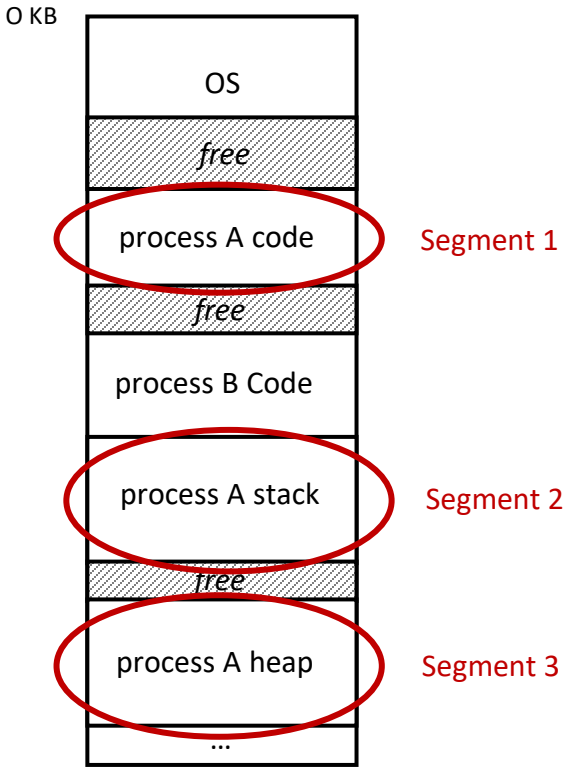
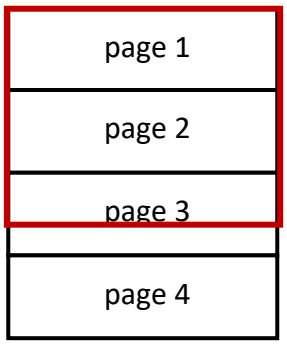
CS323 – Exercises
Week 5
21 March 2019

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*.

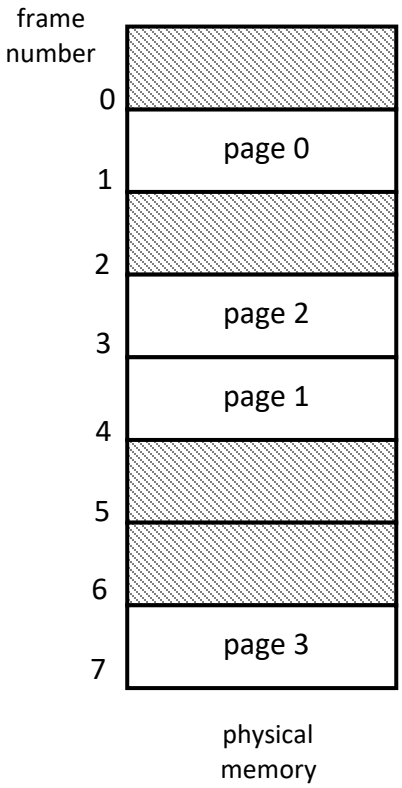
Answer:
A) Base and bounds can have only one segment, so an answer is anything with more than one segment, for e.g.:



B) A linear address space of size that is not a multiple of the page size, for e.g.:



C) Noncontiguous set of frames, for e.g.:



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?

Answer:

We will need 10 bits to uniquely address each of those 1024 addresses ($1024 = 2^{10}$).

A) A logical address space of 64 pages requires 6 bits to address each page uniquely ($64 = 2^6$), and there are 1024 (2^{10}) words in a page, so it will require 6+10=16 bits in total.

B) Physical memory has 32 frames and we need 5 bits to address each frame, requiring in total 5+10=15 bits.

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

Answer:
a. 219 + 430 = 649
b. 2300 + 10 = 2310
c. Illegal reference, trap to operating system
d. 1327 + 400 = 1727
e. Illegal reference, trap to operating system

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?

Answer:

1. First-fit

Note: Search the list of available memory blocks and allocate the first block that is big enough.

- 212 KB is put in 500K partition
- 417 KB is put in 600K partition
- 112 KB is put in 288K partition (new partition 288K = 500K - 212K)
- 426 KB must wait

2. Best-fit

Note: Search the entire list of available memory blocks and allocate the smallest block that is big enough

- 212 KB is put in 300K partition
- 417 KB is put in 500K partition
- 112 KB is put in 200K partition
- 426 KB is put in 600K partition

3. Worst-fit

Note: Search the entire list of available memory blocks and allocate the largest block.

- 212 KB is put in 600K partition
- 417 KB is put in 500K partition
- 112 KB is put in 388K partition
- 426 KB must wait

In this particular example, best-fit turns out to be the best.

Problem 5: Paging

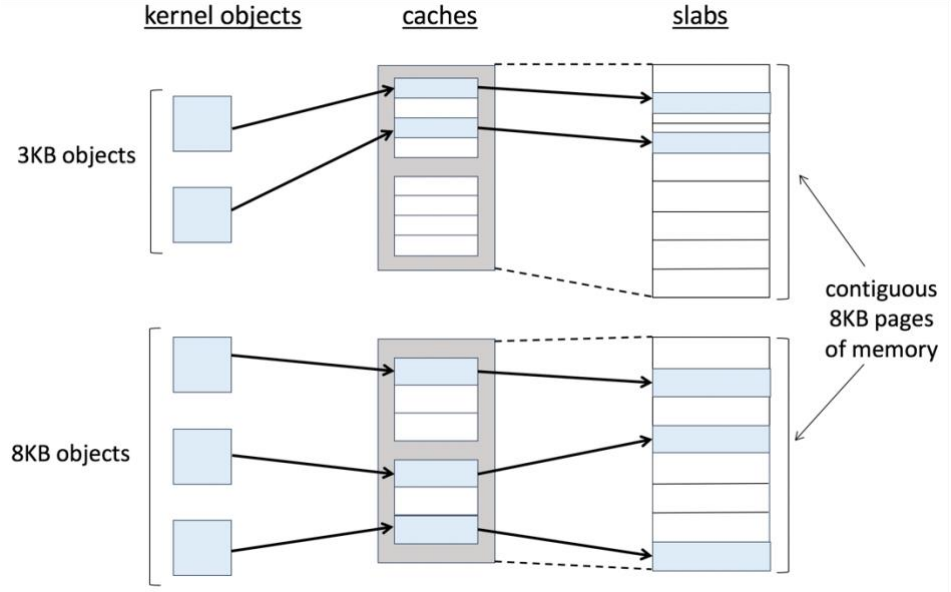
Why is it that, on a system with paging, a process cannot access memory it does not own? How could the operating system allow access to other memory?

Answer:

An address on a paging system is a logical page number and an offset. The physical page is found by searching a table based on the logical page number to produce a physical page number. Because the operating system controls the contents of this table, it can limit a process to accessing only those physical pages allocated to the process. The process cannot refer to a page it does not own because the page will not be present in the page table. To allow such access, an operating system simply needs to allow entries for non-process memory to be added to the process' page table. This is useful when two or more processes need to exchange data — they just read and write to the same physical addresses (which may be at varying logical addresses). This makes for very efficient inter-process communication.

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.

Answer:
The benefits are:
1. No memory is wasted due to fragmentation. Fragmentation is not an issue because each unique kernel data structure has an associated cache, and each cache is made up of one or more slabs that are divided into chunks the size of the objects being represented. Thus, when the kernel requests memory for an object, the slab allocator returns the exact amount of memory required to represent the object.
2. Memory requests can be satisfied quickly. The slab allocation scheme is thus particularly effective for managing memory when objects are frequently allocated and deallocated, as is often the case with requests from the kernel. The act of allocating — and releasing — memory can be a time-consuming process. However, objects are created in advance and thus can be quickly allocated from the cache. Furthermore, when the kernel has finished with an object and releases it, it is marked as free and returned to its cache, thus making it immediately available for subsequent requests from the kernel.

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?

Answer:
This has long been a problem with the slab allocator, poor scalability with multiple CPUs. The issue comes from having to lock the global cache when it is being accessed. This has the effect of serializing cache accesses on multiprocessor systems. Solaris has addressed this by introducing a per-CPU cache, rather than a single global cache.