CS-323: OS Exercise Week 7 (Fall 2023)

November 1, 2023

Last name, first name: ____________________________________

1 Paging

(a) What are the implications of choosing a particular page size? Why not bigger? Why not smaller?

The OS allocates memory to processes at the granularity of a page. It is also the unit used for paging (moving frames between RAM and disk).
Having a bigger-sized page will increase internal fragmentation when a process requires space for a small object. This can be somewhat offset by having a userspace allocator which tracks free and allocated regions within pages (this is exactly what happens with the heap and `malloc`). Another consideration is for paging. Larger pages demand more I/O (disk) bandwidth when paging-out or paging-in.

On the other side, choosing a too small granularity will results in large number of pages for the same memory footprint which means more page-table entries and, thus, larger page tables. This is an overhead. Further, this increases TLB miss rates, as there are a larger number of translations to cache.

(b) Assume that the page offset is 10 bits and that a process has allocated 12000 bytes.

1. How many pages are used by this process?
   A 10-bit offset means $2^{10} = 1024$ byte pages.
   $\lceil \frac{12000}{1024} \rceil = 12$ pages are used by the process.
2. How many bytes are on the last page?
   The first 11 pages hold $11 * 1024 = 11264$ bytes. The last page, thus, holds $12000 - 11264 = 736$ bytes.
   Alternatively, $12000\%1024 = 736$ bytes.
3. What is the fraction of space wasted due to internal fragmentation?
   The process allocates $12 * 1024 = 12288$ bytes but uses 12000 bytes. The fraction of wasted space is $\frac{12288-12000}{12288} = 2.34\%$.

(c) A process generates 100 page faults per second. On each fault, a frame is paged-in from disk. What is the required disk bandwidth when the page size is:

1. 4KiB
   Each page fault results in a 4KiB read from disk. Thus, required bandwidth = 4KiB $*100$ per second = 400KiB/s.
2. 2MiB
   Required bandwidth = 2MiB $*100$ per second = 200MiB/s. Note that most HDDs support between $100 - 200$MB/s reads, so they will be overwhelmed. A solid-state drive should be able to manage.

Name: ____________________________

3. 1GiB

   Required bandwidth = 1GiB $*100$ per second = 100GiB/s. This can not supported by even solid-state storage!

(d) Paging makes it very easy for the OS to implement shared memory between separate processes. How could a page be shared between two processes?

Each process has its own virtual address space and, hence, its own page tables. Suppose process P_0 (with page tables PT_0) shares a page at virtual address VA_0 with process P_1 (with page tables PT_1 at virtual address VA_1). All that the OS has to do is to allocate one frame (at physical address PA, and map it in both page tables at the respective virtual addresses.
$PT_0(VA_0) = PT_1(VA_1) = PA$

2 Single-level/Flat page table

(a) Assume a system with a 32-bit physical/virtual address space, using page table entries of 4 bytes. Physical and virtual pages are 4 KiB in size. How many extra bits are available to store additional information (valid, accessed etc.) in each page table entry (PTE)?

Each page needs 12 bits to specify the offset due to page sizes being 2^{12} bytes. This leaves us with 20 bits for the frame number, because the physical address space is 32 bits. As we use 4 bytes (32 bits) for the page table entry, and the entry only needs to store the frame number, we have 12 bits for additional information.

(b) If we had a 40-bit address space, still using 4 bytes for each PTE, what would be the answer to the previous question?

Since we would have 28 bits for the frame numbers, we would be left with 4 bits for additional information.

(c) Assume an 8-bit virtual address space and an 11-bit physical address space. Pages are of size 32B (bytes). Fill out the following page table after these mentioned operations based on the information provided. The frames are zero indexed. The first bit of each page table entry is a valid bit (set to 1 if page is valid, default case is invalid).

1. A process stores a one byte value at virtual address 0x82 which ends up in frame 0xC.
2. A process stores a one byte value at virtual address 0xFF which ends up in frame 0x1.
3. A process stores a one byte value at virtual address 0x2F which ends up in frame 0x38.
4. Frame 0xC is swapped out to disk.
5. A process stores a one byte value at virtual address 0x4A which ends up in frame 0x1C.
6. A process stores a 64 byte array starting at virtual address 0xA0 which ends up in frames 0x17 & 0x3.
7. Page 0x001 is invalidated.

Virtual page number (VPN)	valid bit	Page frame number (PFN)

Name: ______________________________

Virtual page number (VPN)	valid bit	Page frame number (PFN)
0	0	
1	0	0x38
2	1	0x1C
3	0	
4	0	0x0C
5	1	0x17
6	1	0x03
7	1	0x01

Notes:

- We have a 5-bit offset, and 3-bit VPN. To get the VPN of a virtual address, we need to get the most significant 3 bits. e.g. for 0x82 this is 4, for 0xFF this is 7, for 0x2F this is 1 and for 0xA0 this is 5. A quick way to do this is take the 4 most significant bits (the first hex value), and shift one to the right, which is the same as halving via floor division.
- Swapping out to disk change the valid bit to false.
- Unused pages should have the valid bit set to false.
- In operation 6, we store more than one page of data. This means we will need two table entries (5 & 6).

(d) For each store operation, write the range of physical addresses that will be occupied.

As we have a 5-bit offset, i.e. each frame is 32 bytes, we need to shift the frame number left by 5 bits, or equivalently multiply the value by 32, to get the exact physical address of the start of the frame.

After getting the frame start address, we need to add the least significant 5 bits as offset from the virtual address to get the exact physical address. This is because offsets are the same in both virtual and physical memory.

This is a huge bitwise mishmash, so feel free to write down everything in binary if and convert to hex only at the end!

1. Stored on disk after swap, but may still be in original address!
   Frame start: 0xC (frame #) << 5 = 0x180.
   Offset (5 LSB from VA 0x82) = 0x2.
   Address = Frame start address + Offset = 0x182.
2. Frame start: 0x1 << 5 = 0x20.
   Offset (VA 0xFF) = 0x1F.
   Address = 0x03F.
3. Frame start: 0x38 << 5 = 0x700.
   Offset (VA 0x2F) = 0x0F.
   Address = 0x70F.
4. No store operation.
5. Frame start: 0x1C << 5 = 0x380.
   Offset (VA 0x4A) = 0xA.
   Address = 0x38A.
6. **First page:**
   Frame start: 0x17 << 5 = 0x2E0.
   Offset (VA 0xA0) = 0x0.
   Start: 0x2E0 (included)
   End: 0x2E0 + 0x20 (32 bytes) = 0x300 (excluded) = 0x2FF (included)
   Stored in [0x2E0, 0x2FF].
   Second page:
   Frame start: 0x3 << 5 = 0x060.

Offset (VA 0xA0 + 0x20 = 0xC) = 0x0.
Start: 0x060 (included)
End: 0x060 + 0x20 (remaining 32 bytes) = 0x080 (excluded) = 0x07F (included)
Stored in [0x060, 0x07F].

7. No store operation.

3 Multi-level page table

(a) What is the motivation to use multi-level page tables over a page table with a unique level?

The size of a single-level page table increases linearly with the size of the virtual address space. Thus, going from a 40-bit virtual address space to a 48-bit virtual address space (a 256x increase in VA space) leads to a corresponding 256x increase in page-table space.
In contrast, the size of a multi-level page table increases *almost* linearly with the actual memory allocated (unallocated regions do not need page-table entries). Thus, going from a 40-bit to 48-bit VA space will (probably) only require one extra page (or a few more, depending on the process) to hold another level of the page tables.
At a certain point, single-level page table will take too much space. See the example about 64-bit systems in the lecture slides.
On most desktop processors, the size of the VA space is 48-bits (256 terabytes), whereas most programs use a few gigabytes (say 16GiB) of actual RAM. At 4K pages and 8B PTEs, a single-level page tables would require 512GiB of space (!), whereas a multi-level page table requires much less.

(b) Where does the space saving come from?

Space savings occur when a sub-level page table does not hold any valid page. In this case, the parent PTE referencing this page table will not hold the address of the page but a bit specifying that the page table does not yet exist. For example, at the start, only the top level of the multi-level page table exists with every related page table marked as invalid. On the other hand, a single-level page table will always need to hold a PTE for every VA page.

If a process somehow used *all* of its virtual memory, then a multi-level setup would use more memory due to bookkeeping the extra levels. This is far from being the case in modern desktops, however, due to virtual memory being extremely large.

(c) Assume a 32-bit virtual/physical address space with a page size of 4KiB and 4 bytes per page table entry (PTE).
Note: Generally, you might be required to calculate the size of a PTE. You can do this from the size of the frame number, and the number of bits of metadata needed per page. See the next question.

1. How many levels are needed for the page table?

   4KiB pages means 1024 entries per page table, which is equivalent to 10 bits of the address space. We have 12 bits of offset due to the 4KiB page size, and have 20 bits left for the page number. We therefore need two levels to cover the page number (20 bits / 10 bits = 2).

2. How many pages and how much space is taken by the page table when no page is allocated by the process?

   The top level (root node) of the multi-level page table is fully allocated with each entry having the invalid bit set. We therefore have a single 4KiB page in use.

3. How many pages and how much space is occupied by a process when the first and last pages of its virtual address space are allocated?

   The top level table is fully allocated. In addition, at the 2nd-level, the first and last page tables are allocated, each having a single unique entry marked as valid. In total, we have a page for the top-level table, 1 page for each of the two 2nd-level tables, and the 2 pages for the actual data. We, therefore, have 5 pages of 4KiB each, 20KiB of memory in use.

4. Which entries of the multi-page table will be queried when accessing address 0x12345678?
   0x12345678 is 0001001000 1101000101 011001111000 in binary.
   As we have 10-bits page table, we access entry 0x48 (72 in decimal, 0001001000 in binary) in the top level page table (the most significant 10 bits). At the 2nd-level, we access the entry 0x345 (837 in decimal, 1101000101 in binary). The final 12 bits 0x678 are just the offset.
5. What is problematic with this list of multi-level page table requirements? 16-bit offset for pages, 32-bit virtual and 48-bit physical address space, 4B per page table entry.
   We have 32 bits for the page table entry but also 32 bits to specify the frame number. This leaves us with no bits to specify additional information. We can not therefore place an indication that a page table at a sublevel in not yet allocated. This makes the multi-level page table useless. A fix could be to augment the size of the page table entries to 8B for example.

(d) Calculate the size of the PTE in each case below:

1. 32-bit physical address space, 4KiB pages
   12 bit offset, 20 bit PFN. The PTE will probably be 4bytes (32-bits), allowing for 12 bits of metadata.
2. 48-bit physical address space, 64KiB pages
   16 bit offset, 32 bit PFN. Including metadata, the PTE will not fit in 4 bytes. Therefore, PTE will likely be 8B.

4 PTE Bits in x86_64

A page table entry (PTE) on a modern x86_64 processor contains the bits listed below. For each item, explain what the bit means, and what the OS can use these bits for.

1. Valid (V) or Present (P) This bit states whether the PTE is valid. If not set, the other permissions are ignored and any access generates a fault. The OS sets this bit for all frames present in RAM.
   Otherwise, the OS can reset this bit for any pages it plans to page-out soon, but has not done yet. If the process accesses this page, it still generates a fault. However, to fix the fault, the OS just needs to set the PTE bit, not read in the page from disk.

2. Read,Write,Execute (RWX) These bits encode permissions for the data stored in the page. Read pages (those with R bit set) allow loads, Write pages allow stores, and eXecutable pages allow the data to be executed. The OS uses these to enforce access to various sections. The .text section holds code, and therefore have RX bits set. The heap and stack store data, not code, and have RW bits set. The global constants are held in pages with only the R bit set. Remember you can view these mappings by checking `/proc/<PID>/maps` (from Tutorial 5).

   Extra: These permissions can also be used to create *copy-on-write* frames. Such frames are shared between processes, or even within the same process, and save RAM space. For example, all zero pages (pages containing all zeros) often map to the same frame. A fresh copy with a new frame is made only after the process attempts to write on the page.

3. Accessed (A) This bit is set by the MMU when the page is accessed (read, write or executed), and can be reset by the OS. The OS can use this to detect which pages are used often, and which ones are not. It does this by periodically resetting the accessed bits for all pages. It can later check, after a while, which pages were accessed by the process, since the accessed-bits in their PTEs will be set.

4. Dirty (D) This bit is set by the MMU when the page is written to, and can be reset by the OS. When paging-out a file-backed page that is not dirty, the OS can skip the write to disk (an identical copy already exists on disk). Otherwise, the dirty page has to be rewritten.

5. User (U) This bit is set for pages accessible from userspace, and is zero for pages only accessible by the kernel. The OS uses this to reserve some parts of the virtual memory space for exclusive use by the kernel. In Linux, the upper half of the memory space is not available for userspace.

Name: ______________________________

5 HugePage

Having a unique page size might not be ideal for each application. Linux supports mixing different page sizes via HugePage. You can read more about them here.

(a) How many 4KiB pages are needed to store 1GiB of data?

$2^{30}/2^{12}$ = 262,144 pages

(b) What is problematic for the TLB when accessing this blob of data linearly?

The number of entries that a TLB can store is usually in the thousands, it cannot store a quarter million entries. The TLB will therefore constantly face cache misses which will slow down program execution.

(c) How would you implement handling bigger page size in a multi-level page table in a transparent way?

We can implement bigger page size by storing the frame number at a higher level. As we keep the addressable structure the same, it means that we can only have pages with a size a multiple of the original size: $2^k * page\_original\_size$, where k is the number of bits used for the page table level. Essentially, we will be *collapsing* multiple lower levels into larger pages.

(d) Assume three-level page table in a 36-bit VA, 8B PTE and 4KiB pages. What would be the size of the huge pages if we store the frame number at the Page Global Directory (top level)? At the Page Middle Directory (middle level)?

We have 12 bits pages, this leaves us with 3 level of 8 bits each. Having a huge page at the PGD level results in $2^8 * 2^8 * 2^{12}$ = 256MiB pages. At the PMD we $2^8 * 2^{12}$ = 1MiB pages.

(e) What additional benefit do we gain by having a more shallow tree for certain pages?

In case of a TLB miss, translation time will be shorter because of having to go through fewer levels to find the PFN for the translation.