CHARLES LEISERSON: Today we're going to talk about dynamic storage allocation, which is a hugely rich and interesting topic, and hopefully cover some of the basics of dynamic storage allocation today. I want to start out with perhaps the simplest storage allocation scheme, and that's stack allocation, which I think you're probably fairly familiar with. So stack allocation is just an array and a pointer into the array, an index or a pointer into the array. And when you have a stack in a computer memory, you're talking about the array of all memory, and the stack pointer pointing that's used, for example, for calls.

The reason stack allocation is neat is because it's so simple. If I want to allocate x bytes, then what I do, is I just simply increment my stack pointer by x bytes, and then I return the original position of the stack pointer as being the location of the storage that's just been allocated. So pretty simple. Typically if you write a stack allocator, the routines are so simple they get inlined, although it's of course advantageous to say that things are inlined, but most compilers these days will inline. And as you can see, it's just a couple of operations here to perform this calculation.

To deallocate the bytes, we basically-- oh, yeah, I should mention, this isn't checking for overflow. So technically, one should check for overflow. One way to do that is for the assert package, and that way, at run time, you don't have to pay the extra overhead of checking. And see, these operations are so cheap that actually checking for overflow is perhaps as expensive as doing the rest of the operations themselves. Checking to see whether you went over the right end of the array here.

So to free x bytes, you basically just simply decrement the stack pointer. And of course, technically, you should also check for underflow, make sure somebody didn't say to deallocate more bytes than were actually already allocated on the stack. And so course, check for stack underflow.

So this is pretty good, because allocating and freeing take just constant time. Very quick. But you're required to free memory consistent with the stack discipline, which is the last in, first out discipline. So you don't get to free something that you allocated long ago until you've freed everything else that's been allocated since. So therefore, it has limited applicability, but it's great when it works.

Now it turns out, sometimes you can actually use the call stack yourself to store stack values, using a function called alloca. But this function is now deprecated, meaning that people advise you not to use it in any modern programming. And part of the reason is that the compiler is more efficient with fixed size stack frames. You remember when we went through how the compiler actually lays out memory and does the calling convention? If you have a fixed size stack frame, then the frame pointer and the stack pointer are redundant, and so you can reclaim a register, namely, the frame pointer register, and do everything off the stack pointer. But if you use things like alloca, which are pushing the sides of the stack frame, then you actually have to keep both pointers there, and that's a little bit less efficient in terms of the register usage. OK?

So that's pretty much stack storage. Any questions about stacks? Yeah?

AUDIENCE: Is there anything that detects overflow or underflow without using inserts? Like, if you wanted to check--

CHARLES LEISERSON: Yeah, you couldn't check explicitly. Just say, you know, for this, is before you check that, make sure that SP plus x, is it bigger than the right end?

AUDIENCE: But how do you know where the right end is?

CHARLES LEISERSON: Well, because I'm assuming here that in this case, you are allocating it. Now, if you're talking about using the call stack, you mean? You'll never run out. We'll do that analysis in a little bit. But you'll never run out of stack space. Although as a practical matter, most compilers assume a stack of a megabyte, and so if you overflow that, your program will fail in any of a variety of ways.

AUDIENCE: You can do guard pages.

CHARLES LEISERSON: You can do guard pages, which don't always work. But let me explain guard pages, OK?

So what's sometimes done is, if I have my stack growing down in memory-- OK, so here's my stack, and it's growing this way-- and I want to make sure, for example, the thing that's typically growing up this way is heat, and I want to make sure that they don't collide somewhere in here.

So one thing I can do is put some pages in here that are protected. These are read-only pages. Or actually even better is no permissions. So I can put no permissions so that if you ever try to read or write that page, it automatically generates a fault. And so the runtime will sometimes do that.

The difficulty is, what happens if you increased your stack so much that you went beyond your guard pages? But as a practical matter most of the times that people exceed the guard pages is only by small amount. And so just putting a few guard pages in there will make sure that you can use the memory management hardware, which basically ends up being free from your point of view as a programmer, in order to catch a stack overflow problem.

And you can do that yourself. You can use the call mmap, which we haven't talked about, to set permissions on pages at user level.

AUDIENCE: [INAUDIBLE]

CHARLES LEISERSON: No. You can actually handle the error however you want. You can vector into a user to find routine. Now, what are you going to do, I don't know, but you can indeed catch the error yourself and decide that there's some way of recovering, as opposed to crashing and burning. So generally, it's considered poor form for your software to fail, other than in a controlled way.

So you should always have your software exiting or calling error, and it should never segment fault. Most people consider any program that segfaults to be poorly engineered. Is there any case where there's a segfaulting program that is not poorly engineered? I don't know. I think most people, it's probably an understatement. OK?

So this is stack memory. You can build it yourself out of an array and a pointer, or you can I use this the call stack. And when you have something that's using that kind of discipline, it's great, because it's really cheap.

Now in C and C++, and we're going to start moving and talking more about things like C++, we have what's called heap allocation. And in C, the routines are called malloc and free, and C++ provides a new directive and delete. So unlike Java and Python, C and C++ provide no garbage collector. So in other words, you're responsible as a programmer for freeing allocated storage yourself. If you fail to do so, you create a memory leak. So a memory leak means I allocated something, I no longer am in a position to use it, but I failed to free it, so it can't be reused.

And if you keep doing that, especially if you're in a loop, then you have a memory leak. Where if you look at a program with a memory leak, and you look using [? Top ?] or whatever, how big is my program, you watch as it goes and computes and computes, and it keeps getting bigger and bigger and bigger and bigger as it's running, and it just runs away, getting more and more storage.

Also you have to watch out for things like dangling pointers. That is, you deallocated something, but somebody still wanted to use it. Or double-freeing. You have two places in the code where you free the same storage. Very bad to try to free the same storage. Yeah?

AUDIENCE: [INAUDIBLE]

CHARLES LEISERSON: It's basically going to be sitting in the storage structure, which we'll talk about how those are organized in a few minutes-- going to be sitting in the storage structure, but the program that is going to look at that and not recognize that it's in the storage structure, and think that it's something that now all its pointers can be reset. And so you basically clobber all the invariants of the storage structure, and somewhere down the road, you get a segment fault or something. OK? Yeah, John?

AUDIENCE: [UNINTELLIGIBLE] that comes out of that is--

CHARLES LEISERSON: Just a second. Why don't you take this?

AUDIENCE: [INAUDIBLE] One thing that often happens in the real world is when you free something, it adds it to a free, list and if you free something twice, it will add two copies of that pointer to the free list. So later on, when you malloc, malloc might return the same pointer to you twice. So the same memory block might be allocated for two completely different functionalities, and attackers use that to do all sorts of interesting things to your program.

CHARLES LEISERSON: Yeah. OK?

So fortunately, there are some really good-- why don't you keep that down there, and then if you guys want to chime in, you'll have some power to do so. So fortunately, there are now some tools to make that easier to debug these kinds of really nasty bugs. The one we'll be using is Valgrind, and what you do is you just say, Valgrind minus minus leak check equals yes, and then run your program the way you would normally run it. And what it will do is, it will monitor the way that the program is doing allocations and frees, and give you a report as to whether everything that was allocated was free, and whether you tried to free something twice, and so forth.

Now, this works with the existing allocators. If you have your own allocator that you decide you're going to write for a little piece of your code, you say, gee, I'm going to do my own little allocation here, you can actually communicate to Valgrind what that allocator is, and it can check for proper use of that structure as well. So you can see valgrind.org for details. And are a lot of other programs on the market, and out there as open software, such as purify and a variety of others.

So let's talk about heap allocation. I want to start by talking about fixed size allocation. So suppose the world has only one size of objects that you care about. Then it turns out that doing an allocator is really simple, and you could program one yourself.

So here's basically how it works. It's called using a free list. And what you have is, in your array, say, that you're using for storing all of your values, and here they're basically this size, what you do is you have a certain number that are used. That means that the program is using them, not the storage allocator. And what you do is you take advantage of the fact that if it's an unused block of storage, then you can put a pointer in there.

And so what you do, is you string all these unblocked pieces of storage together into a free list. So we point to there, which points to there, which points to there, which points to there, which points nowhere. So you just simply linked together all of the things that are free in that list. And then whenever you need one to allocate an object, what you do is, you grab one off the free list. So you set the value that you're going to want to take off free list to free-- so here x gets a pointer to this object, the one that's at the start of the list. Then you update free with free arrow next, which gives you this guy. So that's the next guy in line. And then you basically return x. And that's all there is to it. You just grab the first element off the free list.

Now, there's a little bit of subtlety in that the thing that you're grabbing off has a garbage pointer in it, because we're not initializing storage. There are some storage allocators that will always set the value that's returned to the user to all zeros, for example. But typically malloc doesn't do that. It basically leaves that as uninitialized storage. And so you can actually go and see, what were the values that used to be stored in there? And so basically, you can end up with a garbage pointer there, but you really don't care about that, because it's no longer on the list.

So you can see, for example, if I then went and I-- well, let's do this, to take a look at what happens when a free an item. So if I want to free an object x, I take some object x here. Let's say it's this guy that's being freed. What I do is, I make it point to the first element of the list. So basically, next equals free, So he basically points there. And then I say free equals x, meaning move the free pointer up to now point to the new beginning of the list. So we're basically pushing and popping things off the front of the list. It's just the stack discipline, except we're using a linked list, rather than using an array. So we push to free, and we pop to allocate.

Now the tricky thing here-- you can sort of see what might go wrong if I then said, oh, let me free this thing again. Suppose you said, now that this is free, suppose you came in and say, oh, I'll free that thing again. Well, now you're going to go through the same code here, and you're going to see, you're going to completely screw up the free list. So that's why you don't want to do a double free. So you can see exactly what's going to go on in the free list implementation.

So this is pretty good and pretty cheap. Allocating and freeing take constant time. That's great, just as with stack pointer. It's got good temporal locality, and the reason is because you're tending to allocate the thing that you freed most recently. The free list operates as a last in, first out data structure. So from a temporal locality point of view, it's really very good. You're always using the most recently freed stuff whenever you allocate, and that means you have very good caching behavior from a temporal locality point of view.

However, it has poor spatial locality, and the reason is due to what's called external fragmentation, where you get your storage distributed across virtual memory. So if you think about this, as you start allocating and freeing in different orders, the order of that free list gets very jumbled up. And as it increases, and you end up with jumping from one thing to another, you basically can end up causing you to use a lot of the page table, because you're basically using a huge piece of it, rather than-- and in particular, the translation look aside buffer can also be a problem. So the translation look aside buffer typically works on a page basis. And if you recently referenced things on a given page, that's very good for the TLB, because it's already got the translation, the virtual address mapping of that page dealt with. So it basically has poor spatial locality, because it's getting all jumbled up. Compared to stack allocation, which is always very nice in terms of its behavior, because you're going across consecutive locations in memory, which means you're always allocating and freeing from the same page, unless you trip over a page boundary.

So here's some ways of mitigating external fragmentation. Of course, this makes the code for free list more complicated. And then you have to weigh, how much am I losing in having a more complicated allocator versus suffering the fragmentation may occur? Yes, question?

AUDIENCE: You specificed before that it was fixed size. Is that why we don't specify anything about the size?

CHARLES LEISERSON: That's right. We're just assuming that it's all whatever the unit size is.

So this is really good when you've created your own data structure, and let's say you're doing nodes in a graph. You know exactly what elements you want in the graph, for each vertex. And now what you want to do is basically be able to give yourself a note of a graph, free a note of a graph, and manage that storage yourself. Yes, question?

AUDIENCE: [INAUDIBLE] allocated across virtual memory [INAUDIBLE] one application?

CHARLES LEISERSON: Why is it allocated across virtual memory if it's all one application? Well, because the size that you may end up needing can end up being large. And I'll show you in just a minute how that happens.

So here's the idea. What you can do is keep a free list per disk page. So disk page, a traditional page is 4,000 bytes. These days they talk about super pages that are megabytes, and who knows what size pages various operating systems will support. I believe the clouds use 4k, as configured right now are using 4k, which leads to some strange anomalies where the TLB can address less stuff than the L3 cache. But so be it.

So basically, they just keep a free list per disk page, and basically always allocate from the free list for the page that's fullest, the page that you've allocated the most of the stuff off of. Rather than for a page that's less empty. So what this means is that when you come to allocate, you have to figure out, which page should I do the allocation off of? But we'll have a free list on each page.

Whenever you free of a block of storage, you have to free it to the page on which the block resides. Don't put into a free list of a different page. You put it into a free list all of that particular page.

Now if a page becomes empty, in other words, there's only the free list on it, there's no actually used elements by the programmer, the virtual memory system will page it out. And since the programmer's never referencing things on that page, it basically costs you nothing. You end up not having to have an entry in the TLB, and you may have written to it, but you will end up allocating things preferentially to the more full pages.

And the basic idea is, it's better to have the use of your pages balance 90/10 than 50/50. So this kind [? of fig ?] configuration is better than that. So why is that? If you have 90% of your storage on one page and only 10% on another-- yeah, let's say we only of two pages in our system, and 90% of the objects are on one page, and 10% of the objects are on the other page, versus if they're 50-50. Then what you can do is look at the probability that two accesses are going to go to the same page. Because if you're going to the same page, then your paging hardware, which is very much like the cache, your TLB, which is very much like a cache-- whenever you get a hit on the same page, that's good, it costs you nothing. If you hit on a different page, that's bad.

So what happens is, the probability that two hit the same page is, well, for the first one, it's, what's the probability that the first one was on this page, versus on the other? So it's going to be on the same page if they're both on this page or they're both on this page. And so that's going to be 0.9 times 0.9 plus 0.1 times 0.1. Whereas if it's 50/50, then the probability that you get hit on the same page is now this 0.5 times 0.5 plus 0.5 times 0.5, which is only 50%. So we get 82% hit rate versus the 50% hit rate. So it's much more likely, if I'm stacking everything out in the extreme, it's better to have everything on one page and nothing on the other. Then I get 100% hit rate, because whenever I'm accessing something, I always am hitting on the page I had before.

So this kind of heuristic helps it so that you're swinging your accesses to using fewer pages, and the fewer pages, the less likely it is that you're going to stress your virtual memory paging, disk swapping, you're going to stress your TLB and what have you. Good for that? We're going to get deeper into this, so.

So that's basically fixed size allocation. It's really nice, because you can basically use a very simple free list. But of course what most people need and want is variable sized allocation. The most popular way of doing variable sized allocation is a scheme called binned free lists, and of that, then there are a zillion variations on it. But mostly, people use binned free lists.

The idea is, we're going to leverage the efficiency of normal free lists. Normal free lists I can allocate and free in just one operation, in just a constant amount of time I make. And what we're going to do is accept some internal fragmentation. So by internal fragmentation I mean the following. When somebody asks for something of a given size, what we're going to do is actually give them something which is a little bit bigger, maybe. And so we'll waste some.

And in particular, what we'll do is, we'll organize our system so that all the blocks that we ever give out are always a size the power of two. In practice, you don't always give out exactly a power of two, because then you get cache alignment problems. But this is the basic scheme, is to give out things are powers of two. And that way, if somebody asks for something that has 13 bytes, you give them 16 bytes. For somebody who asked for 65 bytes, have to give them 128 bytes. So you might waste up to as much as a factor of two.

But now, for a given that range of values, there's only a logarithmic number of different sizes that we have to give out, because we're only giving them out in powers of two.

So here's the basic allocation scheme to allocate x bytes. So what I do is, if I'm allocating x bytes, what I do is, I look in the bin ceiling of log x, because that's where everything of size two to the ceiling of log x is, which is basically rounding x up to the next power of two. If it's non-empty, I just return a block as if it were an ordinary free list. And what I'll be doing is returning a block which is at most twice the size of the requested block.

But that's generally OK. Somebody wants at least that many bytes. The fact that it's a few more bytes won't matter to them. They don't know that there's more bytes hidden there. They only take advantage of the fact that there are many bytes as I've asked for.

Otherwise, suppose that that bin is empty. There's nothing in that bin at this time. Well, then what you do is you start looking to find a block in the next larger non-empty bin, and you split it up. So for example here, suppose that the request is x equals 3. I have to look in bin 2, which is blocks of size 4. Uh oh, bin 2 is empty. So what I do is, I start going down, looking in the next bin. Oops, that's empty too, that's empty-- and then finally, I get to one where there is a block in it.

So what I do then is I dice this up. I cut it in half, and then cut that one in half, until I get a piece that's exactly the size I want to return. OK? And then I distribute all of these guys up to the other bins, like this.

So I return one of this size, and I populate these other guys with one block each of the appropriate size. And as you can see, they're basically growing in a geometric fashion. And then I distribute that bin.

Now there's a caveat here. Suppose that no larger blocks exist in my storage structure. Well, then I have to ask the operating system for more bytes, to allocate x more bytes of virtual memory. Now in practice what you do is you don't actually ask for x, since you never ask the operating system for small values. You always ask the operating system for big values. Give me another page worth, or give me another five pages, or something. Because that's an expensive call to the operating system, to ask it to allocate more storage.

Let me explain how that works. And this gets back to the question you had before, which is, where's the stuff coming from anyway? What the operating system is allocating is actually not physical pages, but rather parts of virtual memory. So we looked at this before, about how virtual memory is laid out for a typical program.

So what we do is, on the very high addresses, we have the stack, and the stack grows downwards. This is the call stack. At the low address, we typically start out with what's called the text segment, which holds your code. Then there's a region called the data segment, and this consists of data that needs to be initialized. So the compiler typically distinguishes between data that needs initialization and data which is going to be zero, because data that needs initialization, it writes into the binary executable file and stores that on disk. So when you load in your program, it just loads as part of the text here, it loads that data segment, saying what all of the storage is.

The next segment is, for historical reasons, called the BSS segment. And what BSS consists of is all the variables that at the start of the program should be initialized to 0. It doesn't bother storing these in the file. Why not? Because it's easy enough to just bring it in, understand what that distance there is, and then do a memset of this whole region with zero, to zero it out, rather than trying to read in a whole bunch of zeroes. An obvious kind of data compression.

Then heap storage is allocated in the space above the BSS segment. So the idea is, there's a part of heap which has already been allocated stuff. And whenever you need more heap storage, you ask the operating system-- the operating system moves just as if it were a stack, moves the line for where heap is to allocate, to give you more pages of heap storage that you can now allocate out of. And what it's doing is allocating pieces of virtual memory.

Now, here's a good mind question, which I think gets directly to your point. So think about this. We have a 64-bit address space. If I try to write at the rate of 4 billion bytes per second, that's pretty fast, because that's the speed at which the registers get written. Not the speed that memory gets written. So really, it's something like 4 billion bytes per second. It takes over a century to write all of virtual memory.

So as a practical matter, that means I never run out of stack space. I never run out of heap space. They're never going to cross, no matter how much I'm allocating. Even if I allocate on every cycle, there's not going to be a program-- unless you're going to set a program running for a millennium sort of thing, you're not going to see that. Most programs run for a few minutes. Some may run for days. Some may run for years. Very few do people write to run for a century.

So why not just allocate out of free virtual memory and never bother freeing? Why am I worried about freeing stuff all the time? Why not just allocate it? It's virtual memory. It's virtual. I just allocate more storage as I need it. Yeah?

AUDIENCE: [INAUDIBLE] memory?

CHARLES LEISERSON: So it is tied to real memory, and that's part of the answer. It is tied to real memory, because anything that you write must find itself a place on disk. Because it's going to be written to pages, and those pages are going to be paged and then written to what's called the swap space on disk, which allows you to put things there.

Even so, though, writing to disk-- it's still not going to take very long. I mean, you're never going to be able to use up disk, because disk takes so long to write for. But that is part of the answer.

AUDIENCE: So you need to basically use [INAUDIBLE] software? And so--

CHARLES LEISERSON: And the page table.

AUDIENCE: [UNINTELLIGIBLE] on the disk.

CHARLES LEISERSON: So the issue is, this is where the external fragmentation would be horrendous. If you just kept writing across memory, and you only had a couple of words written on every page that were actually used, the rest is just garbage, then what you're saying is that in order to access a given datum, I'm not going to have it in memory with it a lot of other stuff. What I'm going to have in memory with it is going to be all garbage, this stuff that I would have freed. And so therefore, you end up having your program distributed across a large region of virtual memory, and that means as we're doing the paging, et cetera, we're getting very poor page locality as I'm accessing things.

And so I have a finite number of pages that I fit in memory, and now I may be blowing that out, and every access, rather than going to something nearby, is instead going to disk. And so that can cause disk thrashing, where every access causes a disk access. So how fast are disk accesses? Order of magnitude? How fast are disk accesses? What's a disk access cost? Fast disk?

AUDIENCE: [INAUDIBLE]

CHARLES LEISERSON: No, it's slower than that. Disk access. So good memory access, DRAM memory is, like, 60 nanoseconds for DRAM access. It's like 60 nanoseconds. So the processor is going somewhere around, these days, between 3 and 4 gigahertz. So memory is like 200 processor cycles away.

How fast is disk? An ordinary disk, not a solid state device. Ordinary disk.

10 milliseconds is a good ballpark to give. That's a good number to know. 10 milliseconds, not 10 microseconds, not 10 nanoseconds. 10 milliseconds. So you are, what? 6 to 7 orders of magnitude slower if you go to disk. Really slow. So one of the things in performance optimization-- we're focusing a lot on the CPU-- is often the case for database things. How do you minimize those disk accesses? And all the things that we're learning about, performance for CPUs, can be applied to disk accesses, as well.

So the idea here is we want to avoid-- that's why we don't want to have our stuff strewn across a whole bunch of pages where we only have a fixed amount of internal memory, because then we'll end up paging all the time. Every access will cause us to do a disk access, because it won't be very likely that we're getting stuff on the same page. So getting back, for example, to the fixed allocation, that's why it's a good idea to try to keep those pages full, so that we have a small thing.

So the goal of most storage allocators is therefore to use as little virtual memory as possible, and try to keep the used portions very compact. Because the more compact it is, the more likely it is that when you access one, you'll be able to access something nearby without causing a page fault. So that's the goal of almost all storage allocators. Try to keep things compact, try to keep things tight and in very little virtual memory space, even though virtual memory is huge.

Now, that doesn't mean that everything has to be contiguous. For example, in this organization as I showed before, actually, I have it just on the previous slide, so why don't I go back there-- these are very far away in memory, but there's only a couple of them, and they're tight and local in the regions that they're operating in. What I don't want to do is have it where I have just little stuff. It's OK to go into the middle of virtual memory and have a whole bunch of consecutive pages full of data. That's fine. What's bad is having just a little bit of data, a little bit of data, and a little bit of data sort of sprinkled all over. That's bad. But as long as I can put a whole bunch of data nearby, it doesn't matter where in virtual memory I put it, particularly. So the idea is, let's try to use as little virtual memory as possible. Any questions about that? OK.

So how do we do that? So the first thing is, binned free list turns out to do a really good job of using relatively little memory. So suppose that the user program is using, at most, m memory at any point during the execution. Maybe allocating, maybe freeing. But if I took the high watermark of everything he says that he needs, we call that m.

Then it turns out that if you use a bin free list allocator, the amount of virtual memory consumed is at most m times log n. Order m times log n. And why is that? Here's kind of a hand-wavy proof sketch. So whenever I make a request for a block of size x, it may consume at most twice x storage that's actually in use.

And so it turns out that the total amount of memory devoted to blocks of size 2 to the k is therefore at most 2m. So for any given size, I might, at any given point in time, have the program using all blocks of that given size. Let's say size 2 to the k.

And since there are a logarithmic number of those bins, each bin of which could have a size at most m, the total is order m log m.

Now it turns out that there's actually a stronger property. In fact, you can show that binned free list is order one competitive. Who knows the term "competitive" in this sense? The algorithmic meaning of "competitive"? So what it means is, it's at most a constant factor slower than the optimal omniscient algorithm that could figure out exactly what's going on in every step. So you're within a constant factor competitive with the optimal allocator, assuming that the optimal allocator does not do what's called coalescing. If it simply is dividing up things, then it can subdivide, but it can never glue things back together again. It turns out that in fact, binned free list is really much better than that.

So let's talk a bit about coalescing, because many, many memory allocators use coalescing. So the idea is, I'm concerned about the case where I end up with a whole bunch of little small objects, and yet the allocations now come from big chunks. And so I have to go and use more of my virtual memory, or maybe I can glue together those little pieces into a big chunk, and not have to go off to memory. So that's what coalescing is all about.

So the idea is, if two things are adjacent, let me glue them together. And there are really clever systems for doing that, including a very famous one called the buddy system. But using any of these mechanisms for coalescing, you always have to trade off between the simplicity of not coalescing. Not coalescing so stick it in the free list, I'm done. Coalescing is, stick it in the free list, oh, am I next to somebody I can merge with? Now let me merge. So you always have to weigh, is that really going to be an advantageous thing?

It turns out there are no good theoretical bounds that I'm aware that prove the effectiveness of coalescing. People have not figured out how to analyze this, or what an appropriate model is for understanding whether coalescing is a good idea. However, it does seem to work in practice for some things, and I think the reason for that-- and this is just my own take on it-- is that storage tends to be deallocated as a stack. So you tend to go into a region of a program where you do a lot of heap allocation-- let's say I'm building up a graph, and I keep doing a whole bunch of mallocs. And when I'm done with the graph, what do I do? I free it all. So I tend to have these batch allocations and then deallocations, and they tend to work often as a stack, often in a batch. And that means that when I do that freeing, I can sometimes take advantage of that and do coalescing.

In some systems, people have what are called memory pools, where you have a region where when you allocate out of it, you know you're allocating out of your own particular region specifically so that you can do coalescing of things in that region. And so you may have multiple allocators, and you'll call the allocator on the region, say, for a graph, because let's imagine you're creating two graphs, and one of them you're going to keep around, and the other you're going to dump. Well, if I create them at the same time, the storage gets all intermixed, I'll deallocate one graph. The other one will be, I'll have fragmentation all over. These little spots appeared where the other graph used to be-- not so good.

But if I say, use this allocator for this graph, use that allocator for that graph, then when I'm done and I'm allocating those out of two different regions of memory, that when I'm done with the first graph and deallocate everything, now I've created a nice big piece of contiguous storage. So people will sometimes play with different memory pools for allocators, and C++ has a fairly rich set of libraries and so forth for that, to allow you to do that kind of thing.

So let's go on and talk about garbage collection, because that's something you've probably seen in Java and Python, and what's going on under the covers there. Because even though C and C++ do not have garbage collection, you can actually implement your own garbage collector. It's not that hard, except for some details. It's only hard when you want it to perform well.

So the idea is to free the programmer from freeing objects. The downside, by the way, is when you free the programmer from freeing objects, they tend to create lots of garbage. And so what happens in Java programs, for example, is you'll have these-- I remember once having a Java program where, when it was sitting, waiting for user input, it would periodically go, garbage collect. It was just sitting there, and then it would go, garbage collect. We weren't doing anything. It was just periodically, the way they had written the code, they were somehow generating garbage, doing allocations and frees, when you were sitting around doing nothing. And so it would basically use up all of the space and then do a garbage collection, use up all the space and do a garbage collection. Naturally that's kind of a slow system.

And so even in something like Java, where you have a garbage collector, you do have to worry about whether you're creating garbage or not, and you can generate faster code by not creating as much garbage, or by finding ways of recycling the garbage yourself. So even in Java, it's worthwhile saying, give me a big chunk of memory, let me manage it myself, now let me free it. I don't have to free it, but when I'm done with it, I'll be done with it. But in the meantime, let me manage it myself so that I don't have it continually generating more garbage.

So the garbage collector can be built in, or you can do it yourself. And mostly in this class, we do stuff ourselves, so we know how they work. So that even when you're doing it with built in, you know what's going on under the covers.

So here's some terminology. The roots in garbage collection are the objects that are directly accessible by the program. And those typically are things like globals, stacked variables, and so forth. Things that the program can directly address by name. The live objects are reachable from the roots by following pointers, and the debt objects are the ones that the programmer can never get to again. And so what we're interested in doing in garbage collection is finding all the dead objects and recycling them so we can use them again, so that they don't end up taking up space in our virtual memory and slowing us down by making our virtual memory space continue to grow and grow and grow across many disk pages.

So one of the questions, in order for one to be able to do garbage collection, is how do I know where pointers are in objects? How do I know what's accessible? And so typically, in order to do that, you have to have a very strong idea of where is a pointer and where is data? Because you don't want to follow data as if it's a pointer, you want to know which regions of a given block of memory are pointers. And so strong typing helps a lot with that, which is why mostly people do garbage collection in strongly typed languages, because then you can actually identify it.

It also requires you to prohibit pointer arithmetic. So the ability to take a pointer and then index off that pointer means I can get to some piece of storage that I can't access directly. And so typically in these kinds of systems, pointer arithmetic is illegal. So you can do array indexing, as long as you treat the array as a single object. You can't treat an array as the component pieces. You have to treat it as a single object. So pointer arithmetic.

Well, you folks, who's familiar with pointer arithmetic? At this point, almost all of you have done stuff with pointer arithmetic, if your code is any good. Almost all of you have done adding to pointers in order to move through an array or what have you as being somewhat cheaper than doing an array index every time. It saves you an extra memory reference, typically. So that's one of their restrictions when you want to do garbage collection in a general setting.

Now probably the simplest and most useful one as a programmer to use for garbage collection is what's called reference counting. I know many, many programs where what you do is you do reference counting because you can. But as you'll see, it has a limitation.

So the idea in reference counting is to keep a count of the number of pointers referencing each object. So here we have a whole bunch of different objects of different sizes, and whenever there's a pointer going to the object, I add one to the reference count shown in green at the top here. I hope I got the count right. Whenever the count drops to zero, you free the object. That means nobody's pointing to this object [UNINTELLIGIBLE]. Let's free it. So let's see how that goes.

So suppose that, for example, that pointer got moved to over here. Went away from there, got moved to there. Well, now there are two fields that have to be updated. The one that we took the pointer away from and the one that we put it to.

So we update those. Oops, that guy went to zero. That means this stuff is garbage. But I can't just immediately go and collect that piece of storage. Why not?

AUDIENCE: To save the value of the pointer, and [INAUDIBLE] pointer later?

CHARLES LEISERSON: Yeah, to save the value of the pointer--

AUDIENCE: To save the value of the pointer and then use it again later. The pointer isn't there, but you know where--

CHARLES LEISERSON: Not quite, not quite. Yes?

AUDIENCE: [INAUDIBLE] the reference counts for [INAUDIBLE], and also--

CHARLES LEISERSON: Yeah. If you're going to deallocate this guy, he's got pointers. They have to follow his pointers and see which ones need to be deallocated. Good.

So this guy now, we realize, is garbage. But before we can just free him, I've got to deallocate these pointers. So this guy also got decremented, but he didn't. So this guy, we have to decrement these two fields, because we took away those pointers because we're going to garbage collect that. And when we do that, it turns out this thing turns to garbage, as well. So when you do reference counting, you have to make sure that when you decrement things and decide you're going to free something, you also then go through and free the things, and you continue that process until everything that has been deallocated can be deallocated. And then this becomes recyclable, put back into the storage, into your free list or whatever scheme you have.

So this is a very simple way, when you're implementing data structures, of building your own dynamic storage. Because you basically can allocate stuff, you can free it, and whenever the reference count goes to 0, boom. You just have it deallocated. OK? But you can program that yourself [INAUDIBLE].

Now there's a problem with the scheme, though. What's the problem? Cycles. Yeah, cycles are the problem for reference counting. So here's an example. The problem is that a cycle is never garbage collected. Let's take a look at this structure here. So everybody's got a pointer from somewhere, but it turns out that there's a cycle here where, notice that they are pointing to each other, we even have a guy coming off here, but nothing points into the cycle.

So there's no way that, from the roots, I can ever access these four guys. They're garbage, but my reference counting scheme will never find it, because no one's ever going to decrement one of those counters here. So those are all garbage, but how would I know? And that's basically the disadvantage of reference counting. And uncollected garbage, as we all know, stinks. So we don't want that situation to occur.

So nevertheless, reference counting is great if you've got an acyclic structure. And the number of times acyclic structures come up where reference counting is the simple and easy way to do the storage management is huge when you get into doing any kind of interesting programming. Lots and lots of opportunity to use reference counting approach. So questions about reference counting? You have a question? Yeah?

AUDIENCE: So this means you actually have to do this [UNINTELLIGIBLE] ever do a [INAUDIBLE]? You can, like, save the work--

CHARLES LEISERSON: That's right. Whenever you move a pointer, if you're going to move a pointer, you first decrement the counter on the place you're moving it from, and then garbage collect if need be, and then you move the pointer-- that's typically what you do-- you move the pointer and increment the count at that location, which typically doesn't require any extra work to decrement, which is the trick thing. Sorry?

AUDIENCE: There's no good way to do this in idle time. Like I'll have some idle time later, let me do it--

CHARLES So that's what you use true garbage collection for Yeah?

LEISERSON:

AUDIENCE: You could actually, there's techniques where you can defer the work of incrementing it. Like you basically put the increment and decrement operations that you'd like to do into a buffer, and then you can do them later. But it's sort of the same.

CHARLES LEISERSON: Yeah, you could log. So there are schemes where you log that, oh, I decremented this, better check it later, type thing. Or I decrement it to 0. Let me just put it in there, and I'll go actually do the stuff in batch mode at some later time. Yep. So programming is fun, because you can be clever with schemes like that. And that could be quite a reasonable thing to do, which is wait until you're waiting on the user for input, and at that point, collect your garbage.

Now, the thing you have to risk is, what happens if that takes a long time to get there? And then there's the complexity of your algorithms and so forth. So you get into a lot of issues of software maintenance and so forth. But that can be a very effective thing, is take it out of the critical path of the actual complication you're doing, if that's mission critical, and put it in some part of the computation where it isn't mission critical. Very good idea.

I'm sorry?

AUDIENCE: For thread safety. Like if you have shared reference accounts, like multiple threads updating the counts, you don't want to have them all in the same count at the same time. So they fill it like a thread local buffer, and--

CHARLES LEISERSON: I see what you're saying. Yes. So we're going to get into that as we deal with multi-threading and so forth, that if you've got two guys that are operating on the same data structure, you have to be very careful about the consistency, and that one doesn't do something while the other is doing something else, and so forth. So for that reason, it can be a good idea to postpone operation.

So general garbage collection is based on a graph abstraction. So let's move into the graph world. So I can view objects and pointers as forming a directed graph G equals VE, where V are the objects and E are the pointers from one object to another. The live objects are reachable from the roots. And now the thing is that finding all of the live objects is then like doing a graph search. The common ways of doing a graph search are depth for search and breadth-first search. It turns out we're going to use breadth-first search for reasons that I'll show you.

So the idea in breadth-first search, to find all the objects. So the idea is, what we're going to do is go through and find, where are all the objects in my system that are reachable from the root? And then anything that I didn't find, ah. That must be garbage.

So here's the idea. What we're going to use is FIFO queue called Q. So that's a first in, first out. It has a head and a tail. And the mechanism is very much like a stack. Whenever I need to enqueue something, I put it on the tail end and increment the tail pointer. Whenever I need to take something off the queue, I increment the head pointer. So I'm always doing head and tail, pushing on the tail and pulling off the head. Enqueue and dequeue.

So the way that this works is, I basically go through all my objects, and if they're a route, I put a mark on it. And I enqueue-- this is pseudocode, by the way, so don't hold me to my C programming standard here. And I market, and then I enqueue it on the queue Q. And otherwise, I just simply mark it as zero.

So being marked means whether we've discovered it's reachable from a route. So to begin with, what we're going to do is mark everything that's reachable from the routes, and we're going to enqueue them on the FIFO. And everything else, we're going to mark as unreachable, for now. And now what we're going to do is see which ones we discover.

So what we do is, while the queue is non-empty, we're going to take something off of the queue and then look at all of its neighbors. So all the edges UV, look at all the things that go from U to V And if it's not marked, then we're going to mark it and put it on the queue. That's it. If it's already marked, we don't have to do anything with it, but if it's unmarked, we'll mark it and do it. And then in the end, everything that's marked is going to be something that is live, and everything that is unmarked is going to be things that are dead. But it's going to be actually even sexier and more clever than that, as you'll see.

So here's an example. We start out with my queue being empty, with head and tail pointing to the same place. And what we do, is we go through and we first-- in this case, I have just one route R, right here. So we basically mark R and put it on, and that's how we get started. And everything else is unmarked up here. So I'm going to mark with green up here, and I'm going to show what's on the queue with the green down here, and dark green when I pop something off.

So the first thing I do is I say, OK. Let me dequeue something. And in this case, I'm dequeueing R, because it's the only thing on there. And what I do now is I visit all the neighbors. So I visit b, so I put that on, mark it and put that on, and then I visit c, so I visit and put that on. Then I'm done with R. I visited all its neighbors. So then I go and I dequeue B, and now visit all its neighbors. Well, in [? visit ?] c, there's nothing to be done there, because c is already marked. So I don't enqueue it again. I already have observed that. And there's nothing else adjacent to b, so basically, I then go on and dequeue c. So I have c here, and now we go visit its neighbours, and the neighbors are d and e. So we enqueue d, enqueue e, and now we are done dequeuing all his neighbors and marking them, so now we pop off d. And now d has no neighbors, so there's nothing to be done. And now I dequeue e, and basically, e has one neighbor f, so we mark f. Then we dequeue f, we mark g, then we dequeue g, and g has-- the only neighbor is e. And now my queue is empty, and so I'm done. And now you'll see, what did I do? I marked all the things that were reachable from r, and everything else is garbage.

Now, that's only half the story. So I've marked them, but more relevantly, it turns out for garbage collection, is look at what I've got in my queue. What do I have in my queue? I've got all the live objects, got put in the queue as we were going along. All the live objects are put in the queue. And not only that, but notice they're all adjacent to each other in the queue. They're compact in the queue.

And so the idea is, we're going to take advantage of that property. That when I do breadth-first search, that the queue, when I'm done, I put every object in there. So all the live vertices are placed in contiguous storage in queue.

So this is the principle behind the copying garbage collector, which is one of the more popular garbage collectors that's used. The idea is that as I'm executing, I'm going to have some live objects and some dead objects, and I'm going to have some place that I'm doing my next allocation at. And as I go through, I allocate more things in what's called the from space, and I keep allocating, and I may do a deallocation, and maybe some more allocation, and maybe another deallocation, and some more allocation, and eventually I run out of my memory that I've assigned for my heap storage.

So when I get to this point, now what I do is I'm going to do breadth-first search on the from space. And what I'm going to do is I'm going to copy using the live storage here, using [UNINTELLIGIBLE] to a TO space, where the TO space implements the FIFO queue. So here's the TO space, and when I go through and I walk all that, I've now copied all of the values, all of the objects down to this space over here. And now I have this amount of storage left. I make the two space exactly the same size as the from space. I have to make it at least that big, because I know that it may be that everything here is alive. Maybe nothing got deallocated. So I have to make sure that the TO space is large enough to handle everything that might be alive there, and it may be everything. So the TO space is generally allocated to be exactly the same size as the FROM space. So basically, the way I compacted is I copied things down using the breadth-first search algorithm.

Now there's one problem with this, and that is that we have pointers in these objects. And when I copy it down here, how did I make sure that all the pointers now point to the new locations? So here's how you do that. So since the FROM address of the object is not generally equal to the TO address, pointers must be updated. So I have to know, of course, where the pointers are. So the idea is that when an object is copied to the TO space, what we're going to do in FROM space is store a forwarding pointer in FROM space saying, I copy you to here. So that whenever I look at something in the FROM space that's already been copied, I can say, oh, here's where it's been copied to. It's been copied to this region in the TO space.

When an object is removed from the FIFO queue in the TO space, we're going to update all its pointers. I'm going to give you an example of this. So if you think about it, when I remove something from the FIFO queue, at that point, I know all of its adjacent vertices have been placed into the queue, into the TO space. And so at that point, I know where all the final destinations of pointers are.

So here's an example. So suppose that I'm doing the copy of these guys, and I've got all the pointers around here, and now some of these guys have already made it into the FIFO queue. Let's say this guy has made it in, because I've got forwarding pointer to there, and this guy has made it in, so I've got forwarding pointer to there. And this is what my queue currently holds.

So when I remove an item from the queue, I then look at this guy, and what I do is first of all, I visit all of the adjacent neighbors. So I visit this guy and I say, oh, he's already been copied, because I have a forwarding pointer. So nothing to be done there. But this guy, he hasn't been copied yet. So I have to make sure that that guy is copied by enqueuing the adjacent vertices.

So I copy my neighbor to there. And part of copying is, if I'm actually making a copy, I'm going to have a pointer from here. The pointer now has to point to this guy over here, because that's how I'm copying. Otherwise it isn't a copy. So in the copy, this pointer is the same as that one, is pointing to the same storage location, namely this place over here. So I do that for all my adjacent vertices. People follow so far?

Now I've copied all of the neighbors for this guy. Now this guy, all of his neighbors are in their final destinations in the TO space. So that means I can now update these pointers to be the pointers in the TO space. So I do that as follows. Oh, first of all, I forgot to place the forwarding pointer here to indicate that he's been copied. That's how I'm doing the marks. Sorry about that.

So I copied the values, and that's just a straight copy, because all the pointer values are now pointing into this space. Then I put a forwarding pointer in. Now I'm ready to do the update of the pointers here.

So I update the pointers to the removed item. So here's one pointer. I get rid of him there and I make him point to the forwarding thing. He used to point to here, now he's pointing to here. This one used to point to here, I want to make him point to there. There we go. So now he's pointing here. And now this guy here now has his pointers to his final destinations. And so the invariance that I'm maintaining is that everything before the head of the queue all have their final pointers in place in the TO space. Question?

AUDIENCE: So do you know where to put the pointers-- how do [UNINTELLIGIBLE] pointers go from the TO space to the TO space so that they're actually getting to their final destination? So my question is, [UNINTELLIGIBLE] by following [UNINTELLIGIBLE]. But how do you know what pointer to follow? How do you know that [INAUDIBLE]?

CHARLES LEISERSON: If I have a pointer to something in the FROM space, then I update it to be the forwarding pointer in the TO space.

AUDIENCE: Yeah, the question is, how do you know that [INAUDIBLE] it's being pointed to [? the ?] actually in the TO space? Can you check, is this in the TO space?

CHARLES LEISERSON: Well, yeah. I mean, you know what the range of the two spaces is, so you could check that. OK. Question?

AUDIENCE: So the reason you would be using [? BFS ?] instead of [? DFS ?] is because [INAUDIBLE].

CHARLES LEISERSON: The reason you're using BFS is because it has this nice property that the queue represents the copying of all the values. If I use DFS, I need a stack to do it, and then I'm overwriting the locations where the vertices would be stored in the data structure. By using DFS, there's this just clever thing that the queue, when I'm done with DFS, represents all of the visited vertices. It's very clever. Wish I'd thought of that. But I was, I think, only 10 years old or something when they invented this, and I wasn't that precocious. OK? So very, very clever.

So this is the scheme behind the Minsky, Fenichel, Yochelson garbage collector. So Marvin Minsky, you may have heard of, was one of the inventors of a scheme like this. One reason he's so famous.

Good. So this is really nice, because it takes me linear time to copy and update all of the vertices. It's just one pass of breadth-first search. Very fast to copy and update all the vertices. Sometimes, by the way, people call these forwarding pointers, they call them broken hearts. You'll see that in the literature. They call them a broken heart if it's-- I don't quite remember why that's the-- it's like, I went there, oh, you're not there, it's over there-- it's like, I don't know.

Now, the last thing you have to do-- and this is something that's not talked about a lot in the literature-- is how do you manage these from and to spaces in virtual memory? So here's the strategy that works well. So basically, typically what I'm doing is after I've done a copy, I've emptied everything out of the FROM space, and I've got a portion that's used in the TO space, and then an unused portion that I'm going to be able to do allocations out of for the next pass around.

So what I do at that point is I make the old TO become the new FROM. So here's the new FROM. This is available space. And now what I do is I extend this space, if necessary, to make the new heap space that I'm going to allocate out of equal in size to the used space. And the reason for doing this is to amortize the cost of going through all of the storage, so that the cost of going the garbage question can be amortized against what's used. If I didn't extend this, I might have only freed up a tiny little bit of space, and now when I run out of that space, I'll run the garbage collector going over all the elements, but I'll only have gotten a few elements [INAUDIBLE]. So what I want to do is make sure that I've done at least a certain number of allocations proportional to the number that I'm going to have to walk over in order to finish it. So that's why you make this be the same size.

Now when the space runs out, the new TO is allocated with the same size as FROM. So if you can allocate it back in the old FROM space here, that's great. If there's room here to allocate it at the front, super. That's where you allocate it. But as in this example, the FROM space here is bigger than the space that goes up to here. So therefore what I do is I allocate it at the end, and then I go through and copy to here.

Now if this ended up being used, then the next time around, I would be able to reuse this space at the front for the next time that I flip the garbage collection. The idea is, if things will fit as early in memory as possible, that's where I put them. But if they don't fit, I put them later. And you can prove, if you do this, that the virtual memory space used is order 1 times the optimal, where you're getting nearly the best virtual memory space used compared to the amount of items that are in use by the user at any time. So it keeps everything sort of compact towards the front end here.

So in this case, I had to allocate to here. If the use had been very small, I could have allocated just a very small amount for the heap. When I was done, the new TO space would have fit at the beginning. I would have allocated it at the beginning. And it's a nice little exercise, for those of you who are analytically inclined, to figure out what the constant is here, because it's a little bit of a game to figure out what the amortized constant, for those who have some algorithmic energy and our confident about working in this kind of thing instead of studying for the quiz.

So dynamic storage allocation. Hugely interesting area. Lots is known and lots is unknown. For example, I mentioned coalescing. We really don't understand coalescing. Disaster people have been doing it for 50 years or more. We do not understand coalescing.

There are a lot of strategies that people have studied, things like the buddy system, I mentioned. There are other types of garbage collectors called mark and sweep garbage collection. Those are typically inferior to the copying garbage collection, because they leave you with a memory that isn't compact. And so you end up using more and more of your virtual memory, because you're leaving things in place. On the other hand, you leave things in place so there's no need to move pointers. So that can be a valuable thing.

There's things called generational garbage collection. Generational garbage collectors realize that things that are allocated typically are freed very soon thereafter, and so therefore, if you especially take care of the things that are allocated and guess that they're going to be used again, and that things that haven't been deallocated for a long time are very unlikely to be allocated, you can get better performance out of a garbage collector.

There's real time garbage collection. So this is what I've explained, is what's called a stop and copy garbage collector. Meaning that when you run out of storage, oops! Stop the world, run the garbage collector. There are real time garbage collectors which, whenever you do an allocation, it goes and it does a few cycles of garbage collection, and is such that you can always keep up with the allocations and deallocations going on. By doing a few cycles, you eventually get to garbage collect everything, but you do it intermixed, so that you're slowing everything down by a constant amount, rather than running and suddenly, OK, your robot arm is going like this, oops, garbage collect, garbage collect, garbage collect, [CREAKING SOUND]. So you can have your robot arm move smoothly, because it's paying the price all the way. It's not stopping and going.

There is multi-threaded storage allocation. What happens when you have a multi-threaded environment where you have many processors that are all allocating out of the same pool? How do they organize it? You can have the interesting things going there where things are allocated on one processor, and then get deallocated on another. And if you're not careful, you have something like a memory leak going on, which is called memory drift, where the memory gets allocated on one, and one guy's busy creating it, and it keeps getting deallocated on the other, but never gets recycled back to the guy who needs it. And so how is it that you figure out to make sure that memory drift doesn't end up having you grow your memory? In fact, many, many garbage collectors that are out there have this memory drift problem for multi-threaded storage allocation.

There's also parallel garbage collection. How do you, in a parallel environment, have the garbage collector going on as a separate process while you have other things going on at the same time? So a huge, huge, huge area of storage. And for our next lab, lab 3, you'll be building your own storage allocator. So you have a lot of fun with this. [? Reid ?] says this is his favorite lab.

Great. Any questions? OK. Good luck on the quiz next time.