EECE.4810/EECE.5730: Operating Systems

Spring 2018

Extra Credit Homework Solution

1. *(30 points)* ***Page table organization***

Given a system using 8 KB pages, a 48-bit virtual address, and 8 GB of physical memory, determine the amount of space required to store each of the following types of page tables. Assume each page table entry (PTE) requires 4 bytes unless the problem states otherwise.

Solution: To solve these problems, consider the following general information:

- The virtual address space is 2^{48} bytes = 256 TB
- Since each page is 8 KB, that virtual address space is divided into $2^{48} / 2^{13} = 2^{35}$ pages
- Each PTE is $4 = 2^2$ bytes unless otherwise stated

a. *A basic, one-level page table that covers the entire virtual address space.*

A one-level page table has one PTE for every page in the virtual address space, so the table would take up $2^{35} * 2^2 =$ **2^{37} bytes = 128 GB**

b. *A two-level page table, assuming the currently running process is using the entire first half of its virtual address space.*

- ***NOTE:*** *Typically, in multi-level paging, the upper address bits are split evenly to index into each level of the page table. For example, given a 32-bit address with a 10-bit page offset, in a two-level page table, the remaining 22 address bits are split so 11 bits are used for the first-level table and 11 bits are used for the second-level tables.*

  However, if the number of remaining bits are odd, assume the extra bit is used to index the first-level table. So, in the example above, if the page offset were 11 bits, not 10, the remaining 21 address bits would be split such that 11 bits were used for the first-level table and 10 bits were used for the second-level tables.

As noted above, the upper address bits are split evenly to index into each level of the table, with the uppermost bits used to access the first-level table and the remaining bits accessing the appropriate second-level table.

Since each page is 8 KB = 2^{13} bytes, the page offset is 13 bits, leaving the upper 48 – 13 = 35 address bits to index into the page table. They're split such that 18 bits index into the first-level table and 17 bits index into a second-level table.

1b. (continued)

Therefore:

- The number of index bits helps determine the first-level table size:

  2^{18} entries * 2^2 bytes per entry = 1 MB

- That table contains 2^{18} ~~/ 2^2 = 2^{16}~~ entries, which implies there are 2^{18} second-level tables.
- Having the process use half of its virtual address space implies that half of the second-level tables are in use: 2^{18} / 2 = 2^{17} second-level tables
- Again, based on the number of index bits, we can find the size of each second-level table: 2^{17} entries * 2^2 bytes per entry = 512 KB
- So, the total required storage space is:

  (2^{20} bytes for 1st-level table) + (2^{19} bytes per 2nd-level table) * (2^{17} second-level tables) =

  2^{20} + 2^{36} bytes = 1 MB + 64 GB = 68,720,525,312 bytes

c. *A hash table in which:*

  i. *Each entry in the hash table contains a single 32-bit pointer to a chain of PTEs*

  ii. *Each PTE consists of 8 bytes—4 bytes for the typical PTE content plus a 4 byte pointer to the next PTE in the chain.*

  iii. *The number of entries in the hash table (not necessarily the number of entries in each chain) is 1/16 of the number of entries in the one-level page table*

I didn't give you enough information to solve this part of the problem, as I didn't tell you how much of the address space is being used. You can definitely determine the size of the hash table:

- The number of hash table entries = # one-level PTEs / 16 = 2^{35} / 2^4 = 2^{31} entries. Table size = 2^{31} entries * 2^2 bytes per hash table entry = **2^{33} bytes** = **8 GB**. (In practice, the hash table would probably be much, much smaller than this)

If you use the same assumption as in part (b) (that the process is using half of its available virtual address space), then the process is using 2^{35} / 2 = 2^{34} pages. Since each PTE is 8 bytes, the chained PTEs take up (2^{34} PTEs) * (2^3 bytes per PTE) = 2^{37} bytes = 128 GB, meaning the total amount of required storage would be 128 GB + 8 GB = **136 GB**.

d. *An inverted page table, with one PTE for each physical frame.*

The 8 GB = 2^{33} bytes of physical memory is divided into 2^{33} / 2^{13} = 2^{20} frames, so an inverted page table would take up (2^{20} PTEs) * (2^2 bytes per PTE) = **2^{22} bytes = 4 MB**.

2. (12 points) ***Page replacement***

Say the currently running process has 16 active pages, P0-P15. P0 and P8 both have their reference bits set to 0, while all other pages have their reference bits set to 1. If the operating system uses the clock algorithm for page replacement, the pages are ordered numerically around the "clock" (P0 is first, P1 is second, etc.), and the "clock hand" currently points to P4, what are the first four pages to be replaced, assuming none of the currently active pages are referenced before four replacements are required? ***Explain your answer for full credit.***

Solution: Recall that, in the clock algorithm, the list of pages is searched until one is found for which the reference bit is 0. As each page is checked, if its reference bit is 1, the page is not evicted, but its reference bit is cleared (set to 0) to indicate that it may be a candidate for eviction the next time a replacement is needed.

The initial state of the "clock" is:

- 1st page: P4, reference bits set to 0: P0 & P8, all others (P1-P7, P9-P15) set to 1

So, let's look at each page replacement:

Replacement 1 (new page = "P16"):

- The "clock" starts at P4 and visits page in order until one with a reference bit = 0 is found
- Counting up from P4, the first eviction candidate is **P8**—that page is evicted and replaced by P16, and the "clock hand" moves to the next page (P9).
- P4-P7 are all visited, so their reference bits are 0.
- New clock state:
  - 1st page: P9, ref bits set to 0: P0, P4-P7, all others (P1-P3, P16, P9-P15) set to 1
  - Page order around clock: P0-P7, P16, P9-P15

Replacement 2 (new page = "P17"):

- The clock starts at P9 and goes through pages in increasing order
- Since P9-P15 all have reference bits = 1, after visiting P15, the clock hand will return to P0. That page's reference bit = 0, so **P0** is evicted and replaced by P17, and the clock hand moves to P1.
- Reference bits for P9-P15 are all set to 0.
- New clock state:
  - 1st page: P1, ref bits set to 0: P4-P7, P9-P15, all others (P17, P1-P3, P16) set to 1
  - Page order around clock: P17, P1-P7, P16, P9-P15

Replacement 3 (new page = “P18”):

- The clock hand starts at P1 and moves until it finds **P4**, which is evicted and replaced with P18. The clock hand moves to P5.
- Reference bits for P1-P3 are all cleared
- New clock state:
  - 1st page: P5, ref bits = 0: P1-P3, P5-P7, P9-P15, all others (P17, P18, P16) set to 1
  - Page order around clock: P17, P1-P3, P18, P5-P7, P16, P9-P15

Replacement 4 (new page = “P19”):

- This eviction is the easiest—the clock hand is pointing to a page, **P5**, for which the reference bit is 0. That page is replaced with P19, and the clock hand moves to P6.

So, the pages evicted, in order, are **P8, P0, P4, and P5**.

3. (18 points) ***Virtual memory***

A portion of the currently running process's page table is shown below:

Virtual page #	Valid bit	Reference bit	Dirty bit	Frame #
7	1	1	1	3
8	0	0	0	--
9	1	0	1	4
10	0	0	0	--
11	1	1	0	0
12	1	1	0	1

Assume the system uses 20-bit addresses and 16 KB pages. The process accesses four addresses: 0x25FEE, 0x2B149, 0x30ABC, and 0x3170F.

Determine (i) which address would cause a page to be evicted if there were no free physical frames, (ii) which one would mark a previously clean page as modified, if the access were a write, and (iii) which one accesses a page that has not been referenced for many cycles. ***For full credit, show all work.***

Solution: First, note that 16 KB = 2^{14} byte pages imply a 14-bit offset. Therefore, in each virtual address, the upper 6 bits hold the page number, and the lower 14 bits hold the page offset.

Each virtual address is translated below, with the page number underlined:

- 0x25FEE = 0010 01 01 1111 1110 1110 à page number 9
- 0x2B149 = 0010 10 11 0001 0100 1001 à page number 10
- 0x30ABC = 0011 00 00 1010 1011 1100 à page number 12
- 0x3170F = 0011 00 01 0111 0000 1111 à page number 12
  (I didn't intend to duplicate a page number, but that's what happens when you write exam questions after midnight)

Now, to answer the questions above:

(i) For an access to cause an eviction, it must be to a page not currently in physical memory, as shown by the valid bit being 0. From the group above, that would be the access to page 10, or address **0x2B149.**

(ii) For an access to mark a previously clean page as modified, the dirty bit for that page must be 0. From the group above, either access to page 12 would qualify: address **0x30ABC** or **0x3170F**. Note that page 10 does not count because it is not a valid access.

(iii) Pages that have not been referenced for many cycles have their reference bits set to 0. From the group above, the access to page 9 qualifies—address **0x25FEE**. Again, the access to page 10 does not count because it is not a valid access.

4. (30 points) ***File systems: organization***

This problem asks you to assess file access time for each of the three file systems (FAT, FFS, NTFS) discussed in class. Assume, in all cases:

- *Each disk block is 8 KB*
- *The time required to access a new disk block is 20 ms*
- *Once a disk block is accessed for the first time, it remains cached in main memory, and every access to the cached block after the first one will take 2 ms.*
- *The address size is 32 bits.*

Determine the worst case access time for a single read operation to an undetermined word of data in a 16 MB file if the file system used is (a) FAT, (b) FFS, (c) NTFS.

- *For NTFS, assume all records in the MFT fit in a single disk block, and the file in question requires 4 MFT records to store all necessary metadata.*

Solution: In all cases, note that a 16 MB file divided into 8 KB blocks will require $2^{24} / 2^{13} = 2^{11}$ disk blocks. Assessing each of the file systems:

FAT: In a file allocation table, the master file table (MFT) is stored on a single disk block, and each MFT entry contains a pointer to the entry indexing into the next block of the given file. The worst case access time therefore requires traversing the entire linked list of MFT entries and then reading the very last block of the file.

So, the worst case access time is:

(20 ms for the first MFT entry) +
(2 ms for each additional entry) * (2^{11}- 1 additional entries) +
(20 ms to access the actual data) =
20 ms + (2 ms) * (2047) + 20 ms = **4134 ms = 4.134 sec**

FFS: FFS uses an asymmetric tree of index blocks, with blocks in a small file directly accessible from the inode and up to three levels of indirection used for larger files. Each indirect block contains a set of 32-bit pointers (based on the address size) to either another index block or the actual data. The worst case access time depends on the number of levels of indirection used, and therefore the file size.

In this case, note that each 8 KB index block contains $2^{13} / 2^{2} = 2^{11} = 2048$ different pointers. So:

- The 12 direct pointers in an FFS inode support file sizes up to 12 * 8 KB = 96 KB.
- The singly indirect block supports file sizes up to $2^{11} * 2^{13} = 2^{24}$ bytes = 16 MB—exactly the size of the file in this example.
- Therefore, accessing a block in this file (in the worst case) requires 3 accesses—one to the inode, one to the indirect block, and one to the actual data block. So, the worst case access time is 3 * 20 ms = **60 ms**.

4 (continued)

NTFS: The problem states that the file would require 4 MFT records when stored using NTFS. What isn't immediately obvious from our discussion of NTFS is how those records can be accessed—it appears from the figure in the lecture slides that, sometimes, the first record points to all other records for a file. But, in other cases, it appears a linked list of MFT records is used.

I'd therefore accept any reasonably well thought-out answer to this problem, but the linked list would be slower than having the first record point to all others, making the worst case time:

(20 ms to get first MFT record) + 3 * (2 ms to get each additional record) +
(20 ms for actual data block) =
20 ms + 6 ms + 20 ms = **46 ms**

5. (12 points) ***File systems: reliability***

a. (6 points) Explain why methods for ensuring reliability in file systems center on operations that write a single sector.

Solution: Writing a single sector is the only atomic operation disks support, so methods for reliability in file systems—preventing loss of data—aim to make multi-step changes to the file system reliant on that one atomic operation.

b. (6 points) A Linux variant called TxOS, developed at the University of Texas, supports transactions with shadowing by decomposing inodes into two parts: a header that contains infrequently modified data about each file, and a data component holding fields that are commonly modified by system calls. The header contains a pointer to the related data component, and the data component contains a pointer to the header.

Explain how this inode organization makes it relatively easy to implement shadowing for changes to a file's metadata.

Solution: Shadowing requires the file system to create a "shadow copy" of any data to be changed while maintaining a pointer to the current version. Changes are made to the shadow copy and committed by changing the pointer from the current version to the shadow copy.

By splitting the metadata into a header and data section, creating a shadow copy of the frequently changed data without copying the header is simple. The pointer within the header section can be pointed to the shadow copy of the data once changes to it are complete.