Vika 13

Sturla Freyr Magnússon

Define virtual memory and describe its benefits.

Virtual memory is used by all major operating systems today. VM is the seperation of logical memory and physical memory so the logical address space does not need to map 1:1 to existing physical memory. This has benefits such as only the parts of a program necessary for execution needs to be in memory and therefore only parts of the process are swapped in and out instead of the whole process and requiring less I/O significantly increases speed.

Since only parts of the logical address space need to be in physical memory for each process the degree of multiprocessing is higher. The logical address space can be larger than the physical address space and this has the effect that programmers(& users) do not need to worry about the available physical memory.

Virtual Memory is typically implemented using demand paging.

Illustrate how pages are loaded into memory using demand paging.

Roughly comparable to idea of swapping however, instead of whole processes, pages are used

- Swaps single pages at a time
- Does not bring in a page until accessed (lazy paging)
  - Page currently not in physical memory will have in its page table entry the valid bit set to invalid: access to page triggers Page Fault Interrupt that is serviced by OS and will call pager routine to bring in page.
  - Only bring a page out when physical memory is needed for a new page
  - At each context switch, page table pointer of PMMU needs to be updated to point to the page table stored in the PCB of the particular process.
- Requires Paged MMU and CPU that supports restarting an instruction after a page fault interrupt occurred at exactly the same place and state where it was interrupted.

Procedure for Handling a Page Fault: Lecture Slides 9 page 8

Apply the optimal, FIFO, LRU and Second-Chance page-replacement algorithms.

- **optimal** Theoretical algorithm for a best, case scenario
- **FIFO** When replacing a page, choose the fram with the oldest page
  - Easy to implement
  - Oldest page might be frequently referenced
- **LRU** Replace page with the longest time period since it's last reference
  - Good approximation of optimal
  - No PMMU supports timestamping a page at each access in modern CPU's
- **Second-Chance** FIFO but page table entries have a reference bit that if set, moves the page from the head of queue to the tail
  - Easy to implement
  - Used by all major OS
  - Is essentially an approximation of LRU

  - Degenerates to FIFO if all pages have their reference bit set
- **Enhanced Second-Chance** Also has a modified bit giving 4 classes of replacement quality
  - Is a better approximation of LRU
  - Features required to implement are provided by all modern PMMU's
  - Periodic resetting of reference bits causes high overhead
  - May not be sufficient approximation of LRU(See Belady's Anomaly in lecture slides)

Describe thrashing the working set of a process, and explain how it is related to program locality.

A process is thrashing if it is spending more time paging than executing. The working set of a process is the set of memory pages that the process is actively using within a given time window. Program locality refers to the tendency of a process to access a relatively small and localized subset of memory locations during a particular phase of its execution. Program locality can be categorized into two types:

- Temporal locality: If a memory location is accessed, it is likely to be accessed again in the near future. This implies that recently accessed memory pages are more likely to be accessed again soon.
- Spatial locality: If a memory location is accessed, nearby memory locations are also likely to be accessed in the near future. This suggests that memory accesses tend to be clustered within specific regions of the address space.

When the working set of a process is well accommodated within the available physical memory, the process exhibits good locality. The system efficiently utilizes the available memory, and the process spends most of its time executing tasks rather than waiting for pages to be swapped

If the working set of the process cannot be accommodated within the available physical memory, the system starts swapping pages in and out frequently. As a result, the process loses its locality, leading to increased page faults and reduced performance.

To mitigate thrashing, operating systems use techniques like working set model and page replacement algorithms that take program locality into account.

Explain memory compression as alternative to paging out to storage devices

Memory compression can be a useful alternative to paging out to storage devices in situations where the benefits outweigh the costs, helping to improve system performance and manage memory efficiently. The main idea behind memory compression is to take advantage of redundancy in data stored in memory, compressing the data to free up space for other processes.

It's advantages over traditional paging are:
- Faster access because compressed pages are stored in physical memory rather than on disk.
- Reduced I/O overhead.
- Energy efficiency: accessing storage devices can consume more power during I/O operations.

Memory compression disadvantages to memory compression:
- CPU overhead: compression and decompression processes require additional CPU cycles,d- Limited compression ratio: depends on the compressibility of the data. On average: 2-3 compressed pages fit into one frame

Describe advanced applications of virtual memory, e.g. copy-on-write or demand loading

Demand loading

- If the infrastructure of demand paging is available, demand loading becomes possible.
- When a process is started, do not load whole binary file containing instructions just mark all pages initially as invalid.
  - At a page fault, load the according instructions for that page.
- Advantage: no unnecessary loading of instructions that might never get executed..- Disadvantage: resulting page faults lead to an overhead.

Prepaging (exact opposite of demand loading):

- Load all instructions into memory to avoid page faults.
- Advantage: reduced number of page faults.
- Disadvantage: unnecessary loading of instructions may occur.

Growing Heap & Stack

- Without virtual memory, reserving the right amount of memory is difficult
  - Too small: stack or heap overflow possible
  - Too huge: memory is wasted

With VM, we can just reserve the whole logical address space for a process and reserve a big amount of it for the stack and heap.

- While sufficient space for stack and heap is reserved in the logical address space only as many frames as currently required are used
- As stack and heap increase, just more pages are actually used.
- Overwriting shared libraries by stack or heap impossible as shared libraries serving as a sentinel or buffer in-between are read-only.(Illustrated on page 28 is lecture slides pack 9)

While paging avoids external fragmentation, internal fragmentation may still occur within the heap of a process: when releasing allocated heap space, holes in the logical address space of the heap occur.

Fork Using Copy-on-Write/Lazy-Copy

At fork, the child process gets an exact copy of the address space of the parent. This is only possible using a programmable MMU: The child's copy will be located at a different physical address. However, the child will also get a copy of all the parent's address references. These remain only valid, if a programmable MMU can be used to map the different physical addresses of the copy for the child to the same logical addresses that the parent process used.

However, copying the physical memory of the parent is slow. Instead ust map frames (instead of copying) of parent containing instructions and data into address space of child (shared memory).

However, when parent or child modifies its address space, the copy of the other process must not be modified! Mark shared pages as write-protected in page table entry: as soon as parent or child modify data, page fault interrupt occurs. Only then, just these frames are physically copied. ("Copy-on-Write"/"Lazy-Copy")

Explain management of kernel-internal memory

.Kernel memory is often allocated from a free-memory pool different from the list used to satisfy ordinary user-mode processes. There are two primary reasons for this:

1. The kernel requests memory for data structures of varying sizes, some of which are less than a page in size. As a result, the kernel must use memory conservatively and attempt to minimize waste due to fragmentation. This is especially important because many operating systems do not subject kernel code or data to the paging system.
2. Pages allocated to user-mode processes do not necessarily have to be in contiguous physical memory. However, certain hardware devices interact directly with physical memory—without the benefit of a virtual memory interface —and consequently may require memory residing in physically contiguous pages.

The Buddy System

The buddy system allocates memory from a fixed-size segment consisting of physically contiguous pages. Memory is allocated from this segment using a power-of-2 allocator, which satisfies requests in units sized as a power of 2. A request in units not appropriately sized is rounded up to the next highest power of 2

An advantage of the buddy system is how quickly adjacent buddies can be combined to form larger segments using a technique known as coalescing. The obvious drawback to the buddy system is that rounding up to the next highest power of 2 is very likely to cause fragmentation within allocated segments. In fact, we cannot guarantee that less than 50 percent of the allocated unit will be wasted due to internal fragmentation.

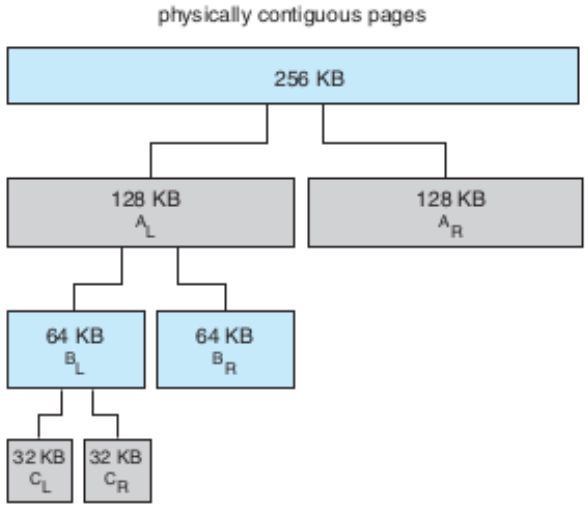

Figure 9.26 Buddy system allocation.

Slab allocation

A second 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 semaphores, and so forth.

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 contin-

guous 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.

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.

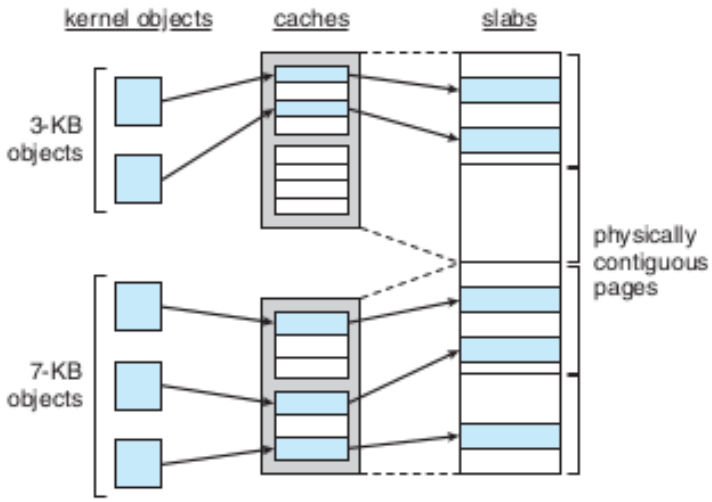

Figure 9.27 Slab allocation.

The slab allocator provides two main benefits:

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.