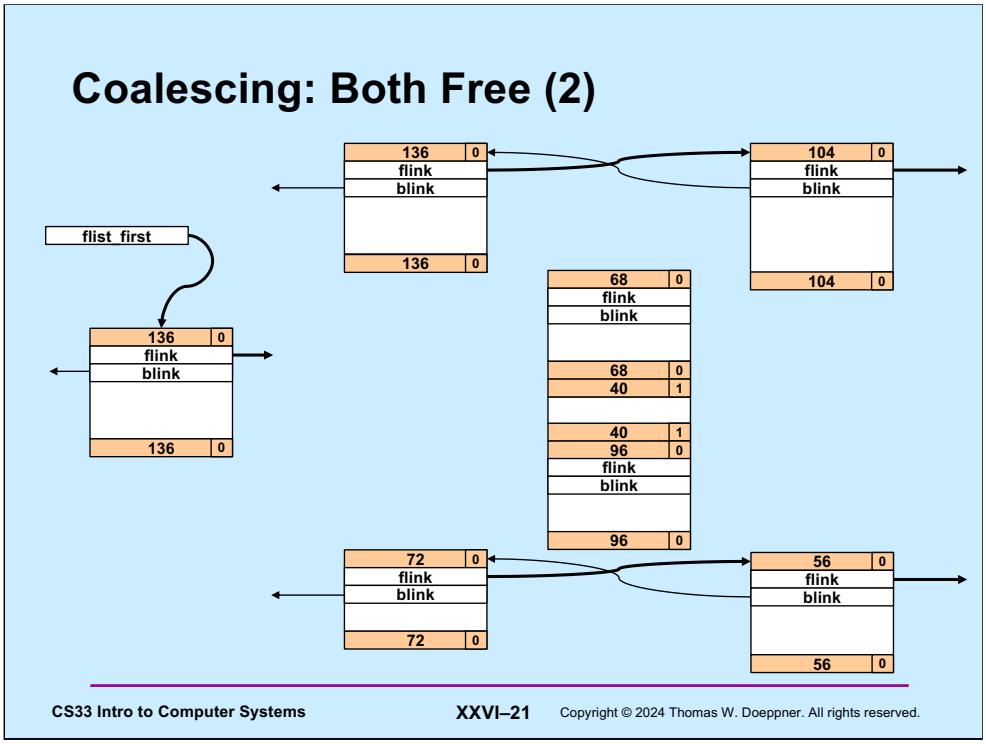


Finally, we insert the combined block into the beginning of the free list.

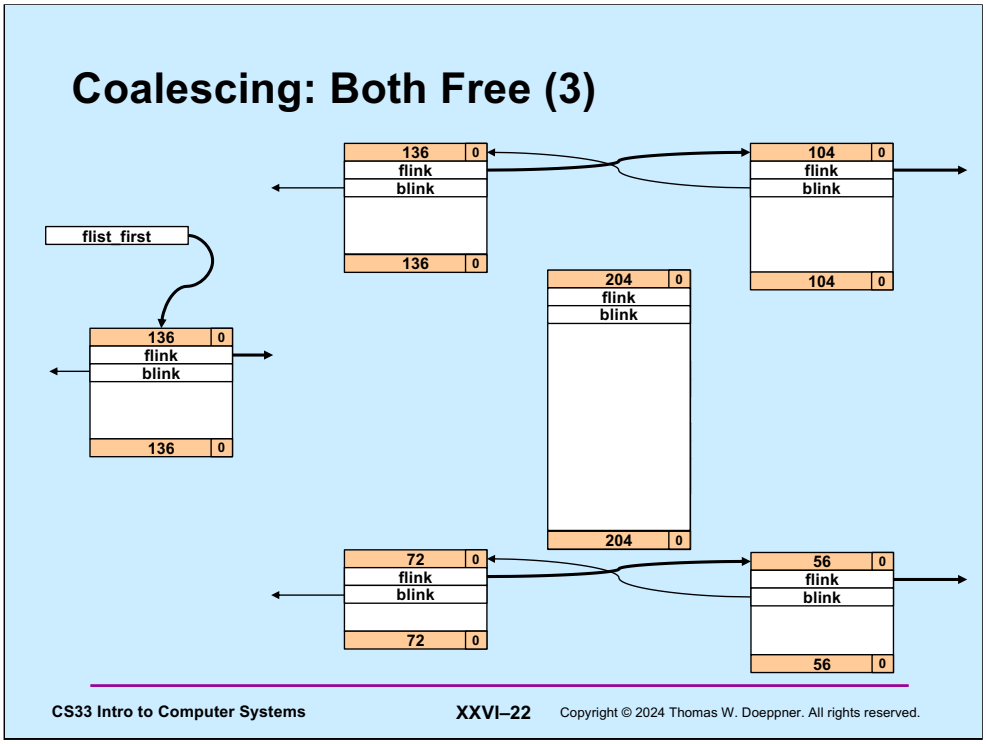
Again, there are other ways for doing this. In particular, one might simply replace the next block with the combined block, putting it into the free list where the next block was.



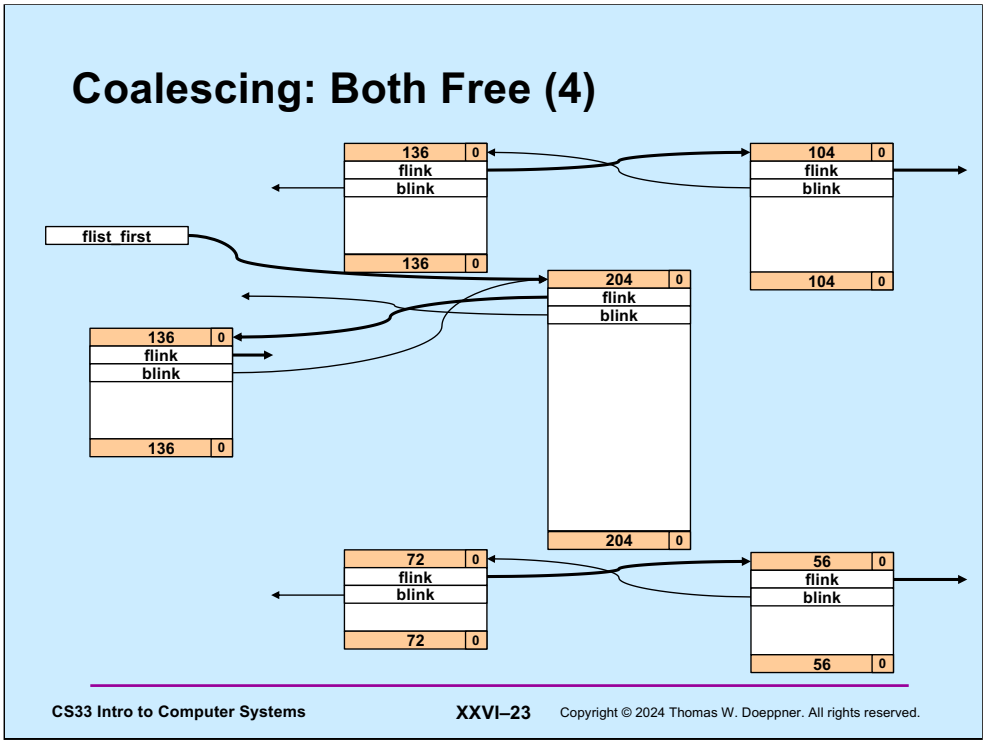
Finally, we have the case in which both the previous and the next blocks are free.



We remove both the prev and next blocks from the free list.

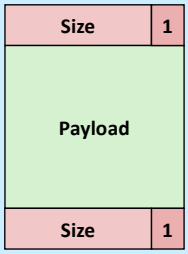


We merge the block we're freeing with the prev and next blocks.



Finally we insert the combined block into the beginning of the free list.

C vs. Storage Allocation




```
typedef struct block {
  long size;
  long payload[size/8 - 2];
  long end_size;
} block_t;
```

```
typedef struct free_block {
  long size;
  struct free_block *flink;
  struct free_block *blink;
  long filler[size/8 - 4];
  long end_size;
} free_block_t;
```

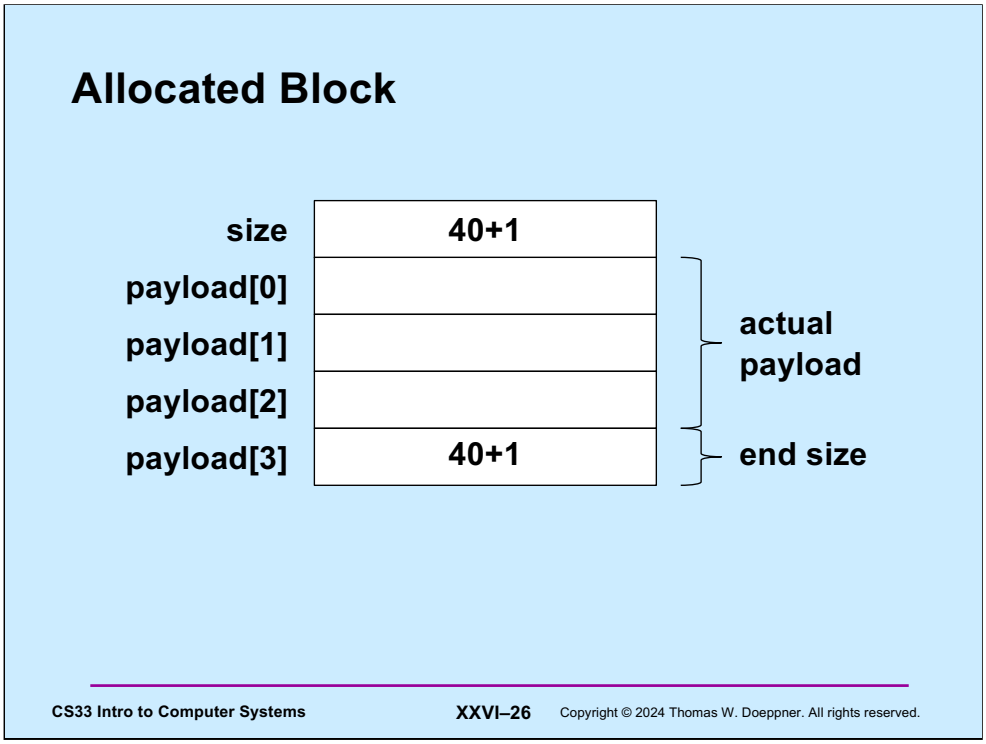

What we might like to be able to do in C is expressed on the slide. Unfortunately, C does not allow such variable-sized arrays. Another concern is the allocated flag, which we'd like to be included in the size fields.

Overcoming C

- **Think objects**
  - **a block is an object**
    - » **opaque to the outside world**
  - **define accessor functions to get and set its contents**

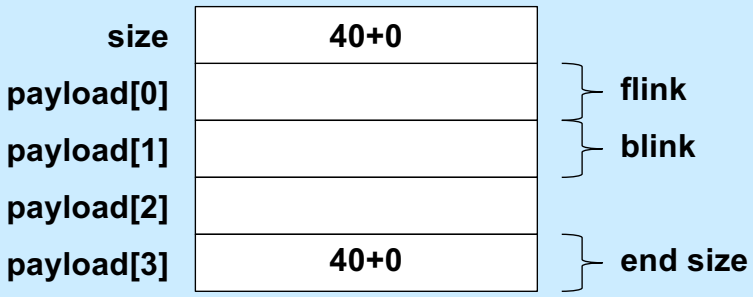
```
typedef struct block {
  size_t size;
  size_t payload[0];
} block_t;
```


Putting a zero for the dimension of payload is a way of saying that we do not know a priori how big payload will be, so we give it an (arbitrary) size of 0. Note that sizeof(size_t) is 8 (i.e., **size_t** is a typedef for a **long**).



In this example we have an allocated block of 40 bytes. Its size and end size fields have their low-order bits set to one to indicate that the block is allocated. (Since each element of payload is 8 bytes long, the entire allocated block, including tags, is 40 bytes long.)

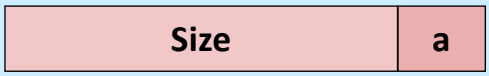
Free Block



- In general, end size is at *payload[size/8 – 2]*

For a free block, the size fields contain the exact size of the block: the allocated bits are zeroes. The first two elements of payload are the flink and blink pointers, respectively.

Overloading Size

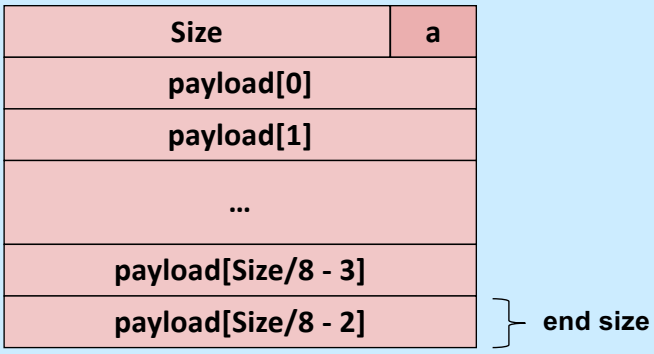



```
size_t block_allocated(block_t *b) {
  return b->size & 1;
}

size_t block_size(block_t *b) {
  return b->size & -2;
}
```


If we assume that the size of a block is always even (in practice it's probably a multiple of 4 or 8), then we can assume the least significant bit is zero and use that bit position to represent the allocated flag.

End Size




```
size_t *block_end_tag(block_t *b) {
  return &b->payload[b->size/8 - 2];
}
```


The block_end_tag function returns the address of a block's end tag, given the address of the beginning of the block (where its front tag is).

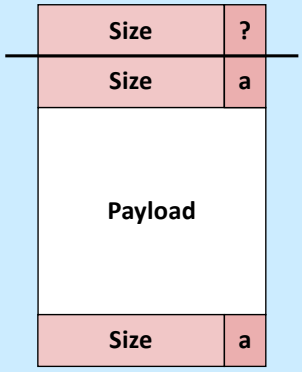
Setting the Size

```
void block_set_size(block_t *b, size_t size) {
  assert(!(size & 7));          // multiple of 8
  size |= block_allocated(b);  // preserve alloc bit
  b->size = size;
  *block_end_tag(b) = size;
}

void block_set_allocated(block_t *b, size_t a) {
  assert((a == 0) || (a == 1));
  if (a) {
    b->size |= 1;
    *block_end_tag(b) |= 1;
  } else {
    b->size &= -2;
    *block_end_tag(b) &= -2;
  }
}
```


Here we have functions for setting both tags of a block.

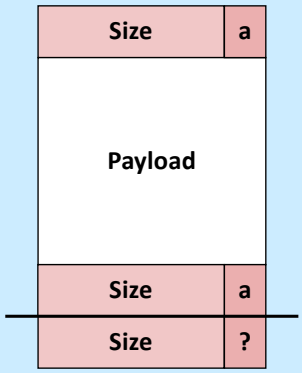
Is Previous Adjacent Block Free?




```
size_t block_prev_allocated(
    block_t *b) {
  return b->payload[-2] & 1;
}
```


We take advantage of the boundary-tags approach to determine if the previous block is free.

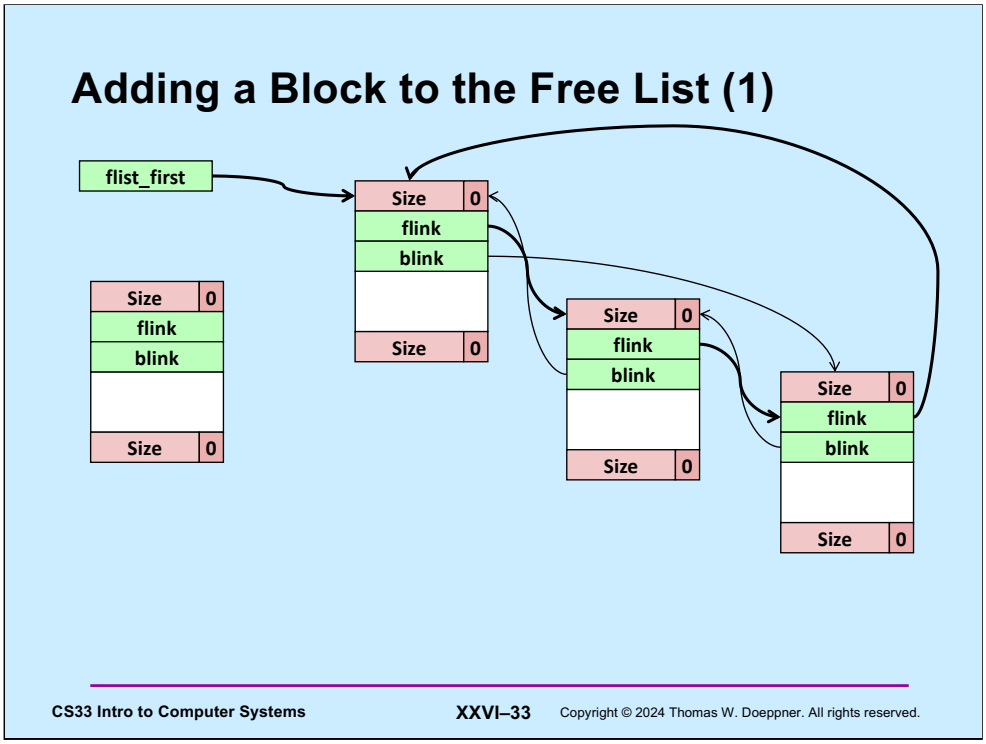
Is Next Adjacent Block Free?



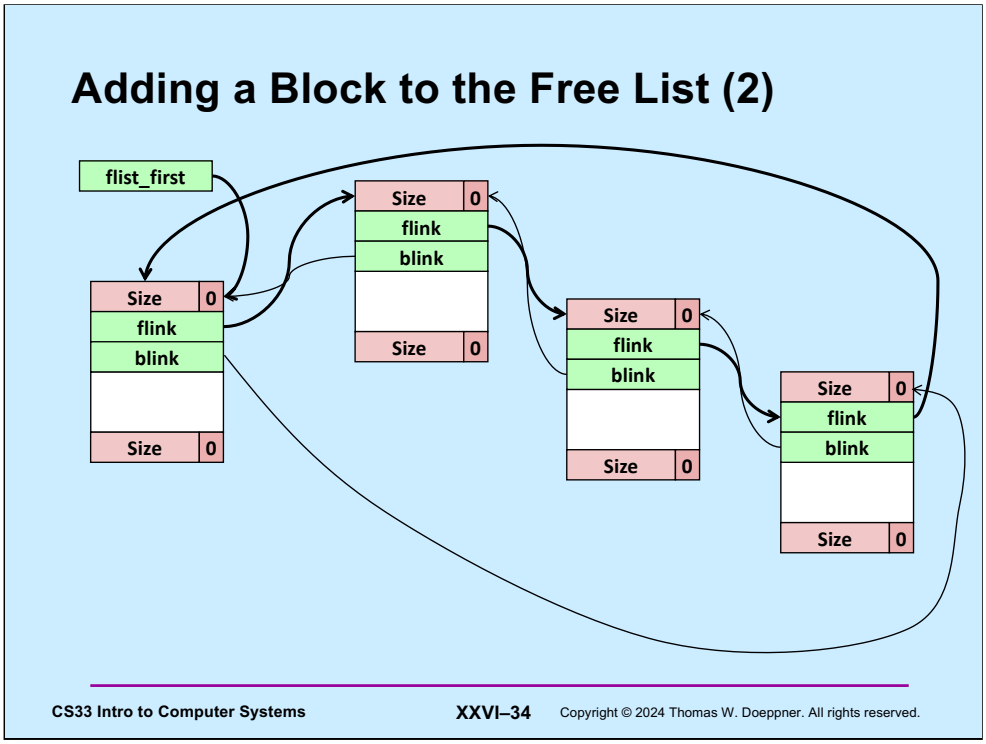

```
block_t *block_next(
    block_t *b) {
  return (block_t *)
    ((char *)b + block_size(b));
}

size_t block_next_allocated(
    block_t *b) {
  return block_allocated(
    block_next(b));
}
```


Similarly, we can determine if the next block is free.



An important operation is to add a block to the beginning of the free list. We start with a picture of the free list.



Here's what it looks like after we add the block.

Accessing the Object

```
block_t *block_flink(block_t *b) {
  return (block_t *)b->payload[0];
}

void block_set_flink(block_t *b, block_t *next) {
  b->payload[0] = (size_t)next;
}

block_t *block_blink(block_t *b) {
  return (block_t *)b->payload[1];
}

void block_set_blink(block_t *b, block_t *next) {
  b->payload[1] = (size_t)next;
}
```


Here are a few more simple functions we need to access and set fields of blocks.

Insertion Code

```
void insert_free_block(block_t *fb) {
  assert(!block_allocated(fb));
  if (flist_first != NULL) {
    block_t *last =
      block_blink(flist_first);
    block_set_flink(fb, flist_first);
    block_set_blink(fb, last);
    block_set_flink(last, fb);
    block_set_blink(flist_first, fb);
  } else {
    block_set_flink(fb, fb);
    block_set_blink(fb, fb);
  }
  flist_first = fb;
}
```


Using our functions, here's the code to insert a block at the beginning of the free list.

Performance

- Won’t all the calls to the accessor functions slow things down a lot?
  - yes — not just a lot, but tons
- Why not use macros (#define) instead?
  - the textbook does this
  - it makes the code impossible to debug
    - » gdb shows only the name of the macro, not its body
- What to do????

We've used a lot of functions without thinking about their effect on performance. While we know that the overhead of a function call is not great, there is still some overhead that might best be eliminated.

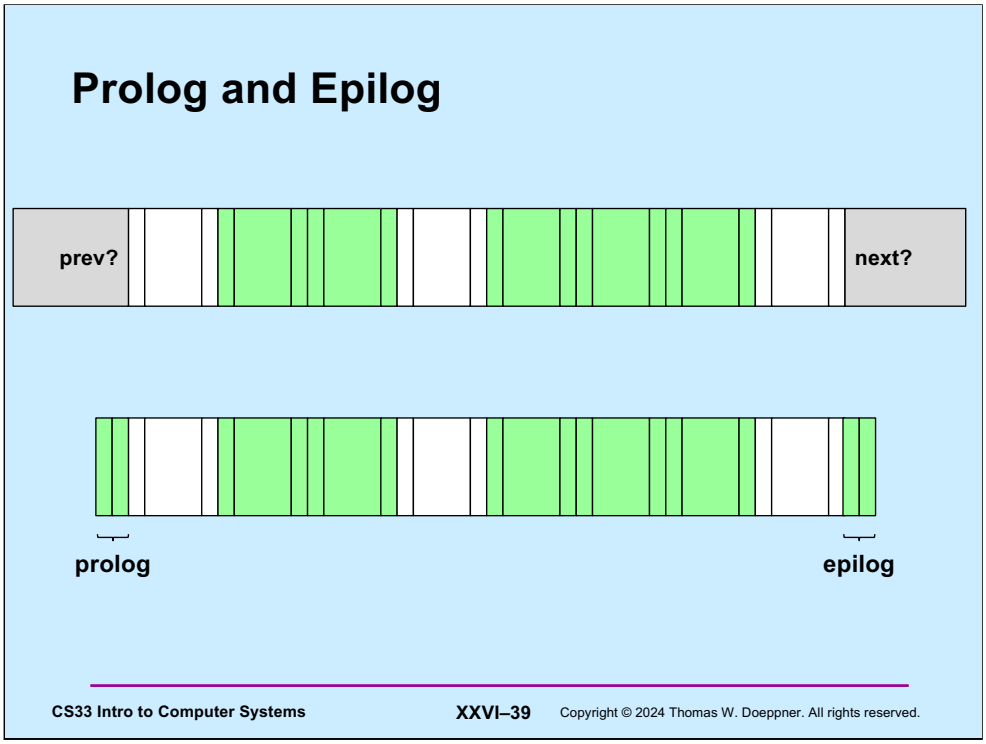
Inline Functions

```
static inline size_t block_size(
    block_t *b) {
  return b->size & -2;
}
```

- when debugging (–O0), the code is implemented as a normal function
  - » easy to debug with gdb
- when optimized (–O1, –O2), calls to the function are replaced with the body of the function
  - » no function-call overhead

If we declare a function to be **inline**, the C compiler is instructed to replace calls to the function with its actual code (unless –O0 is specified). Thus, inlined functions have no function-call overhead (though they do increase the total size of our code, which does come at some cost).

Note that inline functions are declared to be *static*. This makes it possible to have two .c files that use an inline function, with one compiled with –O0 and the other perhaps with –O1 – since the function is static, it can be different in the two source files.



The slide shows our heap, with allocated blocks shown in green and free blocks in white. At either end of each block are its tags.

An issue that comes up when implementing malloc/free is dealing with the first and last blocks, whether they are allocated or free. What is the **prev** block relative to the first block? What is the **next** block relative to the last block? Having to special-case the first and last blocks can help make your code unnecessarily complicated. To avoid these complications, we use **prolog** and **epilog** blocks. These are blocks of minimum size (containing just two tags and no payload) that are marked **allocated**. They are on either end of the list. Since they're marked allocated, when a check is made of the **prev** block relative to the first real block, it will always appear to be allocated, and similarly with the **next** block relative to the last real block.

Thus, the initial heap might consist of three blocks: the prolog, a block representing the initial free space, and an epilog. Of course, when the heap is expanded by calling **sbrk**, the epilog must be moved to the new end of the heap.

CS 33

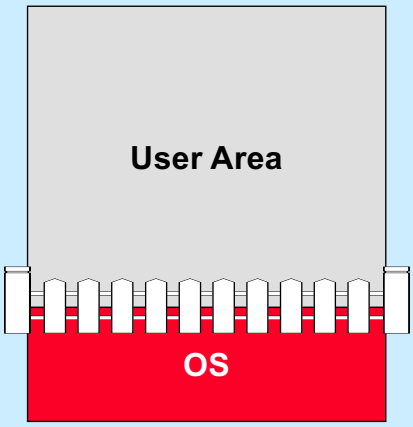
Virtual Memory

The Address-Space Concept

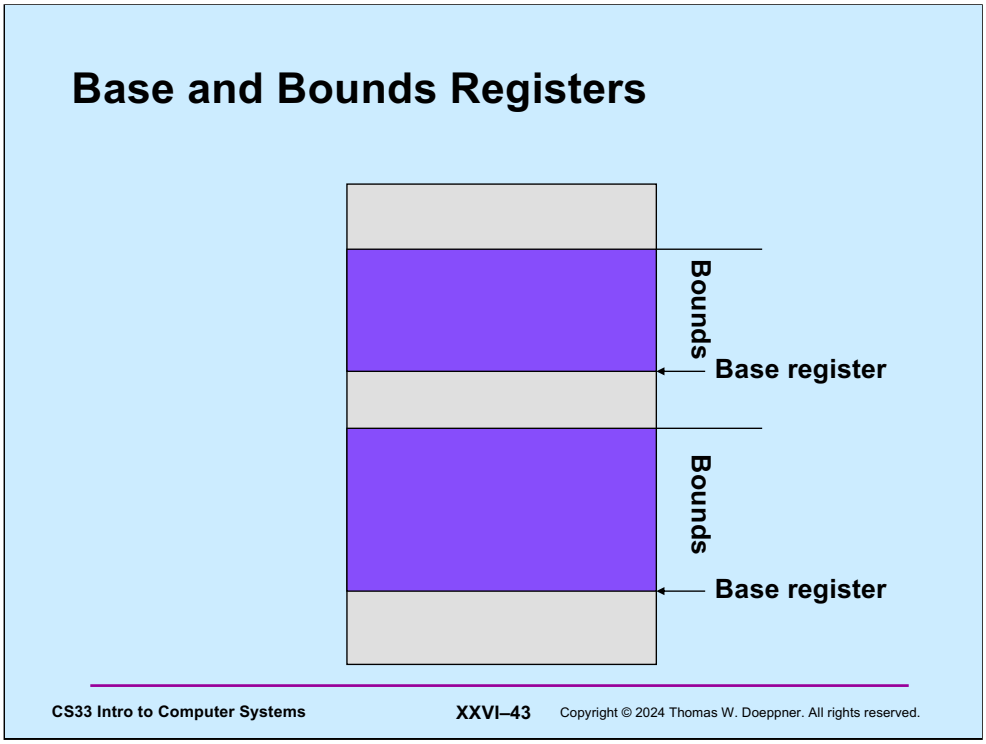
- **Protect processes from one another**
- **Protect the OS from user processes**
- **Provide efficient management of available storage**

The concept of the address space is fundamental in most of today's operating systems. Threads of control executing in different address spaces are protected from one another, since none of them can reference the memory of any of the others. In most systems (such as Unix), the operating system resides in address space that is shared with all processes, but protection is employed so that user threads cannot access the operating system. What is crucial in the implementation of the address-space concept is the efficient management of the underlying primary and secondary storage.

Memory Fence

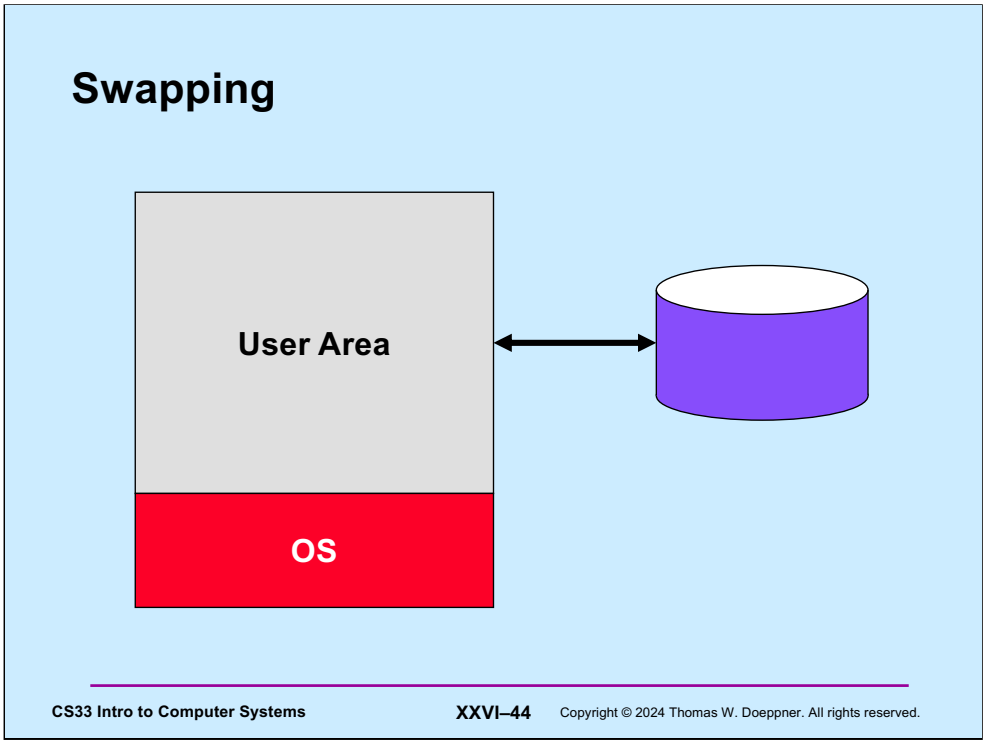


Early approaches to managing the address space were concerned primarily with protecting the operating system from the user. One technique was the hardware-supported concept of the **memory fence**: an address was established below which no user mode access was allowed. The operating system was placed below this point in memory and was thus protected from the user.



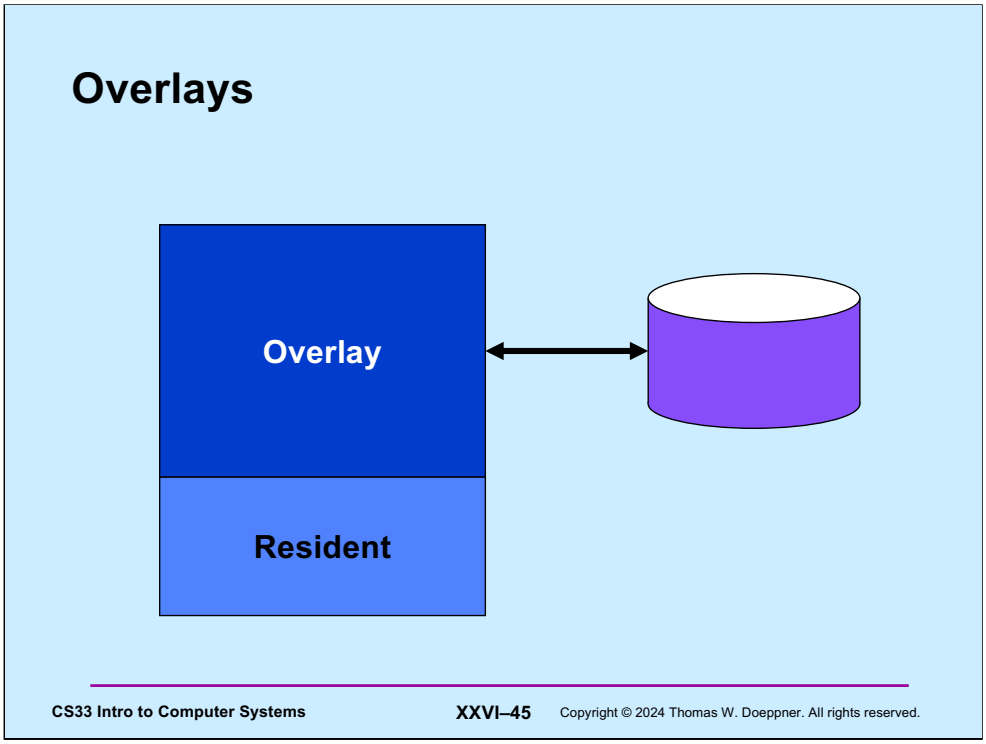
The memory-fence approach protected the operating system, but did not protect user processes from one another. (This wasn't an issue for many systems—there was only one user process at a time.) Another technique, still employed in some of today's systems, is the use of **base and bounds registers** to restrict a process's memory references to a certain range. Each address generated by a user process was first compared with the value in the bounds register to make certain that it did not reference a location beyond the process's range of memory, and then was modified by adding to it the value in the base register, ensuring that it did not reference a location before the process's range of memory.

A further advantage of this technique was to ensure that a process would be loaded into what appeared to be location 0 — thus no relocation was required at load time.



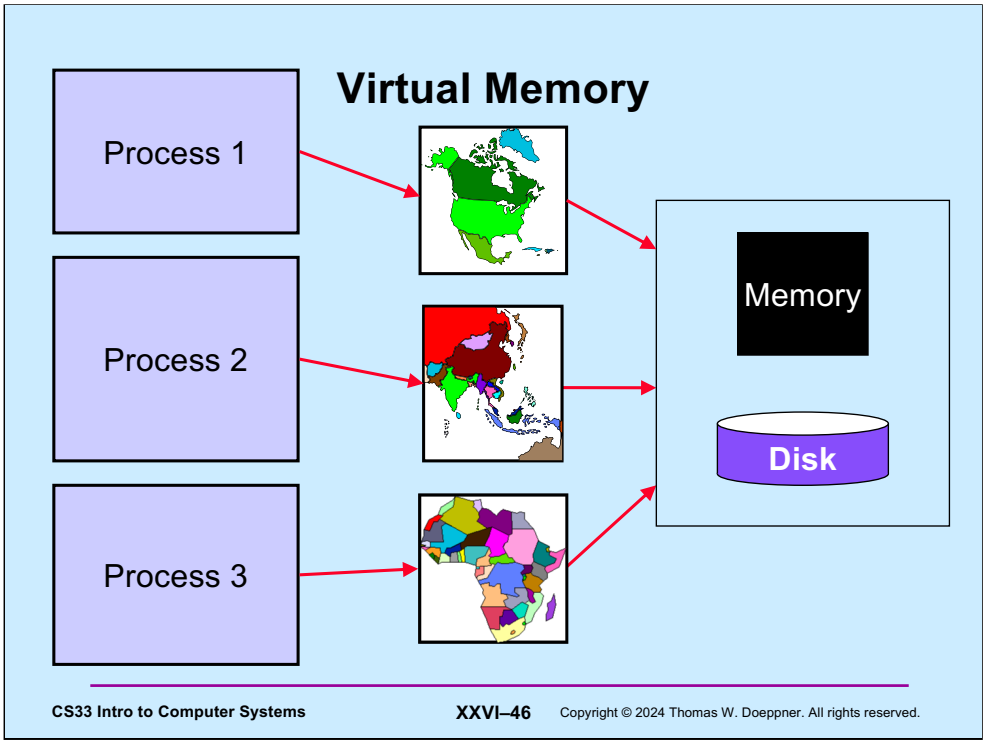
Swapping is a technique, still in use today, in which the images of entire processes are transferred back and forth between primary and secondary storage. An early use of it was for (slow) time-sharing systems: when a user paused to think, his or her process was swapped out and that of another user was swapped in. This allowed multiple users to share a system that employed only the memory fence for protection.

Base and bounds registers made it feasible to have a number of processes in primary memory at once. However, if one of these processes was inactive, swapping allowed the system to swap this process out and swap another process in. Note that the use of the base register is very important here: without base registers, after a process is swapped out, it would have to be swapped into the same location in which it resided previously.



The concept of overlays is similar to the concept of swapping, except that it applies to pieces of images rather than whole images and the user is in charge. Say we have 100 kilobytes of available memory and a 200-kilobyte program. Clearly, not all the program can be in memory at once. The user might decide that one portion of the program should always be resident, while other portions of the program need be resident only for brief periods. The program might start with routines A and B loaded into memory. A calls B; B returns. Now A wants to call C, so it first reads C into the memory previously occupied by B (it *overlays* B), and then calls C. C might then want to call D and E, though there is only room for one at a time. So, C first calls D, D returns, then C overlays D with E and then calls E.

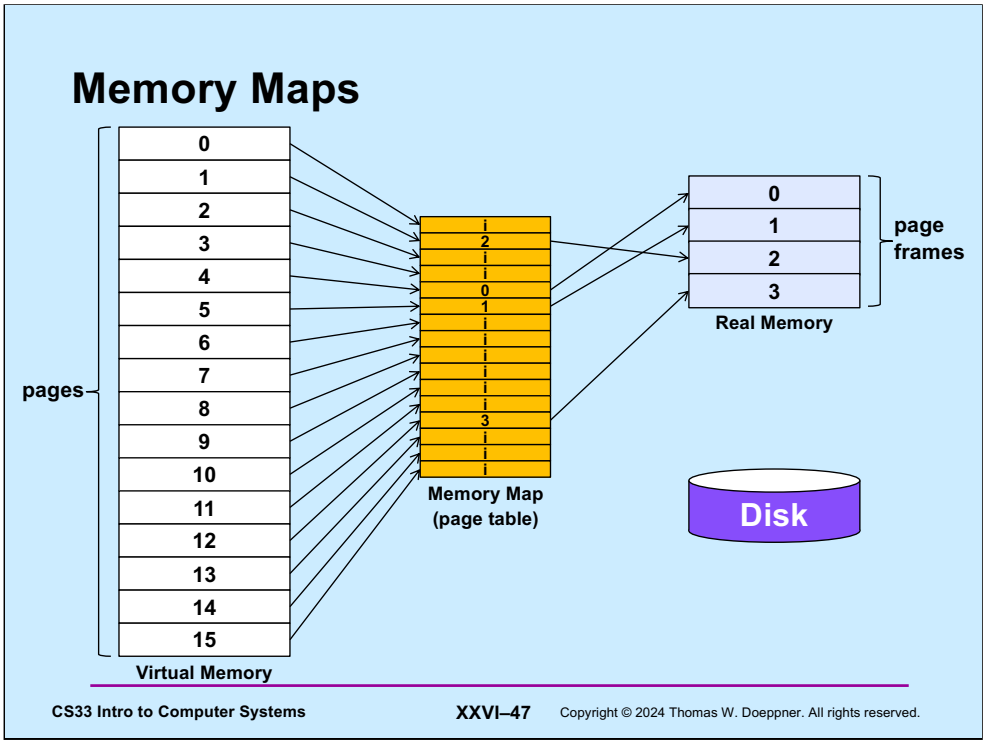
The advantage of this technique is that the programmer has complete control of the use of memory and can make the necessary optimization decisions. The disadvantage is that the programmer **must** make the necessary decisions to make full use of memory (the operating system doesn't help out). Few programmers can make such decisions wisely, and fewer still want to try.



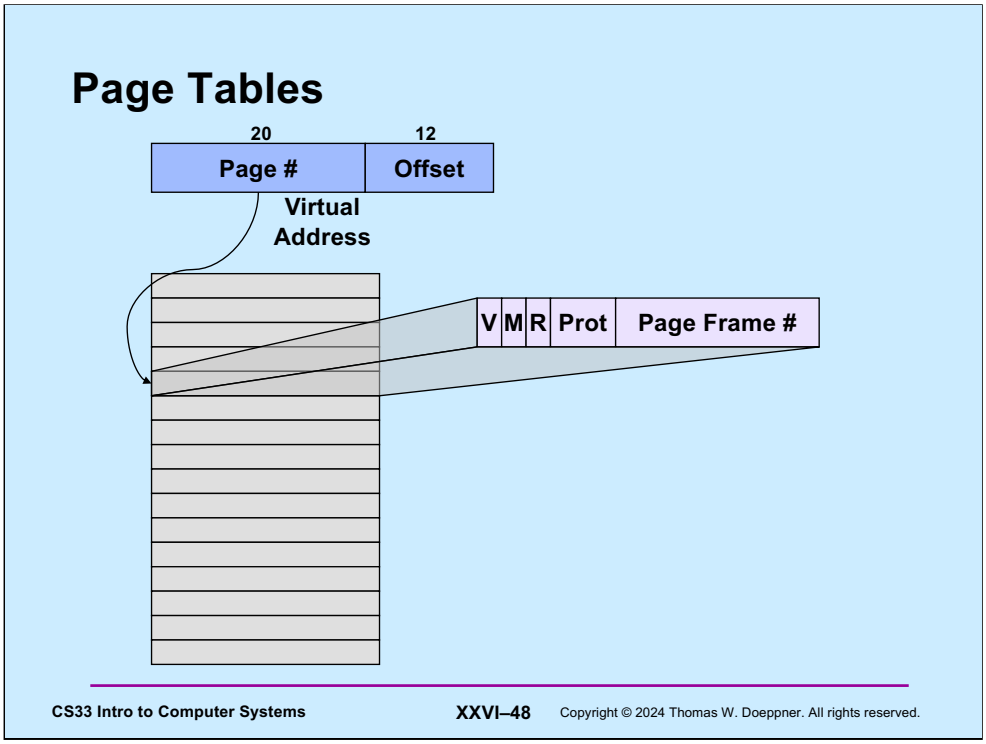
One way to look at virtual memory is as an automatic overlay technique: processes "see" an address space that is larger than the amount of real memory available to them; the operating system is responsible for the overlaying.

Put more abstractly (and accurately), virtual memory is the support of an address space that is independent of the size of primary storage. Some sort of mapping technique must be employed to map virtual addresses to primary and secondary stores. In the typical scenario, the computer hardware maps some virtual addresses to primary storage. If a reference is made to an unmapped address, then a fault occurs (a **page fault**) and the operating system is called upon to deal with it. The operating system might then find the desired virtual locations on secondary storage (such as a disk) and transfer them to primary storage. Or the operating system might decide that the reference is illegal and deliver a seg fault to the process.

As with base and bounds registers, the virtual memory concept allows us to handle multiple processes simultaneously, with the processes protected from one another.



Virtual memory (what the program sees) is divided into fixed-size pages (on the x86 these are usually 4 kilobytes in size). Real memory (DRAM) is also divided into fixed-size pieces, called page frames (though they're often referred to simply as pages). A memory map, implemented in hardware and often called a page table, translates references to virtual-memory pages into references to real-memory page frames. In general, virtual memory is larger than real memory, thus not all pages can be mapped to page frames. Those that are not are said to have invalid translations.



A page table is an array of *page table entries*. Suppose we have, as is the usual case for the x86, a 32-bit virtual address and a page size of 4096 bytes. The 32-bit address might be split into two parts: a 20-bit **page number** and a 12-bit **offset** within the page. When a thread generates an address, the hardware uses the page-number portion as an index into the page-table array to select a page-table entry, as shown in the picture. If the page is in primary storage (i.e. the translation is valid), then the validity bit in the page-table entry is set, and the page-frame-number portion of the page-table entry is the high-order bits of the location in primary memory where the page resides. (Primary memory is thought of as being subdivided into pieces called **page frames**, each exactly big enough to hold a page; the address of each of these page frames is at a "page boundary," so that its low-order bits are zeros.) The hardware then appends the offset from the original virtual address to the page-frame number to form the final, real address.

If the **validity bit** of the selected page-table entry is zero, then a page fault occurs and the operating system takes over. Other bits in a typical page-table entry include a **reference bit**, which is set by the hardware whenever the page is referenced, and a **modified bit**, which is set whenever the page is modified. We will see how these bits are used later in this lecture. The **page-protection bits** indicate who is allowed access to the page and what sort of access is allowed. For example, the page can be restricted for use only by the operating system, or a page containing executable code can be write-protected, meaning that read accesses are allowed but not write accesses.