JULIAN SHUN: Good afternoon, everyone. Let's get started. So welcome to the 11th lecture of 6.172. It seems that there are many fewer people here today than on Tuesday.

[LAUGHTER]

All right. So today we're going to talk about storage allocation. And it turns out that storage allocation is about both allocating memory and also freeing it. But in the literature, it's just called storage allocation, so that's the term we're going to use. And whenever you do a malloc or a free, then you're doing storage allocation. So how many of you have used malloc or free before? So hopefully all of you, since you needed it for the projects and homeworks.

So the simplest form of storage is a stack. And in a stack, you just have an array and a pointer. So here we have an array, which we call A, and then there's some portion of this array that's used for memory, and the rest of it is free-- it's not used. And then there's a pointer, sp, that points to the end of the used region in this stack. And if you want to allocate x bytes on the stack, all you do is, you just increment this sp pointer by x. And then, of course, you should also check for overflow to make sure that you don't actually go off the end of this array, because if you do that then you'll get a segmentation fault.

But actually, nowadays, compilers don't really check for stack overflow because your stack is usually big enough for most of your program, and when you do get a stack overflow, you'll just get a segfault and then you go debug your program. So for efficiency reasons, the stack overflow isn't actually checked. OK? And then it returns a pointer to the beginning of the memory that you just allocated, so that's just as sp minus x.

So that's pretty simple. And in fact, this is how the C call stack works. It also uses a stack discipline. So when you call a function, you save local variables and registers on the stack, and you also save the return address of the function that's calling another function. And then, when you return, you pop things off the stack. So you can also free things from the stack.

So what you do is, you just decrement sp by x if you want to free x bytes. So here we just decremented sp by x, and everything after sp is now considered to be free. And again, if you're careful, you would check for a stack underflow. But again, the compiler usually doesn't do this because if you do have a stack overflow, there's a bug in your program, and you'll get a segfault, and you should go fix it.

So allocating and freeing in a stack takes constant time because all you have to do is manipulate the stack pointer, so it's pretty efficient. However, you have to free consistently with the stack discipline. So the stack has limited applicability. Does anybody see why you can't do everything with just a stack? So what's one limitation of the stack? Yes?

AUDIENCE: [INAUDIBLE]

JULIAN SHUN: So it turns out that you can actually pass a compile time constant to make the stack bigger if you wanted to. There's actually a more fundamental limitation of the stack. Yes.

AUDIENCE: You can only read things in the reverse order in which you allocate them.

JULIAN SHUN: Yeah, so the answer is that you can only free the last thing that you allocated, so there's no way to free anything in the middle of this used region here. You have to free the last thing here because the stack doesn't keep track of the objects in the middle of this used region. So there's limited applicability, but it's great when it works because it's very efficient, and all of this code can essentially be inline. You don't have to have any function calls, so it's very fast.

And it also turns out that you can allocate on the call stack using this function called alloca. It's actually not a function. It's just a keyword that the compiler recognizes, and it will transform it to instructions that manipulate the stack. However, this function is now deprecated because it turns out that the compiler is actually more efficient when you're dealing with these fixed-size frames if you just allocate a pointer on the stack that points to some piece of memory on the heap.

But nevertheless, if you want to allocate on the call stack, you can call this alloca function, but you should check that is doing the right thing since it's now deprecated and the implementation is compiler dependent. So what's another type of storage besides the stack? So you can't do everything with a stack. So what else can we use? Yes?

AUDIENCE: Heap.

JULIAN SHUN: Yes. So we also have the heap, which is more general than the stack. So a stack looks very nice and tidy, and it's very efficient to use the stack, but it doesn't work for everything. So that's why we have the heap. And a heap is much more general, but it's very messy. It's very hard to organize this and work with it efficiently.

And for the rest of this lecture, I am going to be talking about how to manage memory in the heap. And I found these pictures on Stack Overflow, so maybe they're biased towards stacks. [CHUCKLES] OK, so how do we do heap allocation? So let's first start with fixed-size heap allocation, where we assume that all of the objects that we're dealing with are of the same size. In general this isn't true, but let's just start with this simpler case first.

OK, so as I said earlier, if you use malloc and free in C, then you're doing heap allocation. C++ has the new and delete operators, which work similarly to malloc and free. They also call the object constructor and destructor, and the C functions don't do that. And unlike Java and Python, C and C++ don't provide a garbage collector, so the programmer has to manage memory him or herself, and this is one of the reasons for the efficiency of C and C++, because there's no garbage collector running in the background.

However, this makes it much harder to write correct programs in C because you have to be careful of memory leaks, dangling pointers, and double freeing. So a memory leak is if you allocate something and you forget to free it and your program keeps running and allocating more and more stuff but without freeing it. Eventually, you're going to run out of memory, and your program is going to crash. So you need to be careful of memory leaks.

Dangling pointers are pointers to pieces of memory that you have already freed, and if you try to dereference a dangling pointer, the behavior is going to be undefined. So maybe you'll get a segmentation fault. Maybe you won't see anything until later on in your program because that memory might have been reallocated for something else, and it's actually legal to dereference that memory. So dangling pointers are very annoying when you're using C. If you're lucky, you'll get a segfault right away, and you can go fix your bug, but sometimes these are very hard to find.

There's also double freeing. So this is when you free something more than once. And again, this will lead to undefined behavior. Maybe you'll get a segfault, or maybe that piece of memory was allocated for something else, and then when you free it again, it's actually legal. But your program is going to be incorrect, so you need to be careful that you don't free something more than once.

And this is why some people prefer to use a language like Java and Python that provide these built-in garbage collectors. However, these languages are less efficient because they have a general-purpose garbage collector running in the background. So in this class, we're going to use C because we want to be able to write the fastest programs as possible, so we need to study how to manage memory.

And there are some tools you can use to reduce the number of memory bugs you have in your program. So there's memory checkers like AddressSanitizer and Valgrind, which can assist you in finding these pernicious bugs. So AddressSanitizer is a compiler instrumentation tool. When you compile your program, you pass a flag, and then, when you run your program, it's going to report possible memory bugs you have in your program.

And then Valgrind-- it works directly off the binaries. You don't need to do anything special when you compile it. You can just pass your binary to Valgrind, and if there is a memory bug, it might find it. But Valgrind tends to be slower than AddressSanitizer, and it tends to catch fewer bugs because it knows less about the program than AddressSanitizer. And AddressSanitizer sees the source code of the program and has more information, whereas Valgrind just works directly off the binary.

Also, don't confuse the heap with the heap data structure that you might have seen before in your algorithms or data structures courses. So these are two different concepts. The heap data structure in your algorithms course was a data structure used to represent a priority queue, where you can efficiently extract the highest priority element, and you can also update the priorities of elements in the set. And this could be used for algorithms like sorting or graph search.

But today we're going to be talking about another heap, which is the heap that's used for storage allocation. So don't get confused. So any questions so far? OK. All right. So we're going to first start with fixed-size allocations, since that's the easier case. So we're going to assume that every piece of storage has the same size. Some of these blocks are used, and some of them are unused.

And among the unused blocks, we're going to keep a list that we call the free list, and each block in this free list has a pointer to the next block in the free list. And since this memory is unused, we can actually use the memory to store a pointer as part of our storage allocator implementation. There's actually another way to do fixed-size allocations. Instead of using a free list, you could actually a place a bit for each block saying whether or not it's free, and then when you do allocation, you can use bit tricks.

But today I'm going to talk about the free list implementation. So to allocate one object from the free list, you set the pointer x to be free. So free is pointing to the first object in this free list. Then you set the free pointer to point to the next thing in the free list, so this is doing free equal to the next pointer of free. And then, finally, you return x, which is a pointer to the first object in the free list.

So here's an animation. So x is going to point to what free points to. You also need to check if free is equal to null, because if free is equal to null, that means there are no more free blocks in the free list, and the programmer should know this. You should return a special value. Otherwise, we're going to set the free pointer to point to the next thing in the free list, and then, finally, we return x to the program, and now the program has a block of memory that I can use.

There is still a garbage pointer in this block that we pass back to the program because we didn't clear it. So the implementation of the storage allocator could decide to zero this out, or it can just pass it back to the program and leave it up to the programmer to do whatever it wants with it. So in the latter case, the programmer should be careful not to try to dereference this pointer.

OK, so how about deallocation? So let's say we want to free some object x. What we do is, we just set the next pointer of x to be equal to free, so it's going to point to the first thing in the free list. And then we set free equal to x. So right. So now free is pointing to x, and this x object that we wanted to free now is a pointer to the first object in the original free list. So pretty simple. Any questions on this?

So this sort of acts like a stack in that the last thing that you've freed is going to be the first thing that you allocate, so you get temporal locality in that way. But unlike a stack, you can actually free any of the blocks and not just the last block that you allocated. So with a free list, allocating and freeing take constant time because you're just adjusting some pointers. It has good temporal locality, because as I said, the things that you freed most recently are going to be the things that are going to be allocated.

It has poor spatial locality due to external fragmentation, and external fragmentation means that your box of used memory are spread out all over the place in the space of all memory. And this can be bad for performance because it can increase the size of the page table, and it can also cause disk thrashing. So if you recall, whenever you access a page in virtual memory, it has to do address translation to the physical memory address. And if your memory is spread out across many pages in virtual memory, then you're going to have a lot of entries in the page table, because the page table is storing this mapping between the virtual memory address of the page and the physical memory address of the page.

So this can complicate the page table, make it less efficient to do lookups in it. And then if you have more pages than you can fit in your main memory, then this can cause disk thrashing because you have to move pages in and out of disk. The Translation Lookaside Buffer or TLB can also be a problem. Does anybody know what a TLB is?

Yes?

AUDIENCE: A cache of the result of translating from virtual memory to physical memory.

JULIAN SHUN: Yeah, so the TLB is essentially a cache for the page table, so it will cache the results of the translation from virtual memory addresses to physical memory addresses for the most recent translations. And looking up a translation in the TLB is much more efficient than going through the page table. And if you have a lot of external fragmentation, then you have a lot of pages that you might access, and this means that when you go to the TLB, it's more likely you'll get a TLB miss, and you have to go to the page table to look up the appropriate address. So that's why external fragmentation is bad.

So let's look at some ways to mitigate external fragmentation. So one way to do this is to keep a free list or a bitmap per disk page, and then when you want to allocate something, you allocate from the free list of the fullest page. So you sort of skew the memory that's being used to as few pages as possible. And when you free a block of storage, you just return it to the page on which that block resides. And if a page becomes completely empty-- there are no more items that are used on that page-- then the virtual memory system can page it out without affecting the program performance because you're not going to access that page anyways.

So this might seem counterintuitive. Why do we want to skew the items to as few pages as possible? So let's look at a simple example to convince ourselves why this is actually good for dealing with external fragmentation. So here I have two cases. In the first case, I have 90% of my blocks on one page and 10% of the blocks on the other page. In the second case, I have half of my blocks on one page and half on the other page.

So now let's look at the probability that two random accesses will hit the same page. So let's assume that all of the random accesses are going to go to one of the two pages. So in the first case, the probability that both of the accesses hit the first page is going to be 0.9 times 0.9, and then the probability that they both hit the second page is 0.1 times 0.1, and if you sum this up, you get 0.82. That's the probability that both of the random accesses are going to hit the same page.

In the other case, the probability that both of the accesses hit the first page is going to be 0.5 times 0.5. The second page is also going to be 0.5 times 0.5, so that sums to 0.5, and that means that there's only a 50% chance that two random accesses are going to hit the same page. So in the first case, you actually have a higher chance that the two random accesses hit the same page, and that's why we want to skew the items as much as possible so that we can reduce the external fragmentation. Any questions?

OK, so that was fixed-size heap allocation, and obviously you can't use that for many programs if you're allocating memory of different sizes. So now let's look at variable-size heap allocation. So we're going to look at one allocation scheme called binned free lists. And the idea is to leverage the efficiency of free list and also accept the bounded amount of internal fragmentation. So internal fragmentation is wasted space within a block, so that means when you allocate possibly more space than you're using, then there's some wasted space in there.

So in binned free lists, what we're going to do is, we're going to have a whole bunch of bins, and each bin is going to store blocks of a particular size. So here I'm going to say that bin k holds memory blocks of size 2 to the k, so I'm going to store blocks of sizes powers of 2. So why don't I just store a bin for every possible size? Does anybody know why? Why am I rounding up to powers of 2 here?

AUDIENCE: You'd have too many bins.

JULIAN SHUN: Yes, if I wanted a bin for every possible size, I would have way too many bins, and just the pointers to these bins are not going to fit in memory. So that's why I'm only using bins that store blocks of size 2 to the k. And now let's look at how I'm going to allocate x bytes from a binned free list. So what I'm going to do is, I'm going to look up the bin for which I should take a block from, and to get that, I'm going to take the ceiling of log base x.

This is log base 2, so recall that lg is log base 2. If that bin is nonempty, then I can just return a block from that bin. However, if that bin is empty, then I need to go to the next highest bin that's nonempty, and then I'm going to take a block from that bin and then split it up into smaller chunks and place them into smaller bins. And then I'll also get a chunk that is of the right size.

So for this example, let's say I wanted to allocate 3 bytes. The ceiling of log base 2 of x is 2, so I go to bin 2. But bin 2 is empty, so I need to look for the next bin that's not empty. And that's going to be bin 4. And I'm going to split up this block into smaller powers of 2. So in particular, I'm going to find a nonempty bin k prime greater than k and split up a block into sizes of 2 to the k prime minus 1, 2 to the k prime minus 2, all the way down to 2 to the k.

So I'm going to split it into sizes of all the powers of 2 less than 2 to the k prime and greater than or equal to 2 to the k. And I'm going to actually have two blocks of size 2 to the k, and one of those will be returned to the program. So here I'm going to split up this block. I'm going to place one of the smaller blocks in bin 3, one of them into bin 2, and then I also have another block here that I'm just going to return to the program.

So any questions on how this scheme works? OK, and if there are no larger blocks that exist-- so that means all of the bins higher than the bin I'm looking at are empty-- then I need to go to the OS to request for more memory. And then after I get that memory, I'll split it up so I can satisfy my allocation request. In practice, this exact scheme isn't used, so there are many variants of this scheme.

So it turns out that efficiency is very important for small allocations because there's not that much work performed on these small pieces of memory, and the overheads of the storage allocation scheme could cause a performance bottleneck. So in practice, you usually don't go all the way down to blocks of size 1. You might stop at blocks of size 8 bytes so that you don't have that much overhead, but this does increase the internal fragmentation by a little bit, because now you have some wasted space.

And then-- one second-- and then you can also group blocks into pages, as I said before, so that all of the blocks in the same page have the same size, and then you don't have to store the information of the size of the blocks. Yes.

AUDIENCE: How do you--

JULIAN SHUN: Yeah, so there are two commands you can use. One is called mmap and the other one is called sbrk. So those are system calls. You just call that, and then the OS will give you more memory, and then your storage allocator can use it. Yes?

AUDIENCE: They don't have to use something like this in order to implement those?

JULIAN SHUN: No, the standard implementation of malloc-- internally, it uses these commands, mmap and sbrk, to get memory from the OS, so the OS just gives you a huge chunk of memory. It doesn't split it up into smaller blocks or anything. That's up to the storage allocator to do. It just gives you a big chunk of memory, and then the storage allocator will break it up into smaller blocks. There are similar commands where you can free memory back to the OS when you're not using them anymore.

AUDIENCE: Can you explain the paging thing [INAUDIBLE]?

JULIAN SHUN: Yeah, so what I said was that you can actually keep blocks of different sizes on different pages, so then you don't actually have to store the size of each block. You can just look up what page that block resides in when you get the memory address. And then for each page, you have one field that stores the size of those blocks on that page. So this saves you the overhead of having to store information per block to figure out its size. Yeah.

AUDIENCE: --changing the size of the blocks.

JULIAN SHUN: Yeah, so I mean, if you do change the size of the blocks, then you can't actually use this scheme, so this is actually a variant where you don't change the size of the blocks. If you do change the size, then you have to change it for the entire page. Yeah, so there are many variants of memory allocators out there. This is just the simplest one that I described.

But it turns out that this exact scheme isn't the one that's used in practice. There are many variants. Like some allocators, instead of using powers of 2, they use Fibonacci numbers to determine the different bins. Yeah. Any other questions? So you'll actually get a chance to play around with implementing some allocators in project 3 and homework 6.

So let's briefly look at the storage layout of a program. So this is how our virtual memory address space is laid out. So we have the stack all the way at the top, and the stack grows downwards, so we have the high addresses up top and the low addresses below. Then we have the heap, which grows upward, and the heap and the stack basically grow towards each other, and this space is dynamically allocated as the program runs.

Then there's the bss segment, the data segment, and the text segment, which all reside below the heap. So the code segment just stores the code for your program. So when you load up your program, this code is going to put your program into this text segment. Then there's this data segment, which stores all of the global variables and static variables, these constants that you defined in your program. These are all stored in the data segment, and when you load your program you also have to read this data from disk and store it into the data segment.

Then there's the bss segment this segment is used to store all the on initialize variables in your program, and this is just initialized to 0 at the start of your program, since your program hasn't initialized it, so it doesn't matter what we set it to. And then the heap-- this is the memory that we're using when we're calling malloc and free. And then we have the stack, which we talked about.

So in practice, the stack and the heap are never actually going to hit each other because we're working with 64-bit addresses. So even though they're growing towards each other, you don't have to worry about them actually hitting each other. And another point to note is that if you're doing a lot of precomputation in your program, for example generating these huge tables of constants, those all have to be read from disk when you start your program and stored in this data segment. So if you have a lot of these constants, it's actually going to make your program loading time much higher.

However, it's usually OK to do a little bit of precomputation, especially if you can save a lot of computation at runtime, but in some cases it might actually be faster overall to just compute the things in memory when you start your program, because then you have to read stuff from disk. So here's a question. So since a 64-bit address space takes over a century to write at a rate of 4 billion bytes per second, we're never effectively going to run out of virtual memory. So why don't we just allocate out of virtual memory and never free anything? Yes?

AUDIENCE: If you allocate a bunch of small things in random places, then it's harder to update than a large segment?

JULIAN SHUN: Yeah, so one thing is that you have this issue of fragmentation. The blocks of memory that you're using are not going to be contiguous in memory, and then it makes it harder for you to find large blocks. So this is called external fragmentation, which I mentioned earlier. So if you do this, external fragmentation is going to be very bad. The performance of the page table is going to degrade tremendously because the memory that you're using is going to be spread all over virtual memory, and you're going to use many pages, and this leads to disk thrashing.

So you have to do a lot of swaps of pages in and out of disk. Your TLB hit rate is going to be very low. And another reason is that you're also going to run out of physical memory if you never free anything. So one of the goals of storage allocation is to try to use as little virtual memory as possible and to try to keep the used portions of the memory relatively compact. Any questions so far?

OK, so let's do an analysis of the binned free list storage allocation scheme. So here's a theorem. Suppose that the maximum amount of heap memory in use at any time by a program is M. If the heap is managed by a binned free list allocator, then the amount of virtual memory consumed by the heap storage is upper bounded by M log M. Does anybody have an intuition about why this theorem could be true? So how many bins do we have, at most?

AUDIENCE: [INAUDIBLE]

JULIAN SHUN: Right. So the number of bins we have is upper bounded by log M, and each bin is going to use order M memory. So let's look at this more formally. So an allocation request for a block of size x is going to consume 2 to the ceiling of log base 2 of x storage, which is upper bounded by 2x, so we're only wasting a factor of 2 storage here. So therefore, the amount of virtual memory devoted to blocks of size 2 to the k is at most 2M.

And since there are at most log base 2 of M free lists, the theorem holds just by multiplying the two terms. So you can only have log base 2 of M free lists because that's the maximum amount of memory you're using, and therefore your largest bin is only going to hold blocks of size M. And it turns out that the bin free list allocation scheme is theta of 1 competitive with the optimal allocator, and here an optimal locator knows all of the memory requests in the future.

So it can basically do a lot of clever things to optimize the memory allocation process. But it turns out that the binned free list is only going to be a constant factor worse than the optimal allocator. This is assuming that we don't coalesce blocks together, which I'll talk about on the next slide. It turns out that this constant is 6, so Charles Leiserson has a paper describing this result. And there's also a lower bound of 6, so this is tight.

So coalescing. So coalescing is when you splice together smaller blocks into a larger block. So you can do this if you have two free blocks that are contiguous in memory. This will allow you to put them together into a larger block. So binned free lists can sometimes be heuristically improved by doing coalescing, and there are many clever schemes for trying to find adjacent blocks efficiently.

So there's something called the buddy system, where each block has a buddy that's contiguous and memory. However, it turns out that this scheme especially, the buddy system scheme, has pretty high overhead. So it's usually going to be slower than just the standard binned free list algorithm. There are no good theoretical bounds that exist that prove the effectiveness of coalescing, but it does seem to work pretty well in practice at reducing fragmentation because heap storage tends to be deallocated as a stack or in batches.

So what I mean by this is that the objects that you free tend to be pretty close together in memory. So if you deallocate as a stack, then all of them are going to be near the top of the stack. And when you deallocate in batches-- this is when you do allocate a whole bunch of things that you allocated together in your program. For example, if you have a graph data structure and you allocated data for the vertices all at the same time, then when you deallocate them all together, this is going to give you a chunk of contiguous memory that you can splice together.

OK, so now let's look at garbage collection. This is going to be slightly different from storage allocation. So the idea of garbage collection is to free the programmer from having to free objects. So languages like Java and Python, they have built-in garbage collectors, so the programmer doesn't have to free stuff themselves, and this makes it easier to write programs because you don't have to worry about double freeing and dangling pointers and so forth. So a garbage collector is going to identify and recycle the objects that the programmer can no longer access so that these memory objects can be used for future allocations.

And in addition to having a built-in garbage collector, you can also create your own garbage collector in C, which doesn't have a garbage collector. So if you have an application, you can actually create a special-purpose garbage collector that might be more efficient than a general garbage collector. Yes?

AUDIENCE: This is the previous topic, but why [? INAUDIBLE ?] order of M memory?

JULIAN SHUN: Why is it not order M?

AUDIENCE: Yeah.

JULIAN SHUN: Because for each of the bins, you could use up to order M memory. So if you don't do coalescing, basically, I could have a bunch of small allocations, and then I chop up all of my blocks, and then they all go into smaller bins. And then I want to allocate something larger. I can't just splice those together. I have to make another memory allocation. So if you order your memory requests in a certain way, you can make it so that each of the bins has order M memory.

OK, so for garbage collection, let's go over some terminology. So there are three types of memory objects, roots, live objects, and dead objects. Roots are objects that are directly accessible by the program, so these are global variables, things on the stack, and so on. Then there are live objects, which are reachable by following the roots via pointers, and then finally, there are dead objects, and these objects are inaccessible via sequences of pointers. And these can be recycled because the programmer can no longer reach these dead objects.

So in order for garbage collection to work in general, you need to be able to have the garbage collector identify pointers, and this requires strong typing. So languages like Python and Java have strong typing, but in C, it doesn't have strong typing. This means that when you have a pointer you don't actually know whether it's a pointer. Because a pointer just looks like an integer. It could be either a point or an integer.

You can cast things in C. You can also do pointer arithmetic in C. So in contrast, in other languages, once you declare something to be a pointer, it's always going to be a pointer. And for those languages that have strong typing, this makes it much easier to do garbage collection. You also need to prohibit doing pointer arithmetic on these pointers. Because if you do pointer arithmetic and you change the location of the pointer, then the garbage collector no longer knows where the memory region starts anymore.

In C, sometimes you do do pointer arithmetic, and that's why you can't actually have a general-purpose garbage collector in C that works well. So let's look at one simple form of garbage collection. And this is called reference counting. The idea is that, for each object, I'm going to keep a count of the number of pointers referencing that object. And if the count ever goes to 0, then that means I can free that object because there are no more pointers that can reach that object.

So here, I have a bunch of roots. So these are directly accessible by my program. And then I have a bunch of objects that can be reached via following pointers starting from the root. And then each of them have a reference count that indicates how many incoming pointers they have. So let's say now I change one of these pointers.

So initially, I had a pointer going to here, but now I changed it so that it goes down here. So what happens now is I have to adjust a reference counts of both of these objects. So this object here, now it doesn't have any incoming pointers, so I have to decrement its reference count. So that goes to 0. And then for this one, I have to increment its reference count, so now it's 3. And now I have an object that has a reference count of 0, and with this reference counting algorithm, I can free this object.

So let's go ahead and free this object. But when I free this object, it actually has pointers to other objects, so I also have to decrement the reference counts of these other objects when I free this object. So I'm going to decrement the counts. And now it turns out that this object also has a reference count of 0, so I can free that, as well. And in general, I just keep doing this process until the reference counts of the objects don't change anymore, and whenever I encounter an object with a reference count of 0, I can free it immediately.

And the memory that I freed can be recycled. It can be used for future memory allocations. So questions on how the reference counting procedure works? So there's one issue with reference counting. Does anybody see what the issue is? Yes?

AUDIENCE: What if it has a reference to itself?

JULIAN SHUN: Yes. So what if it has a reference to itself? More generally, what if it has a cycle? You can't ever collect garbage collect a cycle when you're using reference counts. So here we have a cycle of length 3. They all have a reference count of 1, but you can never reach the cycle by following pointers from the root, and therefore, you can never delete any object in the cycle, and the reference counts are always going to be non-zero.

So let's just illustrate the cycle. And furthermore, any object that's pointed to by objects in the cycle cannot be garbage collected, as well, because you can't garbage collect the cycle, so all the pointer is going out of the objects in the cycle are always going to be there. So there could be a lot of objects downstream from this object here that can't be garbage collected, so this makes it very bad. And as we all know, uncollected garbage stinks, so we don't want that.

So let's see if we can come up with another garbage collection scheme. So it turns out that reference counting is actually pretty good when it does work because it's very efficient and simple to implement. So if you know that your program doesn't have these cycles in them among pointers, then you can use a reference counting scheme. There are some languages, like Objective-C, that have two different types of pointers, strong pointers and weak pointers. And if you're doing reference counting with a language with these two types of pointers, the reference count only stores the number of incoming strong pointers.

And therefore, if you define these pointers inside a cycle to be weak pointers, they're not going to contribute to the reference count, and therefore you can still garbage collect. However, programming with strong or weak pointers can be kind of tricky because you need to make sure that you're not dereferencing something that a weak pointer points to because that thing might have been garbage collected already, so you need to be careful. And C doesn't have these two types of pointers, so we need to use another method of garbage collection to make sure we can garbage collect these cycles.

So we're going to look at two more garbage collection schemes. The first one is called mark-and-sweep, and the second one is called stop-and-copy. So first we need to define a graph abstraction. So let's say we have a graph with vertices V and edges E. And the vertex at V contains all of the memory objects in memory, and the edges E are directed edges between objects. So there's a directed edge from object A to object B if object A has a pointer to object B.

And then, as we said earlier, the live objects are the ones that are reachable from the roots, so we can use a breadth-first-search-like procedure to find all of the live objects. So we just start our breadth-first search from the roots, and we'll mark all of the objects that can be reachable from the roots. And then everything else that isn't reached, those are available to be reclaimed. So we're going to have a FIFO queue, First-In, First-Out queue, for our breadth-first search.

This is represented as an array. And we have two pointers, one to the head of the queue and one to the tail of the queue. And here let's look at this code, which essentially is like a breadth-first search. So we're first going to go over all the vertices in our graph, and we're going to check if each vertex v is a root. If it is a root, we're going to set its mark to be 1, and we're going to place the vertex onto the queue. And otherwise, we're going to set the mark of v to be 0.

And then while the queue is not empty, we're going to dequeue the first thing from the queue. Let that be u. Then we're going to look at all the outgoing neighbors of u. So these are vertices v such that there is a directed edge from u to v. We're going to check if v's mark is equal to 0. If it is, that means we haven't explored it yet, so we'll set its mark to be 1, and we place it onto the queue. And if the neighbor has already been explored, then we don't have to do anything.

So let's illustrate how this algorithm works on this simple graph here. And for this example, I'm just going to assume that I have one root, vertex r. In general, I can have multiple routes, and I just place all of them onto the queue at the beginning, but for this example, I'm just going to have a single root. So I'm going to place it onto the queue, and the location that I place it is going to be where the tail pointer points to. And after I placed it on the queue, I increment the tail pointer.

Now I'm going to take the first thing off of my queue, which is r, and I'll explore my neighbors. So the neighbors are b and c here. Both of them haven't been marked yet, so I'm going to mark them, and I'm going to indicate the marked vertices with shaded blue. And I'll place them onto the queue. Now I'm going to take the next thing, b. I'm going to check its neighbors.

It only has a neighbor to c, but c is already on the queue. It's already marked, so I don't have to do anything. Now I dequeue c, and c has neighbors d and e, so I place them onto the queue. d doesn't have any outgoing neighbors, so I don't to do anything. Now when I dequeue e, it has neighbors f. When I dequeue f, it has a neighbor g, and when I dequeue g, it doesn't have any neighbors.

So now my queue is empty, and my breadth-first search procedure finishes. So at this point, I've marked all of the objects that are accessible from the root, and all of the unmarked objects can now be garbage collected because there is no way I can access them in the program. So the mark-and-sweep procedure has two stages.

The first stage is called the mark stage, where I use a breadth-first search to mark all of the live objects. And the sweep stage will scan over memory to free the unmarked objects. So this a pretty simple scheme. There is one issue with this scheme. Does anybody see what the possible issue is? Yes?

AUDIENCE: You have to scan over all the [INAUDIBLE].

JULIAN SHUN: Yeah, so that's one issue, where you have to scan over all of memory. There are some variants of mark-and-sweep where it keeps track of just the allocated objects, so you only have to scan over those instead of the entire memory space. Besides that, are there any other possible issues with this? Yes?

AUDIENCE: This also requires that you [INAUDIBLE] strong typing.

JULIAN SHUN: Right, so let's assume that we do have strong typing. Any other possible limitations? Anybody else? Think I called on-- yeah.

AUDIENCE: [INAUDIBLE] reference counting, you can see the object that has a reference to it, whereas for here you can find everything that would not be garbage collected [INAUDIBLE].

JULIAN SHUN: Yeah, so for the scheme that I described, you have to look over all of the things that don't have references to it. So that is another overhead. So those are all issues. Good. The issue I want to get at is that the mark-and-sweep algorithm that I presented here doesn't deal with fragmentation. So it doesn't compact the live objects to be contiguous in memory. It just frees the ones that are unreachable, but it doesn't do anything with the ones that are reachable.

So let's look at another procedure that does deal with fragmentation. This is called the stop-and-copy garbage collection procedure. At a high level, it's pretty similar to the mark-and-sweep algorithm. We're still going to use a breadth-first search to identify all of the live objects.

But if you look at how this breadth-first search is implemented, is there any information you can use here to try to get the live objects to be contiguous in memory? Does anybody see anything here that we can use to try to reduce fragmentation? Yes?

AUDIENCE: [INAUDIBLE]

JULIAN SHUN: Yes, so the answer is that the objects that we visited are contiguous on the queue. So in the mark-and-sweep algorithm, I just place the IDs of the vertices on the queue, but if I just place the actual objects onto the queue instead, then I can just use my queue as my new memory. And then all of the objects that are unreachable will be implicitly deleted.

So this procedure here will deal with external fragmentation. So let's see how this works. So we're going to have two separate memory spaces, the FROM space and the TO space. So in the FROM space, I'm just going to do allocation and freeing on it until it becomes full. So when I allocate something I place it at the end of this space. When I free something, I just market as free, but I don't compact it out yet.

And when this FROM space becomes full, then I'm going to run my garbage collection algorithm, and I'm going to use the TO space as my queue when I do my breadth-first search. So after I run my breadth-first search, all of the live objects are going to appear in the TO space in contiguous memory since I used the TO space as my queue. Right, and then I just keep allocating stuff from the TO space and also marking things as deleted when I free them until the TO space becomes full. Then I do the same thing, but I swap the roles of the TO and the FROM spaces.

So this is called the stop-and-copy algorithm. There is one problem with this algorithm which we haven't addressed yet. Does anybody see what the potential problem is? Yes?

AUDIENCE: If nothing is dead, then you're copying over your entire storage every single time.

JULIAN SHUN: Yeah, so that's one good observation. If nothing is dead, then you're wasting a lot of work because you have to copy this. Although with the mark-and-sweep algorithm, you still have to do some copying, although you're not copying the entire objects. You're just copying the IDs. There's actually a correctness issue here. So does anybody see what the correct this issue is? Yes?

AUDIENCE: So maybe the pointers in the TO space have to be changed in order to point to the new [INAUDIBLE].

JULIAN SHUN: Yeah, so the answer is that if you had pointers that pointed to objects in the FROM space, if you move your objects to the TO space, those pointers aren't going to be correct anymore. So if I had a pointer to a live object before and I moved my live object to a different memory address, I need to also update that pointer. So let's see how we can deal with this.

So the idea is that, when an object is copied to the TO space, we're going to store a forwarding pointer in the corresponding object in the from space, and this implicitly marks that object as moved. And then when I remove an object from the FIFO queue in my breadth-first search, in the TO space I'm going to update all of the pointers by following these forwarding pointers. So let's look at an example of how this works.

So let's say I'm executing the breadth-first search, and this is my current queue right now.

What I'm going to do is, when I dequeue an element from my queue, first I'm going to place the neighboring objects that haven't been explored yet onto the queue. So here it actually has two neighbors, but the first one has already been placed the queue, so I can ignore it. And the second one hasn't been placed on the queue yet, so I place it onto the queue.

And then I'm also going to-- oh, so this object also has a pointer to something in the FROM space, which I'm not going to change at this time. But I am going to store a forwarding pointer from the object that I moved from the FROM space to the TO space, so now it has a pointer that tells me the new address. And then, for the object that I just dequeued, I'm going to follow the forwarding pointers of its neighbors, and that will give me the correct addresses now.

So I'm going to update the pointers by just following the forwarding pointer. So the first pointer pointed to this object, which has a forwarding pointer to this, so I just make a point to this object in the TO space. And then similarly for the other pointer, I'm going to make it point to this object. So that's the that's the idea how of how to adjust the pointers.

One question is, why can't we just adjust the pointer is when we enqueue the object? So why do I have to adjust the pointers when I dequeue an object? Yes?

AUDIENCE: Because we haven't processed its neighbors yet.

JULIAN SHUN: Yeah, so the answer is that, when you enqueue object, you don't actually know where your neighbors are going to reside in the TO space. And you only know that when you dequeue the object, because when you dequeue the object, you must have explored your neighbors, and therefore you can generate these forward pointers. So any questions on this scheme?

So how much time does it take to do the stop-and-copy procedure? So let's say n is the number of objects and the number of pointers I have, so it's the sum of the number of objects and number of pointers. How much time would it take to run this algorithm?

AUDIENCE: [INAUDIBLE]

JULIAN SHUN: Yeah, so it's just going to be linear time because we're running a breadth-first search, and that takes linear time. You also have to do work in order to copy these objects to the TO space, so you also have to do work proportional to the number of bytes that you're copying over. So it's linear in the number of objects, the number of pointers, and the total amount of space or copying over.

And the advantage of this scheme is that you don't actually need to go over the objects that aren't reachable because those are going to be implicitly deleted because they're not copied over to the TO space, whereas in the mark-and-sweep procedure, you had to actually go through your entire memory and then free all the objects that aren't reachable. So this makes the stop-and-copy procedure more efficient, and it also deals with the external fragmentation issue.

So what happens when the FROM space becomes full? So what you do is, you're going to request a new heap space equal to the used space, so you're just going to double the size of your FROM space. And then you're going to consider the FROM space to be full when the newly allocated space becomes full, so essentially what you're going to do is, you're going to double the space, and when that becomes full, you're going to double it again, and so on.

And with this method, you can amortize the cost of garbage collection to the size of the new heap space, so it's going to be amortized constant overhead per byte of memory. And this is assuming that the user program is going to touch all of the memory that it allocates. And furthermore, the virtual memory space required by this scheme is just a constant times the optimal if you locate the FROM and the TO spaces in different regions of virtual memory so that they can't interfere with each other.

And the reason why it's a constant times the optimal is because you only lose a factor of 2 because you're maintaining two separate spaces. And then another factor of 2 comes from the fact that you're doubling the size of your array when it becomes full and up to half of it will be unused. But it's constant times optimal since we're just multiplying constants together.

And similarly, when you're FROM space becomes relatively empty-- for example, if it's less than half full-- you can also release memory back to the OS, and then the analysis of the amortized constant overhead is similar. OK, any other questions? OK, so there's a lot more that's known and also unknown about dynamic storage allocation, so I've only scratched the surface of dynamic storage allocation today.

There are many other topics. For example, there's the buddy system for doing coalescing. There are many variants of the mark-and-sweep procedure. So there are optimizations to improve the performance of it. There's generational garbage collection, and this is based on the idea that many objects are short-lived, so a lot of the objects are going to be freed pretty close to the time when you allocate it. And for the ones that aren't going to be freed, they tend to be pretty long-lived.

And the idea of generational garbage collection is, instead of scanning your whole memory every time, you just do work on the younger objects most of the time. And then once in a while, you try to collect the garbage from the older objects because those tend to not change that often. There's also real-time garbage collection. So the methods I talked about today assume that the program isn't running when the garbage collection procedure is running, but in practice, you might want to actually have your garbage collector running in the background when your program is running, but this can lead to correctness issues because the static algorithms I just described assume that the graph of the objects and pointers isn't changing, and when the objects and pointers are changing, you need to make sure that you still get a correct answer.

Real-time garbage collection tends to be conservative, so it doesn't always free everything that's garbage. But for the things that it does decide to free, those can be actually reclaimed. And there are various techniques to make real-time garbage collection efficient. One possible way is, instead of just having one FROM and TO space, you can have multiple FROM and TO spaces, and then you just work on one of the spaces at a time so that it doesn't actually take that long to do garbage collection. You can do it incrementally throughout your program.

There's also multithreaded storage allocation and parallel garbage collection. So this is when you have multiple threads running, how do you allocate memory, and also how do you collect garbage in the background. So the algorithms become much trickier because there are multiple threads running, and you have to deal with races and correctness issues and so forth. And that's actually a topic of the next lecture.

So in summary, these are the things that we talked about today. So we have the most basic form of storage, which is a stack. The limitation of a stack is that you can only free things at the top of the stack. You can't free arbitrary things in the stack, but it's very efficient when it works because the code is very simple. And it can be inlined, and in fact this is what the C calling procedure uses. It places local variables in the return address of the function on the stack.

The heap is the more general form of storage, but it's much more complicated to manage. And we talked about various ways to do allocation and deallocation for the heap. We have fixed-size allocation using free lists, variable-size allocation using binned free lists, and then many variants of these ideas are used in practice. For garbage collection, this is where you want to free the programmer from having to free objects.

And garbage collection algorithms are supported in languages like Java and Python. We talked about various ways to do this reference counting, which suffers from the limitation that it can't free cycles. Mark-and-sweep and stop-and-copy-- these can free cycles. The mark-and-sweep procedure doesn't deal with external fragmentation, but the stop-and-copy procedure does.

We also talked about internal and external fragmentation. So external fragmentation is when your memory blocks are all over the place in virtual memory. This can cause performance issues like disk thrashing and TLB misses. Then there's internal fragmentation, where you're actually not using all of the space in the block that you allocate. So for example, in the binned free list algorithm, you do have a little bit of internal fragmentation because you're always rounding up to the nearest power of 2 greater than the size you want, so you're wasting up to a factor of 2 in space.

And in project 3, you're going to look much more at these storage allocation schemes, and then you'll also get to try some of these in homework 6. So any other questions? So that's all I have for today's lecture.