Lista 7

Zadanie 7

> Rozważmy zarządzanie pamięcią w środowisku wielowątkowym. Czemu algorytmy przydziału pamięci często cierpią na **zjawisko rywalizacji o blokady** (ang.lock contention)?

Lock conetntion: If thread A has a lock and thread B wants to acquire that same lock, thread B will have to wait until thread A releases the lock.

Musimy pozwolić tylko jednemu wątkowi na modyfikowanie sterty (kondycja wyścigowa) -> Malloc używa locka.

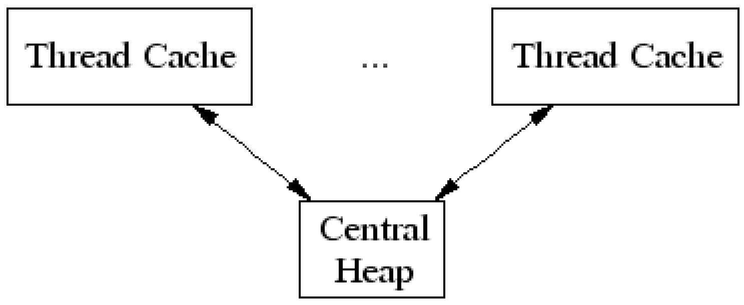
> Dlaczego błędnie zaprojektowany algorytm przydziału może powodować problemy z wydajnością programów wynikających z **fałszywego współdzielenia** (ang. false sharing)?

False sharing is a performance-degrading usage pattern that can arise in systems with distributed, coherent caches at the size of the smallest resource block managed by the caching mechanism.

Mamy dwa procesy, robią malloca zaraz po sobie i dostają pamięć w tym samym keszu. Podczas zapisu w linii keszu ustawia sie bit dirty. Inny rdzeń jak chcez użyć tej strony to musi ją zfeczować z niższego kesza albo ramu. [Z6L4]

> Opisz pobieżnie struktury danych i pomysły wykorzystywane w bibliotece thread-caching malloc.

Overview



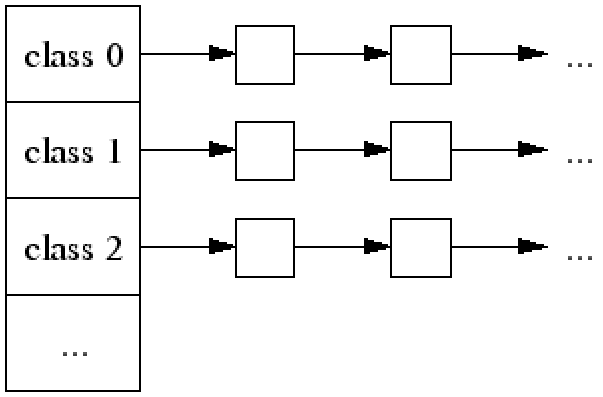
TCMalloc assigns each thread a thread-local cache. Small allocations are satisfied from the thread-local cache. Objects are moved from central data structures into a thread-local cache as needed, and periodic garbage collections are used to migrate memory back from a thread-local cache into the central data structures.

Large (> 32K) objects are allocated directly from the central heap using a page-level allocator (a page is a 4K aligned region of memory). I.e., a large object is always page-aligned and occupies an integral number of pages.

> Wyjaśnij jak zostały użyte do poradzenia sobie z problemem efektywnego przydziału pamięci w programach wielowątkowych?

Small Object Allocation

Each small object size maps to one of approximately 170 allocatable size-classes. For example, all allocations in the range 961 to 1024 bytes are rounded up to 1024.

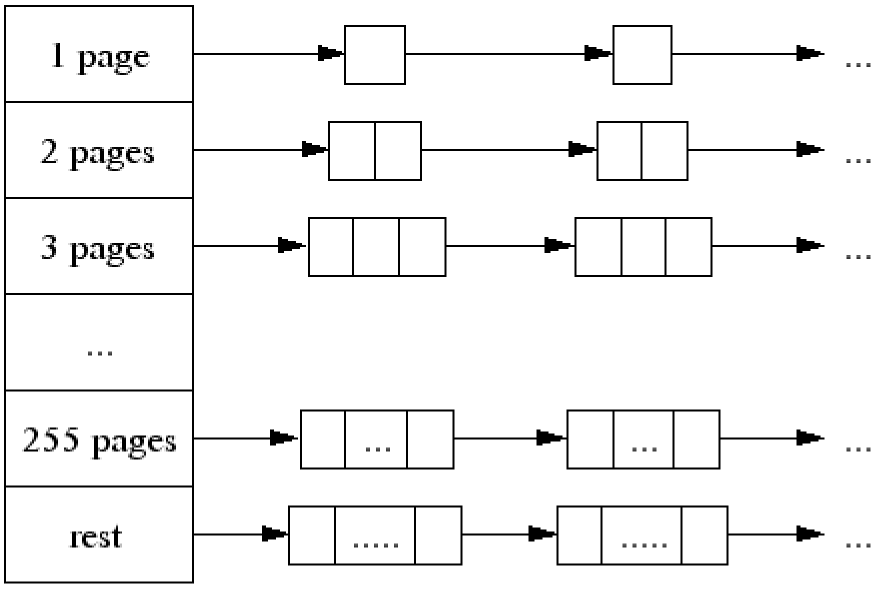


A thread cache contains a singly linked list of free objects per size-class.

- When allocating a small object: (1) We map its size to the corresponding size-class. (2) Look in the corresponding free list in the thread cache for the current thread. (3) If the free list is not empty, we remove the first object from the list and return it. When following this fast path, TCMalloc acquires no locks at all. This helps speed-up allocation significantly because a lock/unlock pair takes approximately 100 nanoseconds on a 2.8 GHz Xeon.
- If the free list is empty: (1) We fetch a bunch of objects from a central free list for this size-class (the central free list is shared by all threads). (2) Place them in the thread-local free list. (3) Return one of the newly fetched objects to the applications.
- If the central free list is also empty: (1) We allocate a run of pages from the central page allocator. (2) Split the run into a set of objects of this size-class. (3) Place the new objects on the central free list. (4) As before, move some of these objects to the thread-local free list.

Large Object Allocation

A large object size (> 32K) is rounded up to a page size (4K) and is handled by a central page heap. The central page heap is again an array of free lists. For i < 256, the kth entry is a free list of runs that consist of k pages. The 256th entry is a free list of runs that have length >= 256 pages:



An allocation for k pages is satisfied by looking in the kth free list. If that free list is empty, we look in the next free list, and so forth. Eventually, we look in the last free list if necessary. If that fails, we fetch memory from the system (using sbrk, mmap, or by mapping in portions of /dev/mem).

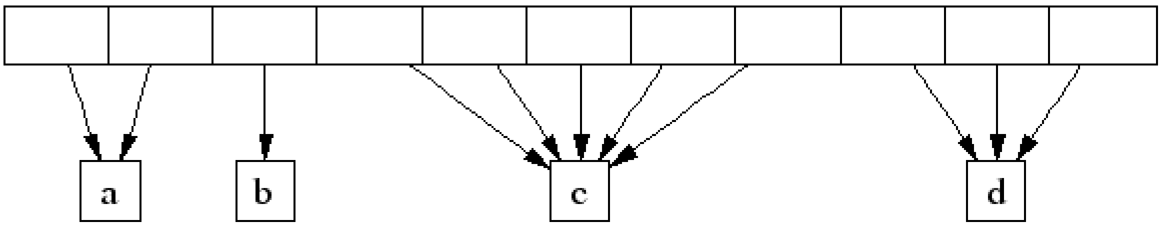
If an allocation for k pages is satisfied by a run of pages of length > k, the remainder of the run is re-inserted back into the appropriate free list in the page heap.

poniżej nudy ale gdyby *ktoś* sie czepiał to ma

Spans

The heap managed by TCMalloc consists of a set of pages. A run of contiguous pages is represented by a Span object. A span can either be allocated, or free. If free, the span is one of the entries in a page heap linked-list. If allocated, it is either a large object that has been handed off to the application, or a run of pages that have been split up into a sequence of small objects. If split into small objects, the size-class of the objects is recorded in the span.

A central array indexed by page number can be used to find the span to which a page belongs. For example, span a below occupies 2 pages, span b occupies 1 page, span c occupies 5 pages and span d occupies 3



pages. A 32-bit address space can fit 2^20 4K pages, so this central array takes 4MB of space, which seems acceptable. On 64-bit machines, we use a 3-level radix tree instead of an array to map from a page number to the corresponding span pointer.

Deallocation

When an object is deallocated, we compute its page number and look it up in the central array to find the corresponding span object. The span tells us whether or not the object is small, and its size-class if it is small. If the object is small, we insert it into the appropriate free list in the current thread's thread cache. If the thread cache now exceeds a predetermined size (2MB by default), we run a garbage collector that moves unused objects from the thread cache into central free lists.

If the object is large, the span tells us the range of pages covered by the object. Suppose this range is [p,q]. We also lookup the spans for pages p-1 and q+1. If either of these neighboring spans are free, we coalesce them with the [p,q] span. The resulting span is inserted into the appropriate free list in the page heap.

Central Free Lists for Small Objects

As mentioned before, we keep a central free list for each size-class. Each central free list is organized as a two-level data structure: a set of spans, and a linked list of free objects per span.

An object is allocated from a central free list by removing the first entry from the linked list of some span. (If all spans have empty linked lists, a suitably sized span is first allocated from the central page heap.)

An object is returned to a central free list by adding it to the linked list of its containing span. If the linked list length now equals the total number of small objects in the span, this span is now completely free and is returned to the page heap.