

Ben Bitdiddle created a new function, `int matrix_magic(Matrix m)` for his company's top-grossing big data processing package. As `matrix_magic` is performance critical, Ben wanted to autotune a bunch of parameters in the function using an autotuner. For that, he found the biggest matrix available, and used it as the input to the autotuner. His autotuned `matrix_magic` function will always be faster than the original `matrix_magic` without autotuning.

True False

Solution:

False. Overfitting - the tuned version might work well for big matrices but not for small matrices.

Professors de Lancie and Barker are working independently on fixed-size serial heap allocation.

Professor de Lancie implements a linked list which supports two $\mathcal{O}(1)$ -time operations: adding an element to the tail, and removing an element from the head. Professor Barker implements a linked list which also supports two $\mathcal{O}(1)$ -time operations: adding an element to the head, and removing an element from the head. The professors use their respective data structures as free lists in their respective allocators.

True or False:

- Professor Barker's implementation will likely run faster than Professor de Lancie's implementation due to increased temporal locality.
- Professor de Lancie's implementation will likely use less space than Professor Barker's implementation due to decreased external fragmentation.

Solutions:

- True. When Professor Barker removes an element from the free list, he removes the same one he's most recently put on - this is good temporal locality. Professor Lancie does the opposite.
- False. Especially when it's a fixed size allocator, exactly which memory block we give out doesn't really impact the overall fragmentation. Either way, the two implementations will use approximately the same total amount of memory for a given trace (and thus have approximately the same fragmentation)

Professor Harrison writes an application that only allocates and frees 1040-byte objects. He has a choice between two allocators. The fixed-sized allocator uses a free list of 1040-byte blocks. The variable-sized allocator uses binned free lists with blocks that are exact powers of 2. What are the likely advantages of the fixed-size allocator over the variable-sized allocator?

- A. Allocating and freeing are faster.
- B. Less internal fragmentation.
- C. Less external fragmentation.
- D. Less false sharing.
- E. Fewer TLB (translation lookaside buffer) misses.

Solution:

A, B, E.

A fixed size allocator is fast because it only has one free list.

Blocks given are exactly the desired size, so no internal fragmentation.

The binned free list should only be dealing with 2048-size blocks, so there shouldn't be any difference in external fragmentation - one allocator gives a block exactly when the other allocator does too.

The fixed size allocator isn't cache aligned, so more false sharing.

The fixed size allocator uses less overall memory, so there are less total memory pages - thus, there are less TLB misses (since TLB misses directly correspond with new pages in memory)

After analyzing the memory request trace of a program, Ben Bitdiddle implements a fixed-size memory allocator that allocates and frees 128-byte objects. His allocator takes a 4096-byte page of memory and splits it into blocks of size 128 bytes. It uses 96 bits at the beginning of the block for bookkeeping. To keep track of which blocks are free, it uses a bitmap placed in the bookkeeping area at the beginning of the page.

Ben now decides to adapt his allocator to work in a multithreaded environment with exactly two threads. He does this by splitting the blocks on the page into two sets such that each thread allocates from its own half. He also splits the bitmap in half. He pads each half of the bitmap to 64 bits so that the two threads can update the two halves independently. Nevertheless, Ben's allocator suffers from poor performance. Which of the following explanations are most likely the reasons for Ben's poor performance? (Select all that apply)

- A. Poor space utilization
- B. External fragmentation
- C. True sharing of the bitmap
- D. False sharing of the bitmap
- E. None of the above

Solution:

A, D

If one thread allocates much more than the other, we waste half the storage.

There's no reason for there to be any difference in external fragmentation.

The bitmap isn't actually shared, so that's not true sharing.

The bitmaps are on the same cache line, so that is false sharing.