JULIAN SHUN: Good afternoon, everyone. So today we're going to talk about storage allocation. This is a continuation from last lecture where we talked about serial storage allocation. Today we'll also talk a little bit more about serial allocation. But then I'll talk more about parallel allocation and also garbage collection.

So I want to just do a review of some memory allocation primitives. So recall that you can use malloc to allocate memory from the heap. And if you call malloc with the size of s, it's going to allocate and return a pointer to a block of memory containing at least s bytes. So you might actually get more than s bytes, even though you asked for s bytes. But it's guaranteed to give you at least s bytes.

The return values avoid star, but good programming practice is to typecast this pointer to whatever type you're using this memory for when you receive this from the malloc call. There's also aligned allocation.

So you can do aligned allocation with memalign, which takes two arguments, a size a as well as a size s. And a has to be an exact power of 2, and it's going to allocate and return a pointer to a block of memory again containing at least s bytes. But this time this memory is going to be aligned to a multiple of a, so the address is going to be a multiple of a, where this memory block starts.

So does anyone know why we might want to do an aligned memory allocation? Yeah?

STUDENT: [INAUDIBLE]

JULIAN SHUN: Yeah, so one reason is that you can align memories so that they're aligned to cache lines, so that when you access an object that fits within the cache line, it's not going to cross two cache lines. And you'll only get one cache axis instead of two. So one reason is that you want to align the memory to cache lines to reduce the number of cache misses.

You get another reason is that the vectorization operators also require you to have memory addresses that are aligned to some power of 2. So if you align your memory allocation with memalign, then that's also good for the vector units. We also talked about deallocations. You can free memory back to the heap with the free function.

So if you pass at a point of p to some block of memory, it's going to deallocate this block and return it to the storage allocator. And we also talked about some anomalies of freeing. So what is it called when you fail to free some memory that you allocated? Yes?

Yeah, so If you fail to freeze something that you allocated, that's called a memory leak. And this can cause your program to use more and more memory. And eventually your program is going to use up all the memory on your machine, and it's going to crash.

We also talked about freeing something more than once. Does anyone remember what that's called? Yeah? Yeah, so that's called double freeing. Double freeing is when you free something more than once. And the behavior is going to be undefined.

You might get a seg fault immediately, or you'll free something that was allocated for some other purpose. And then later down the road your program is going to have some unexpected behavior. OK.

I also want to talk about m map. So m map is a system call. And usually m map is used to treat some file on disk as part of memory, so that when you write to that memory region, it also backs it up on disk. In this context here, I'm actually using m map to allocate virtual memory without having any backing file. So

So our map has a whole bunch of parameters here. The second to the last parameter indicates the file I want to map, and if I pass a negative 1, that means there's no backing file. So I'm just using this to allocate some virtual memory.

The first argument is where I want to allocate it. And 0 means that I don't care. The size in terms of number of bytes has how much memory I want to allocate. Then there's also permissions. So here it says I can read and write this memory region. s private means that this memory region is private to the process that's allocating it. And then map anon means that there is no name associated with this memory region.

And then as I said, negative 1 means that there's no backing file. And the last parameter is just 0 if there's no backing file. Normally it would be an offset into the file that you're trying to map.

But here there's no backing file.

And what m map does is it finds a contiguous unused region in the address space of the application that's large enough to hold size bytes. And then it updates the page table so that it now contains an entry for the pages that you allocated. And then it creates a necessary virtual memory management structures within the operating system to make it so that users accesses to this area are legal, and accesses won't result in a seg fault.

If you try to access some region of memory without using-- without having OS set these parameters, then you might get a set fault because the program might not have permission to access that area. But m map is going to make sure that the user can access this area of virtual memory. And m map is a system call, whereas malloc, which we talked about last time, is a library call. So these are two different things. And malloc actually uses m map under the hood to get more memory from the operating system.

So let's look at some properties of m map. So m map is lazy. So when you request a certain amount of memory, it doesn't immediately allocate physical memory for the requested allocation. Instead it just populates the page table with entries pointing to a special 0 page. And then it marks these pages as read only.

And then the first time you write to such a page, it will cause a page fault. And at that point, the OS is going to modify the page table, get the appropriate physical memory, and store the mapping from the virtual address space to physical address space for the particular page that you touch. And then it will restart the instructions so that it can continue to execute.

You can-- turns out that you can actually m map a terabyte of virtual memory, even on a machine with just a gigabyte of d ram. Because when you call m map, it doesn't actually allocate the physical memory. But then you should be careful, because a process might die from running out of physical memory well after you call m map. Because m map is going to allocate this physical memory whenever you first touch it. And this could be much later than when you actually made the call to m map.

So any questions so far? OK. So what's the difference between malloc and m map? So as I said, malloc is a library call. And it's part of--malloc and free are part of the memory allocation interface of the heat-management code in the c library.

And the heat-management code uses the available system facilities, including the m map function to get a virtual address space from the operating system. And then the heat-management code is going-- within malloc-- is going to attempt to satisfy user requests for heat storage by reusing the memory that it got from the OS as much as possible until it can't do that anymore. And then it will go and call m map to get more memory from the operating system.

So the malloc implementation invokes m map and other system calls to expand the size of the users heap storage. And the responsibility of malloc is to reuse the memory, such that your fragmentation is reduced, and you have good temporal locality, whereas the responsibility of m map is actually getting this memory from the operating system. So any questions on the differences between malloc and m map?

So one question is, why don't we just call m map up all the time, instead of just using malloc? Why don't we just directly call m map? Yes.

STUDENT: [INAUDIBLE]

JULIAN SHUN: Yes, so one answer is that you might have free storage from before that you would want to reuse. And it turns out that m map is relatively heavy weight. So it works on a page granularity. So if you want to do a small allocation, it's quite wasteful to allocate an entire page for that allocation and not reuse it.

You'll get very bad external fragmentation. And when you call m map, it has to go through all of the overhead of the security of the OS and updating the page table and so on. Whereas, if you use malloc, it's actually pretty fast for most allocations, and especially if you have temporal locality where you allocate something that you just recently freed.

So your program would be pretty slow if you used m map all the time, even for small allocations. For big allocations, it's fine. But for small allocations, you should use malloc. Any questions on m map versus malloc?

OK, so I just want to do a little bit of review on how address translation works. So some of you might have seen this before in your computer architecture course. So how it works is, when you access memory location, you access it via the virtual address. And the virtual address can be divided into two parts, where the lower order bits store the offset, and the higher order bits store the virtual page number.

And in order to get the physical address associated with this virtual address, the hardware is going to look up this virtual page number in what's called the page table. And then if it finds a corresponding entry for the virtual page number in the page table, that will tell us the physical frame number. And then the physical frame number corresponds to where this fiscal memory is in d ram. So you can just take the frame number, and then use the same offset as before to get the appropriate offset into the physical memory frame.

So if the virtual page that you're looking for doesn't reside in physical memory, then a page fault is going to occur. And when a page fault occurs, either the operating system will see that the process actually has permissions to look at that memory region, and it will set the permissions and place the entry into the page table so that you can get the appropriate physical address. But otherwise, the operating system might see that this process actually can't access that region memory, and then you'll get a segmentation fault.

It turns out that the page table search, also called a page walk, is pretty expensive. And that's why we have the translation look, a side buffer or TLB, which is essentially a cache for the page table. So the hardware uses a TLB to cache the recent page table look ups into this TLB so that later on when you access the same page, it doesn't have to go all the way to the page table to find the physical address. It can first look in the TLB to see if it's been recently accessed.

So why would you expect to see something that it recently has accessed? So what's one property of a program that will make it so that you get a lot of TLB hits? Yes?

STUDENT: Well, usually [INAUDIBLE] nearby one another, which means they're probably in the same page or [INAUDIBLE].

JULIAN SHUN: Yeah, so that's correct. So the page table stores pages, which are typically four kilobytes. Nowadays there are also huge pages, which can be a couple of megabytes. And most of the accesses in your program are going to be near each other. So they're likely going to reside on the same page for accesses that have been done close together in time.

So therefore you'll expect that many of your recent accesses are going to be stored in the TLB if your program has locality, either spatial or temporal locality or both. So how this architecture works is that the processor is first going to check whether the virtual address you're looking for is in TLB. If it's not, it's going to go to the page table and look it up.

And then if it finds that there, then it's going to store that entry into the TLB. And then next it's going to go get this physical address that it found from the TLB and look it up into the CPU cache. And if it finds it there, it gets it. If it doesn't, then it goes to d ram to satisfy the request. Most modern machines actually have an optimization that allow you to do TLB access in parallel with the L1 cache access. So the L1 cache actually uses virtual addresses instead of fiscal addresses, and this reduces the latency of a memory access.

So that's a brief review of address translation. All right, so let's talk about stacks. So when you execute a serial c and c++ program, you're using a stack to keep track of the function calls and local variables that you have to save. So here, let's say we have this invocation tree, where function a calls Function b, which then returns. And then a calls function c, which calls d, returns, calls e, returns, and then returns again.

Here are the different views of the stack at different points of the execution. So initially when we call a, we have a stack frame for a. And then when a calls b, we're going to place a stack frame for b right below the stack frame of a. So these are going to be linearly ordered.

When we're done with b, then this part of the stack is no longer going to be used, the part for b. And then when it calls c, It's going to allocate a stack frame below a on the stack. And this space is actually going to be the same space as what b was using before. But this is fine, because we're already done with the call to b.

Then when c calls d, we're going to create a stack frame for d right below c. When it returns, we're not going to use that space any more, so then we can reuse it for the stack frame when we call e. And then eventually all of these will pop back up.

And all of these views here share the same view of the stack frame for a. And then for c, d, and e, they all stare share the same view of this stack for c. So this is how a traditional linear stack works when you call a serial c or c++ program. And you can view this as a serial walk over the invocation tree.

There's one rule for pointers. With traditional linear stacks is that a parent can pass pointers to its stack variables down to its children. But not the other way around. A child can't pass a pointer to some local variable back to its parent. So if you do that, you'll get a bug in your program. How many of you have tried doing that before? Yeah, so a lot of you.

So let's see why that causes a problem. So if I'm calling-- if I call b, and I pass a pointer to some local variable in b stack to a, and then now when a calls c, It's going to overwrite the space that b was using. And if b's local variable was stored in the space that c has now overwritten, then you're just going to see garbage. And when you try to access that, you're not going to get the correct value.

So you can pass a pointer to a's local variable down to any of these descendant function calls, because they all see the same view of a stack. And that's not going to be overwritten while these descendant function calls are proceeding. But if you pass it the other way, then potentially the variable that you had a pointer to is going to be overwritten.

So here's one question. If you want to pass memory from a child back to the parent, where would you allocate it? So you can allocate it on the parent. What's another option? Yes? Yes, so another way to do this is to allocate it on the heap.

If you allocate it on the heap, even after you return from the function call, that memory is going to persist. You can also allocate it in the parent's stack, if you want. In fact, some programs are written that way. And one of the reasons why many c functions require you to pass in memory to the function where it's going to store the return value is to try to avoid an expensive heap allocation in the child.

Because if the parent allocates this space to store the result, the child can just put whatever it wants to compute in that space. And the parent will see it. So then the responsibility is up to the parent to figure out whether it wants to allocate the memory on the stack or on the heap. So this is one of the reasons why you'll see many c functions, where one of the arguments is a memory location where the result should be stored.

OK, so that was the serial case. What happens in parallel? So in parallel, we have what's called a cactus stack where we can support multiple views of the stack in parallel. So let's say we have a program where it calls function a, and then a spawns b and c. So b and c are going to be running potentially in parallel. And then c spawns d and e, which can potentially be running in parallel.

So for this program, we could have functions b, d and e all executing in parallel. And a cactus stack is going to allow us to have all of these functions see the same view of this stack as they would have if this program were executed serially. And the silk runtime system supports the cactus stack to make it easy for writing parallel programs. Because now when you're writing programs, you just have to obey the same rules for programming in serial c and c++ with regards to the stack, and then you'll still get the intended behavior.

And it turns out that there's no copying of the stacks here. So all of these different views are seeing the same virtual memory addresses for a. But now there is an issue of how do we implement this cactus stack? Because in the serial case, we could have these later stacks overwriting the earlier stacks. But in parallel, how can we do this?

So does anyone have any simple ideas on how we can implement a cactus stack? Yes?

STUDENT: You could just have each child's stack start in like a separate stack, or just have references to the [INAUDIBLE].

JULIAN SHUN: Yeah, so one way to do this is to have each thread use a different stack. And then store pointers to the different stack frames across the different stacks. There's actually another way to do this, which is easier. OK, yes?

STUDENT: If the stack frames have a maximum-- fixed maximum size-- then you could put them all in the same stack separated by that fixed size.

JULIAN SHUN: Yeah, so if the stacks all have a maximum depth, then you could just allocate a whole bunch of stacks, which are separated by this maximum depth. There's actually another way to do this, which is to not use the stack. So yes?

STUDENT: Could you memory map it somewhere else-- each of the different threads?

JULIAN SHUN: Yes, that's actually one way to do it. The easiest way to do it is just to allocate it off the heap. So instead of allocating the frames on the stack, you just do a heap allocation for each of these stack frames. And then each of these stack frames has a pointer to the parent stack frame.

So whenever you do a function call, you're going to do a memory allocation from the heap to get a new stack frame. And then when you finish a function, you're going to pop something off of this stack, and free it back to the heap. In fact, a lot of early systems for parallel programming use this strategy of heap-based cactus stacks.

Turns out that you can actually minimize the performance impact using this strategy if you optimize the code enough. But there is actually a bigger problem with using a heap-based cactus stack, which doesn't have to do with performance. Does anybody have any guesses of what this potential issue is? Yeah?

STUDENT: It requires you to allocate the heap in parallel.

JULIAN SHUN: Yeah, so let's assume that we can do parallel heap allocation. And we'll talk about that. So assuming that we can do that correctly, what's the issue with this approach? Yeah?

STUDENT: It's that you don't know how big the stack is going to be?

JULIAN SHUN: So let's assume that you can get whatever stack frames you need from the heap, so you don't actually need to put an upper bound on this. Yeah?

STUDENT: We don't know the maximum depth.

JULIAN SHUN: Yeah. So we don't know the maximum depth, but let's say we can make that work. So you don't actually need to know the maximum depth if you're allocating off the heap. Any other guesses? Yeah?

STUDENT: Something to do with returning from the stack that is allocated on the heap to one of the original stacks.

JULIAN SHUN: So let's say we could get that to work as well. So what happens if I try to run some program using this heap-based cactus stack with something that's using the regular stack? So let's say I have some old legacy code that was already compiled using the traditional linear stack. So there's a problem with interoperability here.

Because the traditional code is assuming that, when you make a function call, the stack frame for the function call is going to appear right after the stack frame for the particular call e function. So if you try to mix code that uses the traditional stack as well as this heap-based cactus stack approach, then it's not going to work well together.

One approach is that you can just recompile all your code to use this heap-based cactus stack. Even if you could do that, even if all of the source codes were available, there are some legacy programs that actually in the source code, they do some manipulations with the stack, because they assume that you're using the traditional stack, and those programs would no longer work if you're using a heap-based cactus stack.

So the problem is interoperability with legacy code. Turns out that you can fix this using an approach called thread local memory mapping. So one of the students mentioned memory mapping. But that requires changes to the operating system. So it's not general purpose.

But the heap-based cactus stack turns out to be very simple. And we can prove nice bounds about it. So besides the interoperability issue, heap-based cactus stacks are pretty good in practice, as well as in theory. So we can actually prove a space bound of a cilk program that uses the heap-based cactus stack.

So let's say s 1 is the stack space required by a serial execution of a cilk program. Then the stack space of p worker execution using a heap-based cactus stack is going to be upper bounded by p times s 1. So s p is the space for a p worker execution, and that's less than or equal to p times s 1.

To understand how this works, we need to understand a little bit about how the cilks works stealing algorithm works. So in the cilk work-stealing algorithm, whenever you spawn something of work, or that spawns a new task, is going to work on the task that it spawned.

So therefore, for any leaf in the invocation tree that currently exists, there's always going to be a worker working on it. There's not going to be any leaves in the tree where there's no worker working on it. Because when a worker spawns a task, it creates a new leaf. But then it works immediately on that leaf.

So here we have a-- we have a invocation tree. And for all of the leaves, we have a processor working on it. And with this busy leaves property, we can easily show this space bound. So for each one of these processors, the maximum stack space it's using is going to be upper bounded by s 1, because that's maximum stock space across a serial execution that executes the whole program.

And then since we have p of these leaves, we just multiply s 1 by p, and that gives us an upper bound on the overall space used by a p worker execution. This can be a loose upper bound, because we're double counting here. There's some part of this memory that we're counting more than once, because they're shared among the different processors.

But that's why we have the less than or equal to here. So it's upper bounded by p times s 1. So this is one of the nice things about using a heap-based cactus stack is that you get this good space bound. Any questions on the space bound here?

So let's try to apply this theorem to a real example. So this is the divide and conquer matrix multiplication code that we saw in a previous lecture. So this is-- in this code, we're making eight recursive calls to a divide and conquer function. Each of size n over 2.

And before we make any of these calls, we're doing a malloc to get some temporary space. And this is of size order and squared. And then we free this temporary space at the end. And notice here that the allocations of the temporary matrix obey a stack discipline.

So we're allocating stuff before we make recursive calls. And we're freeing it after, or right before we return from the function. So all this stack-- all the allocations are nested, and they follow a stack discipline. And it turns out that even if you're allocating off the heap, if you follow a stack discipline, you can still use the space bound from the previous slide to upper bound the p worker space.

OK, so let's try to analyze the space of this code here. So first let's look at what the work and span are. So this is just going to be review. What's the work of this divide and conquer matrix multiply? So it's n cubed. So it's n cubed because we have eight solve problems of size n over 2. And then we have to do linear work to add together the matrices.

So our recurrence is going to be t 1 of n is equal to eight times t 1 of n over 2 plus order n squared. And that solves to order n cubed if you just pull out your master theorem card. What about the span? So what's the recurrence here? Yeah, so the span t infinity of n is equal to t infinitive of n over 2 plus a span of the addition. And what's the span of the addition?

STUDENT: [INAUDIBLE]

JULIAN SHUN: No, let's assume that we have a parallel addition. We have nested silk four loops. Right, so then the span of that is just going of be log n. Since the span of 1 silk four loop is log n and when you nest them, you just add together the span. So it's going to be t infinity of n is equal to t infinity of n over 2 plus order log n. And what does that solve to?

Yeah, so it's going to solve to order log squared n. Again you can pull out your master theorem card, and look at one of the three cases. OK, so now let's look at the space. What's going to be the recurrence for the space? Yes.

STUDENT: [INAUDIBLE]

JULIAN SHUN: The only place we're generating new space is when we call this malloc here. So they're all seeing the same original matrix. So what would the recurrence be? Yeah?

STUDENT: [INAUDIBLE]

JULIAN SHUN: Yeah.

STUDENT: [INAUDIBLE]

JULIAN SHUN: So the n square term is right. Do we actually need eight subproblems of size n over 2? What happens after we finish one of these sub problems? Are we still going to use the space for it?

STUDENT: Yeah, you free the memory after the [INAUDIBLE].

JULIAN SHUN: Right. So you can actually reuse the memory. Because you free the memory you allocated after each one of these recursive calls. So therefore the recurrence is just going to be s of n over 2 plus theta n squared. And what does that solve to?

STUDENT: [INAUDIBLE]

JULIAN SHUN: N squared. Right. So here the n squared term actually dominates. You have a decreasing geometric series. So it's dominated at the root, and you get theta of n squared. And therefore by using the busy leaves property and the theorem for the space bound, this tells us that on p processors, the space is going to be bounded by p times n squared.

And this is actually pretty good since we have a bound on this. It turns out that we can actually prove a stronger bound for this particular example. And I'll walk you through how we can prove this stronger bound. Here's the order p times n squared is already pretty good. But we can actually do better if we look internally at how this algorithm is structured.

So on each level of recursion, we're branching eight ways. And most of the space is going to be used near the top of this recursion tree. So if I branch as much as possible near the top of my recursion tree, then that's going to give me my worst case space bound. Because the space is decreasing geometrically as I go down the tree.

So I'm going to branch eight ways until I get to some level k in the recursion tree where I have p nodes. And at that point, I'm not going to branch anymore because I've already used up all p nodes. And that's the number of workers I have.

So let's say I have this level k here, where I have p nodes. So what would be the value of k here? If I branch eight ways how many levels do I have to go until I get to p nodes? Yes.

STUDENT: It's log base 8 of p.

JULIAN SHUN: Yes. It's log base 8 of p. So we have eight, the k, equal p, because we're branching k ways. And then using some algebra, you can get it so that k is equal to log base 8 of p, which is equal to log base 2 of p divided by 3. And then at this level k downwards, it's going to decrease geometrically.

So the space is going to be dominant at this level k. So the space decreases geometrically as you go down from level k, and also as you go up from level k. So therefore we can just look at what the space is at this level k here.

So the space is going to be p times the size of each one of these nodes squared. And the size of each one of these nodes is going to be n over 2 to the log base 2 of p over 3. And then we square that because we're using n squared temporary space.

So if you solve that, that gives you p to the one-third times n squared, which is better than the upper bound we saw earlier of order p times n squared. So you can work out the details for this example. Not all the details are shown on this slide. You need to show that the level k here actually dominates all the other levels in the recursion tree.

But in general, if you know what the structure of the algorithm, is you can potentially prove a stronger space bound than just applying the general theorem we showed on the previous slide. So any questions on this?

OK, so as I said before, the problem with heap-based linkage is that parallel functions fail to interoperate with legacy and third-party serial binaries. Yes, was there a question?

STUDENT: I actually do have a question.

JULIAN SHUN: Yes.

STUDENT: [INAUDIBLE]

JULIAN SHUN: Yes.

STUDENT: How do we know that the workers don't split along the path of the [INAUDIBLE] instead of across or horizontal.

JULIAN SHUN: Yes. So you don't actually know that. But this turns out to be the worst case. So if it branches any other way, the space is just going to be lower. So you have to argue that this is going to be the worst case, and it's going to be-- intuitively it's the worst case, because you're using most of the memory near the root of the recursion tree. So if you can get all p nodes as close as possible to the root, that's going to make your space as high as possible. It's a good question.

So parallel functions fail to interoperate with legacy and third-party serial binaries. Even if you can recompile all of this code, which isn't always necessarily the case, you can still have issues if the legacy code is taking advantage of the traditional linear stack inside the source code. So our implementation of cilk uses a less space efficient strategy that is interoperable with legacy code. And it uses a pool of linear stacks instead of a heap-based strategy.

So we're going to maintain a pool of linear stacks lying around. There's going to be more than p stacks lying around. And whenever a worker tries to steal something, it's going to try to acquire one of these tasks from this pool of linear tasks. And when it's done, it will return it back.

But when it finds that there's no more linear stacks in this pool, then it's not going to steal anymore. So this is still going to preserve the space bound, as long as the number of stocks is a constant times the number of processors. But it will affect the time bounds of the work-stealing algorithm. Because now when a worker is idle, it might not necessarily have the chance to steal if there are no more stacks lying around.

This strategy doesn't require any changes to the operating system. There is a way where you can preserve the space and the time bounds using thread local memory mapping. But this does require changes to the operating system. So our implementation of cilk uses a pool of linear stacks, and it's based on the Intel implementation. OK.

All right, so we talked about stacks, and that we just reduce the problem to heap allocation. So now we have to talk about heaps. So let's review some basic properties of heap-storage allocators. So here's a definition.

The allocator speed is the number of allocations and d allocations per second that the allocator can sustain. And here's a question. Is it more important to maximize the allocator speed for large blocks or small blocks? Yeah?

STUDENT: Small blocks?

JULIAN SHUN: So small blocks. Here's another question. Why? Yes?

STUDENT: So you're going to be doing a lot of [INAUDIBLE].

JULIAN SHUN: Yes, so one answer is that you're going to be doing a lot more allocations and deallocations of small blocks than large blocks. There's actually a more fundamental reason why it's more important to optimize for small blocks. So anybody? Yeah?

STUDENT: [INAUDIBLE] basically not being able to make use of pages.

JULIAN SHUN: Yeah, so that's another reason for small blocks. It's more likely that it will lead to fragmentation if you don't optimize for small blocks. What's another reason? Yes.

STUDENT: Wouldn't it just take longer to allocate larger blocks anyway? So the overhead is going to be more noticeable if you have a big overhead when you allocate small blocks versus large blocks?

JULIAN SHUN: Yeah. So the reason-- the main reason is that when you're allocating a large-- when you're allocating a block, a user program is typically going to write to all the bytes in the block. And therefore, for a large block, it takes so much time to write that the allocator time has little effect on the overall running time.

Whereas if a program allocates many small blocks, the amount of work-- useful work-- it's actually doing on the block is going to be-- it can be comparable to the overhead for the allocation. And therefore, all of the allocation overhead can add up to a significant amount for small blocks.

So essentially for large blocks, you can amortize away the overheads for storage allocation, whereas for small, small blocks, it's harder to do that. Therefore, it's important to optimize for small blocks. Here's another definition. So the user footprint is the maximum over time of the number u of bytes in use by the user program.

And these are the bytes that are allocated and not freed. And this is measuring the peak memory usage. It's not necessarily equal to the sum of the sizes that you have allocated so far, because you might have reused some of that. So the user footprint is the peak memory usage and number of bytes.

And the allocator footprint is the maximum over time of the number of a bytes that the memory provided to the locator by the operating system. And the reason why the allocator footprint could be larger than the user footprint, is that when you ask the OS for some memory, it could give you more than what you asked for. And similarly, if you ask malloc for some amount of memory, it can also give you more than what you asked for.

And the fragmentation is defined to be a divided by u. And a program with low fragmentation will keep this ratio as low as possible, so keep the allocator footprint as close as possible to the user footprint. And in the best case, this ratio is going to be one. So you're using all of the memory that the operating system allocated.

One remark is that the allocator footprint a usually gross monotonically for many allocators. So it turns out that many allocators do m maps to get more memory. But they don't always free this memory back to the OS. And you can actually free memory using something called m unmap, which is the opposite of m map, to give memory back to the OS. But this turns out to be pretty expensive.

In modern operating systems, their implementation is not very efficient. So many allocators don't use m unmap. You can also use something called m advise. And what m advise does is it tells the operating system that you're not going to be using this page anymore but to keep it around in virtual memory. So this has less overhead, because it doesn't have to clear this entry from the page table. It just has to mark that the program isn't using this page anymore.

So some allocators use m advise with the option, don't need, to free memory. But a is usually still growing monotonically over time, because allocators don't necessarily free all of the things back to the OS that they allocated.

Here's a theorem that we proved in last week's lecture, which says that the fragmentation for binned free list is order log base 2 of u, or just order log u. And the reason for this is that you're can have log-based 2 of u bins. And for each bin it can basically contain u bytes of storage.

So overall you can use-- overall, you could have allocated u times log u storage, and only be using u of those bytes. So therefore the fragmentation is order log u.

Another thing to note is that modern 64-bit processors only provide about 2 to 48 bytes of virtual address space. So this is sort of news because you would probably expect that, for a 64-bit processor, you have to the 64 bytes of virtual address space. But that turns out not to be the case.

So they only support to the 48 bytes. And that turns out to be enough for all of the programs that you would want to write. And that's also going to be much more than the physical memory you would have on a machine. So nowadays, you can get a big server with a terabyte of memory, or to the 40th bytes of physical memory, which is still much lower than the number of bytes in the virtual address space.

Any questions? OK, so here's some more definitions. So the space overhead of an allocator is a space used for bookkeeping. So you could store-- perhaps you could store headers with the blocks that you allocate to keep track of the size and other information. And that would contribute to the space overhead

Internal fragmentation is a waste due to allocating larger blocks in the user request. So you can get internal fragmentation if, when you call malloc, you get back a block that's actually larger than what the user requested. We saw on the bin free list algorithm, we're rounding up to the nearest power of 2's.

If you allocate nine bytes, you'll actually get back 16 bytes in our binned-free list algorithm from last lecture. So that contributes to internal fragmentation. It turns out that not all binned-free list implementations use powers of 2. So some of them use other powers that are smaller than 2 in order to reduce the internal fragmentation.

Then there's an external fragmentation, which is the waste due to the inability to use storage because it's not contiguous. So for example, if I allocated a whole bunch of one byte things consecutively in memory, then I freed every other byte. And now I want to allocate a 2-byte thing, I don't actually have contiguous mammary to satisfy that request, because all of my free memory-- all of my free bytes are in one-bite chunks, and they're not next to each other.

So this is one example of how external fragmentation can happen after you allocate stuff and free stuff. Then there's blow up. And this is for a parallel locator. The additional space beyond what a serial locator would require. So if a serial locator requires s space, and a parallel allocator requires t space, then it's just going to be t over s. That's the blow up.

OK, so now let's look at some parallel heap allocation strategies. So the first strategy is to use a global heap. And this is how the default c allocator works. So if you just use a default c allocator out of the box, this is how it's implemented.

It uses a global heap where all the accesses to this global heap are protected by mutex. You can also use lock-free synchronization primitives to implement this. We'll actually talk about some of these synchronization primitives later on in the semester. And this is done to preserve atomicity because you can have multiple threads trying to access the global heap at the same time. And you need to ensure that races are handled correctly.

So what's the blow up for this strategy? How much more space am I using than just a serial allocator? Yeah.

STUDENT: [INAUDIBLE]

JULIAN SHUN: Yeah, so the blow up is one. Because I'm not actually using any more space than the serial allocator. Since I'm just maintaining one global heap, and everybody is going to that heap to do allocations and deallocations. But what's the potential issue with this approach? Yeah?

STUDENT: Performance hit for that block coordination.

JULIAN SHUN: Yeah, so you're going to take a performance hit for trying to acquire this lock. So basically every time you do a allocation or deallocation, you have to acquire this lock. And this is pretty slow, and it gets slower as you increase the number of processors.

Roughly speaking, acquiring a lock to perform is similar to an L2 cache access. And if you just run a serial allocator, many of your requests are going to be satisfied just by going into the L1 cache. Because you're going to be allocating things that you recently freed, and those things are going to be residing in L1 cache.

But here, before you even get started, you have to grab a lock. And you have to pay a performance hit similar to an L2 cache access. So that's bad. And it gets worse as you increase the number of processors. So the contention increases as you increase the number of threads. And then you can't-- you're not going to be able to get good scalability.

So ideally, as the number of threads or processors grows, the time to perform an allocation or deallocation shouldn't increase. But in fact, it does. And the most common reason for loss of scalability is lock contention.

So here all of the processes are trying to acquire the same lock, which is the same memory address. And if you recall from the caching lecture, or the multicore programming lecture, every time you acquire a memory location, you have to bring that cache line into your own cache, and then invalidate the same cache line in other processors' caches.

So if all the processors are doing this, then this cache line is going to be bouncing around among all of the processors' caches, and this could lead to very bad performance. So here's a question. Is lock contention more of a problem for large blocks or small blocks? Yes.

STUDENT: So small blocks.

JULIAN SHUN: Here's another question. Why? Yes.

STUDENT: Because by the time it takes to finish using the small block, then the allocator is usually small. So you do many allocations and deallocations, which means you have to go through the lock multiple times.

JULIAN SHUN: Yeah. So one of the reasons is that when you're doing small allocations, that means that your request rate is going to be pretty high. And your processors are going to be spending a lot of time acquiring this lock. And this can exacerbate the lock contention.

And another reason is that when you allocate a large block, you're doing a lot of work, because you have to write-- most of the time you're going to write to all the bytes in that large block. And therefore you can amortize the overheads of the storage allocator across all of the work that you're doing.

Whereas for small blocks, in addition to increasing this rate of memory requests, it's also-- there's much less work to amortized to overheads across. So any questions? OK, good.

All right. So here's another strategy, which is to use local heaps. So each thread is going to maintain its own heap. And it's going to allocate out of its own heap. And there's no locking that's necessary. So when you allocate something, you get it from your own heap. And when you free something, you put it back into your own heap. So there's no synchronization required. So that's a good thing. It's very fast. What's a potential issue with this approach? Yes.

STUDENT: It's using a lot of extra space.

JULIAN SHUN: Yes, so this approach, you're going to be using a lot of extra space. So first of all, because you have to maintain multiple heaps. And what's one phenomenon that you might see if you're executing a program with this local-heap approach? So it's a space-- could the space potentially keep growing over time? Yes.

STUDENT: You could maybe like allocate every one process [INAUDIBLE].

JULIAN SHUN: Yeah. Yeah, so you could actually have an unbounded blow up. Because if you do all of the allocations in one heap, and you free everything in another heap, then whenever the first heap does an allocation, there's actually free space sitting around in another heap. But it's just going to grab more memory from the operating system. So you're blow up can be unbounded.

And this phenomenon, it's what's called memory drift. So blocks allocated by one thread are freed by another thread. And if you run your program for long enough, your memory consumption can keep increasing. And this is sort of like a memory leak.

So you might see that if you have a memory drift problem, your program running on multiple processors could run out of memory eventually. Whereas if you just run it on a single core, it won't run out of memory. And here it's because the allocator isn't smart enough to reuse things in other heaps. So what's another strategy you can use to try to fix this? Yes?

STUDENT: [INAUDIBLE]

JULIAN SHUN: Sorry, can you repeat your question?

STUDENT: [INAUDIBLE]

JULIAN SHUN: Because if you keep allocating from one thread, if you do all of your allocations in one thread, and do all of your deallocations on another thread, every time you allocate from the first thread, there's actually memory sitting around in the system. But the first thread isn't going to see it, because it only sees its own heap. And it's just going to keep grabbing more memory from the OS.

And then the second thread actually has this extra memory sitting around. But it's not using it. Because it's only doing the freeze. It's not doing allocate. And if we recall the definition of blow up is, how much more space you're using compared to a serial execution of a program.

If you executed this program on a single core, you would only have a single heap that does the allocations and frees. So you're not going to-- your memory isn't going to blow up. It's just going to be constant over time. Whereas if you use two threads to execute this, the memory could just keep growing over time. Yes?

STUDENT: [INAUDIBLE]

JULIAN SHUN: So, it just-- so if you remember the binned-free list approach, let's say we're using that. Then all you have to do is set some pointers in your binned-free lists data structure, as well as the block that you're freeing, so that it appears in one of the linked lists. So you can do that even if some other processor allocated that block.

OK, so what what's another strategy that can avoid this issue of memory drift? Yes?

STUDENT: Periodically shuffle the free memory that's being used on different heaps.

JULIAN SHUN: Yeah. So that's a good idea. You could periodically rebalance the memory. What's a simpler approach to solve this problem? Yes?

STUDENT: Make it all know all of the free memory?

JULIAN SHUN: Sorry, could you repeat that?

STUDENT: Make them all know all of the free memory?

JULIAN SHUN: Yes. So you could have all of the processors know all the free memory. And then every time it grabs something, it looks in all the other heaps. That does require a lot of synchronization overhead. Might not perform that well. What's an easier way to solve this problem? Yes.

STUDENT: [INAUDIBLE]

JULIAN SHUN: So you could restructure your program so that the same thread does the allocation and frees for the same memory block. But what if you didn't want to restructure your program? How can you change the allocator? So we want the behavior that you said, but we don't want to change our program. Yes.

STUDENT: You could have a single free list that's protected by synchronization.

JULIAN SHUN: Yeah, so you could have a single free list. But that gets back to the first strategy of having a global heap. And then you have high synchronization overheads. Yes.

STUDENT: You could have the free map to the thread that it came from or for the pointer that corresponds to-- that allocated it.

JULIAN SHUN: So you're saying free back to the thread that allocated it? Yes, so that that's exactly right. So here each object, when you allocate it, it's labeled with an owner. And then whenever you free it, you return it back to the owner.

So the objects that are allocated will eventually go back to the owner's heap if they're not in use. And they're not going to be free lying around in somebody else's heap. The advantage of this approach is that you get fast allocation and freeing of local objects.

Local objects are objects that you allocated. However, free remote objects require some synchronization. Because you have to coordinate with the other threads' heap that you're sending the memory object back to. But this synchronization isn't as bad as having a global heap, since you only have to talk to one other thread in this case.

You can also bound the blow up by p. So the reason why the blow up is upper bounded by p is that, let's say the serial allocator uses at most x memory. In this case, each of the heaps can use at most x memory, because that's how much the serial program would have used. And you have p of these heaps, so overall you're using p times x memory. And therefore the ratio is upper bounded by p. Yes?

STUDENT: [INAUDIBLE]

JULIAN SHUN: So when you free an object, it goes-- if you allocated that object, it goes back to your own heap. If your heap is empty, it's actually going to get more memory from the operating system. It's not going to take something from another thread's heap.

But the maximum amount of memory that you're going to allocate is going to be upper bounded by x. Because the sequential serial program took that much.

STUDENT: [INAUDIBLE]

JULIAN SHUN: Yeah. So the upper bound for the blow up is p. Another advantage of this approach is that it's resilience-- it has resilience to false sharing. So let me just talk a little bit about false sharing. So true sharing is when two processors are trying to access the same memory location.

And false sharing is when multiple processors are accessing different memory locations, but those locations happen to be on the same cache line. So here's an example. Let's say we have two variables, x and y. And the compiler happens to place x and y on the same cache line.

Now, when the first processor writes to x, it's going to bring this cache line into its cache. When the other processor writes to y, since it's on the same cache line, it's going to bring this cache line to y's cache. And then now, the first processor writes x, it's going to bring this cache line back to the first processor's cache.

And then you can keep-- you can see this phenomenon keep happening. So here, even though the processors are writing to different memory locations, because they happen to be on the same cache line, the cache line is going to be bouncing back and forth on the machine between the different processors' caches. And this problem gets worse if more processors are accessing this cache line.

So in this-- this can be quite hard to debug. Because if you're using just variables on the stack, you don't actually know necessarily where the compiler is going to place these memory locations. So the compiler could just happen to place x and y in the same cache block. And then you'll get this performance hit, even though it seems like you're accessing different memory locations.

If you're using the heap for memory allocation, you have more knowledge. Because if you allocate a huge block, you know that all of the memory locations are contiguous in physical memory. So you can just space your-- you can space the accesses far enough apart so that different processes aren't going to touch the same cache line.

A more general approach is that you can actually pad the object. So first, you can align the object on a cache line boundary. And then you pad out the remaining memory locations of the objects so that it fills up the entire cache line. And now there's only one thing on that cache line. But this does lead to a waste of space because you have this wasted padding here.

So program can induce false sharing by having different threads process nearby objects, both on the stack and on the heap. And then an allocator can also induce false sharing in two ways. So it can actively induce false sharing. And this is when the allocator satisfies memory requests from different threads using the same cache block.

And it can also do this passively. And this is when the program passes objects lying around in the same cache line. So different threads, and then the allocator reuses the object storage after the objects are free to satisfy requests from those different threads. And the local ownership approach tends to reduce false sharing because the thread that allocates an object is eventually going to get it back. You're not going to have it so that an object is permanently split among multiple processors' heaps.

So even if you see false sharing in local ownership, it's usually temporary. Eventually it's going-- the object is going to go back to the heap that it was allocated from, and the false sharing is going to go away. Yes?

STUDENT: Are the local heaps just three to five regions in [INAUDIBLE]?

JULIAN SHUN: I mean, you can implement it in various ways. I mean can have each one of them have a binned-free list allocator, so there's no restriction on where they have to appear in physical memory. There are many different ways where you can-- you can basically plug-in any serial locator for the local heap.

So let's go back to parallel heap allocation. So I talked about three approaches already. Here's a fourth approach. This is called the hoard allocator. And this was actually a pretty good allocator when it was introduced almost two decades ago. And it's inspired a lot of further research on how to improve parallel-memory allocation.

So let me talk about how this works. So in the hoard allocator, we're going to have p local heaps. But we're also going to have a global heap. The memory is going to be organized into large super blocks of size s. And s is usually a multiple of the page size. So this is the granularity at which objects are going to be moved around in the allocator.

And then you can move super blocks between the local heaps and the global heaps. So when a local heap becomes-- has a lot of super blocks that are not being fully used and you can move it to the global heap, and then when a local heap doesn't have enough memory, it can go to the global heap to get more memory. And then when the global heap doesn't have any more memory, then it gets more memory from the operating system.

So this is sort of a combination of the approaches that we saw before. The advantages are that this is a pretty fast allocator. It's also scalable. As you add more processors, the performance improves. You can also bound the blow up. And it also has resilience to false sharing, because it's using local heaps.

So let's look at how an allocation using the hoard allocator works. So let's just assume without loss of generality that all the blocks are the same size. So we have fixed-size allocation. So let's say we call malloc in our program. And let's say thread i calls the malloc.

So what we're going to do is we're going to check if there is a free object in heap i that can satisfy this request. And if so, we're going to get an object from the fullest non-full super block in i's heap. Does anyone know why we want to get the object from the fullest non-full super block? Yes.

STUDENT: [INAUDIBLE]

JULIAN SHUN: Right. So when a super block needs to be moved, it's as dense as possible. And more importantly, this is to reduce external fragmentation. Because as we saw in the last lecture, if you skew the distribution of allocated memory objects to as few pages, or in this case, as few super blocks as possible, that reduces your external fragmentation.

OK, so if it finds it in its own heap, then it's going to allocate an object from there. Otherwise, it's going to check the global heap. And if there's something in the global heap-- so here it says, if the global heap is empty, then it's going to get a new super block from the OS. Otherwise, we can get a super block from the global heap, and then use that one.

And then finally we set the owner of the block we got either from the OS or from the global heap to i, and then we return that free object to the program. So this is how a malloc works using the hoard allocator. And now let's look at hoard deallocation.

Let use of i be the in use storage in heap i. This is the heap for thread i. And let a sub i be the storage owned by heap i. The hoard allocator maintains the following invariant for all heaps i. And the invariant is as follows. So u sub i is always going to be greater than or equal to the min of a sub i minus 2 times s. Recall s is the super block size. And a sub i over 2.

So how it implements this is as follows. When we call free of x, let's say x is owned by thread i, then we're going to put x back into heap i, and then we're going to check if the n u storage in heap i, u sub i is less than the min of a sub i minus 2 s and a sub i over 2.

And what this condition says, if it's true, it means that your heap is, at most, half utilized. Because if it's smaller than this, it has to be smaller than a sub i over 2. That means there's twice as much allocated than used in the local heap i. And therefore there must be some super block that's at least half empty. And you move that super block, or one of those super blocks, to the global heap.

So any questions on how the allocation and deallocation works? So since we're maintaining this invariant, it's going to allow us to approve a bound on the blow up. And I'll show you that on the next slide. But before I go on, are there any questions?

OK, so let's look at how we can bound the blow up of the hoard allocator. So there is actually a lemma that we're going to use and not prove. The lemma is that the maximum storage allocated in the global heap is at most a maximum storage allocated in the local heaps. So we just need to analyze how much storage is allocated in the local heaps. Because the total amount of storage is going to be, at most, twice as much, since the global heap storage is dominated by the local heap storage.

So you can prove this lemma by case analysis. And there's the hoard paper that's available on learning modules. And you're free to look at that if you want to look at how this is proved. But here I'm just going to use this lemma to prove this theorem, which says that, let u be the user footprint for a program. And let a be the hoard's allocator footprint.

We have that a as upper bounded by order u plus s p. And therefore, a divided by u, which is a blowup, is going to be 1 plus order s p divided by u. OK, so let's see how this proof works. So we're just going to analyze the storage in the local heaps.

Now recall that we're always satisfying this invariant here, where u sub i is greater than the min of a sub i minus 2 s and a sub i over 2. So the first term says that we can have 2 s on utilized storage per heap. So it's basically giving two super blocks for free to each heap. And they don't have to use it. They can basically use it as much as they want.

And therefore, the total amount of storage contributed by the first term is going to be order s p, because each processor has up to 2 s unutilized storage. So that's where the second term comes from here. And the second term, a sub i over 2-- this will give us the first-term order u.

So this says that the allocated storage is at most twice the use storage for-- and then if you sum up across all the processors, then there's a total of order use storage that's allocated. Because the allocated storage can be at most twice the used storage.

OK, so that's the proof of the blow up for hoard. And this is pretty good. It's 1 plus some lower order term. OK, so-- now these are some other allocators that people use. So jemalloc is a pretty popular one. Has a few differences with hoard. It has a separate global lock for each different allocation size.

It allocates the object with the smallest address among all the objects of the requested size. And it releases empty pages using m advise, which we talked about-- I talked about earlier. And it's pretty popular because it has good performance, and it's pretty robust to different allocation traces. There's also another one called SuperMalloc, which is an up and coming contender. And it was developed by Bradley Kuszmaul.

Here are some allocator speeds for the allocators that we looked at for our particular benchmark. And for this particular benchmark, we can see that SuperMalloc actually does really well. It's more than three times faster than jemalloc, and jemalloc is more than twice as fast as hoard. And then the default allocator, which uses a global heap is pretty slow, because it can't get good speed up.

And all these experiments are in 32 threads. I also have the lines of code. So we see that SuperMalloc actually has very few lines of code compared to the other allocators. So it's relatively simple. OK so, I also have some slides in Garbage Collection. But since we're out of time, I'll just put these slides online and you can read them.