CS323 – Exercises
Week 5
21 March 2019

Problem 1: Address translation schemes

Draw a picture and give a description of:

A) A virtual address space possible with *segmentation* but not with *base and bounds*.

B) A virtual address space possible with *base and bounds* but not with *paging*.

C) A physical address space possible with *segmentation* with paging but not with *base and bounds*.

Problem 2: Logical and physical address

Consider a logical address space of 64 pages of 1024 words each, mapped onto a physical memory of 32 frames.

A) How many bits are there in the logical address?

B) How many bits are there in the physical address?

Problem 3: Segmentation

Consider a simple segmentation system with the following segment table:

Segment	Base	Length
0	219	600
1	2300	14
2	90	100
3	1327	580
4	1952	96

For each of the following logical addresses, determine the corresponding physical address or indicate if an interrupt is generated.

a) 0, 430
b) 1, 10
c) 2, 500
d) 3, 400
e) 4, 112

Problem 4: Memory allocation strategies

Given five memory partitions of 100KB, 500KB, 200KB, 300KB, 600KB (in order), how would the *first-fit*, *best-fit*, and *worst-fit* algorithms place processes of 212 KB, 417 KB, 112 KB, and 426 KB (in order)? Which algorithm makes the most efficient use of memory?

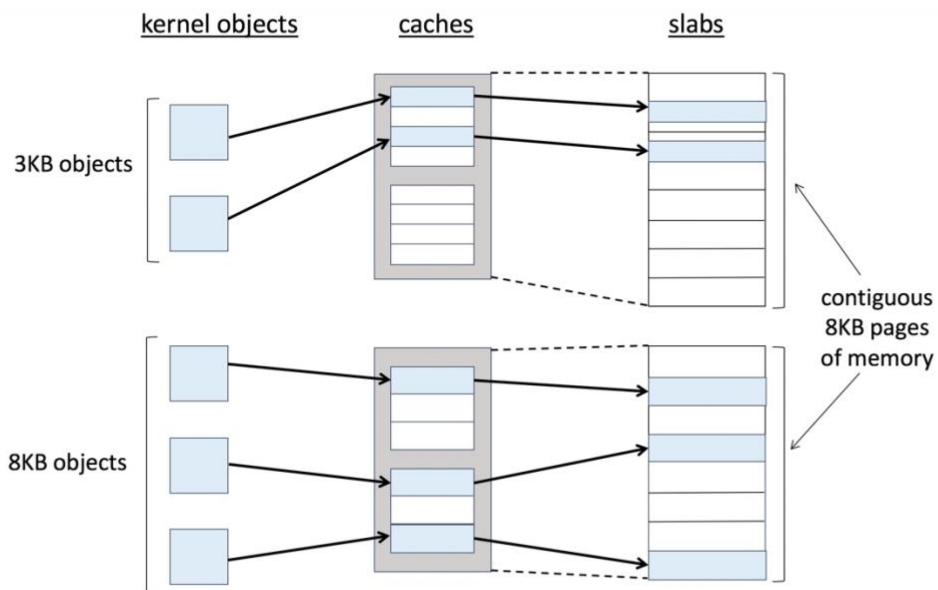
Problem 5: Paging

Why is it that, on a system with paging, a process cannot access memory it does not own?
How could the operating system allow access to other memory?

The slab allocator 1[1]

Another strategy for allocating kernel memory is known as *slab allocation*. A slab is made up of one or more physically contiguous pages. A cache consists of one or more slabs. There is a single cache for each unique kernel data structure — for example, a separate cache for the data structure representing process descriptors, a separate cache for file objects, a separate cache for locks, and so forth. Each cache is populated with objects that are instantiations of the kernel data structure the cache represents. For example, the cache representing process descriptors stores instances of process descriptor objects, and so forth.

The relationship among slabs, caches, and objects is shown in the figure below. The figure shows two kernel objects 3KB in size and three objects 8KB in size, each stored in a separate cache.



The slab-allocation algorithm uses caches to store kernel objects. When a cache is created, a number of objects — which are initially marked as free — are allocated to the cache. The number of objects in the cache depends on the size of the associated slab. For example, a 12-KB slab (made up of three contiguous 4-KB pages) could store six 2-KB objects. Initially, all objects in the cache are marked as free. When a new object for a kernel data structure is needed, the allocator can assign any free object from the cache to satisfy the request. The object assigned from the cache is marked as used.

Let's consider a scenario in which the kernel requests memory from the slab allocator for an object representing a process descriptor. In Linux systems, a process descriptor is of the type struct *task struct*, which requires approximately 1.7 KB of memory. When the Linux kernel creates a new task, it requests the necessary memory for the struct task struct object from its cache. The

[1] Operating Systems Concepts, 9th Ed. Silberschatz et. Al.

cache will fulfill the request using a struct task struct object that has already been allocated in a slab and is marked as free.

In Linux, a slab may be in one of three possible states:
1. Full. All objects in the slab are marked as used.
2. Empty. All objects in the slab are marked as free.
3. Partial. The slab consists of both used and free objects.

The slab allocator first attempts to satisfy the request with a free object in a partial slab. If none exists, a free object is assigned from an empty slab. If no empty slabs are available, a new slab is allocated from contiguous physical pages and assigned to a cache; memory for the object is allocated from this slab.

Problem 6: Slab allocator
What are the main benefits of the slab allocator? Explain why.

Problem 7: Slab allocator
The slab-allocation algorithm uses a separate cache for each different object type. Assuming there is one cache per object type, explain why this scheme doesn't scale well with multiple CPUs. What could be done to address this scalability issue?