EECE.4810/EECE.5730: Operating Systems

Spring 2019

Exam 3 Solution

1. *(38 points)* ***General memory management***

a. *(20 points) Write the function with the prototype below, which could be used in base & bounds or segmented address translation to return the starting address of the hole that best fits a memory request of size* `reqSize`*:*

```
unsigned bestFit(int reqSize);
```

The function uses this structure for each node in the free space (hole) list:

```
typedef struct FSN {
   unsigned addr;         // Starting address of hole
   int size;              // Hole size
   struct FSN *next;      // Address of next node in list
} FreeSpaceNode;
```

Assume you have a global pointer (`FreeSpaceNode *first`*) to the first node in the free space list. Since the pointer is a global variable, it does not need to be passed to the function. Assume there is at least one node in the free space list.*

This function should use best fit allocation to choose a hole in the free space list, update the node representing that hole to reflect the amount of space used, and return the address of the hole. For example, if you request 20 bytes from a 100-byte hole that starts at address 0x1000, the hole's new starting address is 0x1014 (0x14 = 20) and its size is 80 bytes; the function returns 0x1000.

Your function should also account for the special case that the requested space completely fills the hole, in which case that node should be removed from the list, not modified. Assume you have a function with the prototype `remove(FreeSpaceNode *n)` *to remove a node from the list. So, for example, if* `ptr` *points to a node to remove, calling* `remove(ptr);` *modifies the free space list to remove that node. You only need to recognize the conditions under which you call* `remove()` *and call it appropriately—don't worry about the details of that function.*

The comments in the function outline how it should behave. ***Its solution is on the next page.***

1a (continued)

```
unsigned bestFit(int reqSize) {
   // Variable declarations/initialization
   FreeSpaceNode *n = first;   // Pointer to list node
   FreeSpaceNode *best = NULL; // Pointer to hole with "best fit"
   unsigned retAddr;           // Address of hole
   int minDiff = 0x7FFFFFF;    // Difference in reqSize, hole
                               // Initialized to large pos value
   int diff;                   // Temporary difference value

   // Traverse free space list and find hole with best fit
   while (n != NULL) {
      diff = n->size - reqSize;
      if (diff >= 0 && diff < minDiff) {
         best = n;
         minDiff = diff;
      }
   }

   // If no fit found, return 0
   if (best == NULL)    return 0;

   // Otherwise, update node accordingly
   retAddr = best->addr;
   best->size = best->size - reqSize;
   best->addr = best->addr + reqSize;

   // One special case—if hole is completely filled, remove node
   //  (but make sure you've saved your return address first!)
   //  (You don't need a definition for the "remove" function)
   if (minDiff == 0) remove(best);

   // Return address of hole
   return retAddr;
}
```

1 (continued)

b. (10 points) For each of the three types of address translation we discussed (base & bounds, segmentation, paging), explain what type(s) of fragmentation (internal or external) could occur in each and explain how.

Solution: Each address translation type is discussed below:

- In base & bounds, only external fragmentation occurs, since the amount of memory allocated can match the amount of requested space and no space will be wasted inside the allocated area. External fragmentation will occur either because satisfying multiple space requests will eventually leave a hole too small for any address space, or because deallocating space allocated to a finished process might leave too small a hole as well.
- In segmentation, only external fragmentation occurs, for reasons that are similar to what's described above. The space requests are smaller, since you're allocating segments, not entire process address spaces, but the issues are the same.
- In paging, only internal fragmentation occurs, since space is always allocated in fixed-size blocks and there will therefore never be a hole that is "too small" to fill with a new frame. However, a process requiring less than a full frame will still be required to allocate an entire frame, thus leading to wasted space inside that frame.

c. (4 points) Describe one benefit of segmentation over base and bounds as an address translation scheme.

Solution: Segmentation is more flexible than base and bounds, thus making it less likely to lead to external fragmentation. Also, dividing the address space into multiple segments makes it possible to share partial address spaces among multiple processes. (Other answers may apply.)

d. (4 points) Describe one benefit of segmentation over paging as an address translation scheme.

Solution: Since each segment is allocated as a contiguous range of addresses, segmentation makes it easier to allocate and access large contiguous ranges of physical memory. Paging requires these large ranges to be split into multiple pages/frames, while a single segment could be used for one large range of addresses. (Other answers may apply.)

2. *(42 **+ 9** points)* ***Paging***

a. *(18 points) Explain what conditions would create the fastest possible translation for each of the page table organizations we discussed. (For example, if a tree-based page table existed, the fastest possible translation would happen if the necessary translation was at the root of the tree and could therefore be found in a constant amount of time.)*

You should not give a numeric answer, but may provide an example to explain your answer.

i. *Two-level page table*

Solution: The access time for any entry in a two-level page table is the same—one access to the first-level table to find the location of the correct second-level table, then an access to that second-level table to perform the translation.

ii. *Hashed page table using chaining*

Solution: The best case would be that the hashed table had no collisions, at least not for the translation you're searching for, and so that value is found in a constant amount of time.

iii. *Inverted page table*

Solution: Since inverted page tables must be searched to find an entry with a matching page number, the best case would be that the desired PTE is the first one searched.

2 (continued)

b. (12 points) This problem involves a process using the page table below:

Virtual page #	Valid bit	Reference bit	Dirty bit	Frame #
0	1	1	1	3
1	0	0	0	--
2	1	0	1	4
3	0	0	0	--
4	1	1	0	0
5	1	1	0	1

Assume the system uses 16-bit addresses and 8 KB pages. The process accesses four addresses: 0x1234, 0x4755, 0x6A13, and 0xB0A9.

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 of all, note that 8 KB = 2^{13} B pages require an 13-bit page offset, and therefore allow for a 3-bit virtual page number within the 16 bit address.

The next step is to identify the virtual page numbers within each of the given addresses:

- 0x1234 = 000$\underline{1}$ 0010 0011 0100_2 → page 0
- 0x4755 = 010$\underline{0}$ 0111 0101 0101_2 → page 2
- 0x6A13 = 011$\underline{0}$ 1010 0001 0011_2 → page 3
- 0xB0A9 = 101$\underline{1}$ 0000 1010 1001_2 → page 5

By referring back to the page table, we can see that the answers are as follows:

(i) Address causing an eviction if there are no free frames: an invalid page that must be brought in from disk → page 3 → 0x6A13

(ii) Address marking a previously clean page as modified if access is write: valid page dirty bit = 0 → page 5 → 0xB0A9

(iii) Address accessing a page that hasn't been accessed for many cycles: valid page with reference bit = 0 → page 2 → 0x4755

2 (continued)

c. *(12 points) Say the currently running process has 16 active pages, P0-P15. P1, P3, P5, and P7 all 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 P2, how many page faults will occur before P4 is replaced, assuming none of the currently active pages are referenced before P4 is replaced?* ***Explain your answer for full credit.***

Solution: To solve this problem, consider how the "clock" changes with each page fault, as described below. In each state, the reference bit value is shown for each page, so "P0 = 1" means the reference bit for page 0 is 1. Pages are shown in the order they're considered for replacement, starting with the current "clock hand" (shown in bold). Pages starting with "NP" are new pages brought in after a page fault.

- Initial state: **P2 = 1**, P3 = 0, P4 = 1, P5 = 0, P6 = 1, P7 = 0, P8-15 = 1, P0 = 1, P1 = 0
- Page fault #1: after P2, the first page with a reference bit = 0 is P3, which is replaced. P2's reference bit is cleared, and the clock hand moves to P4.
    - New clock state: **P4 = 1**, P5 = 0, P6 = 1, P7 = 0, P8-15 = 1, P0 = 1, P1-P2 = 0, NP0 = 1
- Page fault #2: after P4, the first page with a reference bit = 0 is P5, which is replaced. P4's reference bit is cleared, and the clock hand moves to P6.
    - New clock state: **P6 = 1**, P7 = 0, P8-15 = 1, P0 = 1, P1-P2 = 0, NP0 = 1, P4 = 0, NP1 = 1
- Page fault #3: after P6, the first page with a reference bit = 0 is P7, which is replaced. P6's reference bit is cleared, and the clock hand moves to P8.
    - New clock state:, **P8 = 1**, P9-15 = 1, P0 = 1, P1-P2 = 0, NP0 = 1, P4 = 0, NP1 = 1, P6 = 0, NP2 = 1
- Page fault #4: after P8, the first page with a reference bit = 0 is P1, which is replaced. Reference bits are cleared for P8-P15 and P0, and the clock hand moves to P2.
    - New clock state: **P2 = 0**, NP0 = 1, P4 = 0, NP1 = 1, P6 = 0, NP2 = 1, P8-15 = 0, P0 = 0, NP3 = 1
- Page fault #5: Since P2's reference bit is 0, that page is replaced. The clock hand moves to NP0
    - New clock state: **NP0 = 1**, P4 = 0, NP1 = 1, P6 = 0, NP2 = 1, P8-15 = 0, P0 = 0, NP3-4 = 1
- Page fault #6: afer NP0, the first page with a reference bit = 0 **is P4—the page the problem asks about—which is replaced.** NP0's reference bit is cleared, and the clock hand moves to NP1.

At this point, we don't need to write the new state—we've found that P4 is replaced **after 5 page faults (on the 6th page fault).**

2 (continued)

*d. (9 points, **EECE.5730 only**) Assume you have a system using 48-bit virtual addresses, 32-bit physical addresses, 4 KB pages, and a multilevel page table. Assume a page table entry in each lowest level table (in a 2-level page table, for example, the second level would be the lowest level) uses 4 bytes, and each entry in the upper level table(s) contains a single physical address.*

*If you want each table (1st level table, each 2nd level table, and so on) to be the same size, and that size should be small enough to fit inside a single physical frame, what is the minimum number of levels the multilevel table must have? **Show all of your work for full credit.***

Solution: Consider the following:

- First, let's determine some information about the size of each table:
    - Each entry in an upper level table contains a single physical address, which is 32 bits or 4 bytes—the same size as a PTE in a lowest level table. So, for all tables to be the same size, they must hold the same number of entries.
    - If every table holds the same number of entries, each table will require the same number of address bits. So, the upper address bits of the virtual address must be divided evenly to index into each of the different page table levels.
    - Since each table must fit inside a single 4 KB page, each table can have at most (4 KB) / (4 bytes/entry) = 1K entries = 2^{10} entries. That means the index for each page table level can use at most 10 bits.
- Now, let's talk about determining the number of levels based on how the virtual address breaks down:
    - The page offset is based on the page size: 4 KB = 2^{12} B, so the offset is 12 bits.
    - A 12 bit offset leaves 48 – 12 = 36 bits to index into the page table.
    - Dividing 36 bits evenly so that at most 10 bits index into each level requires us to divide by at least 4. (36 / 3 = 12, 36 / 4 = 9).

Therefore, our multilevel table must have **at least 4 levels** to satisfy the problem requirements.

3. *(20 + 6 points) File systems*

a. *(8 points) Consider two of the file systems we discussed, FFS and NTFS. Describe a case in which the NTFS index scheme would allow you to access file data more quickly than FFS would for a comparably sized file, and describe a case in which FFS would allow you to access file data more quickly for a comparably sized file.*

Solution: NTFS would typically allow for faster access to data than FFS would in a very small file, since the data is stored in the NTFS record for that file. All data accesses in FFS require at least one level of indirection—accessing the inode to find a direct pointer to the desired data.

FFS will allow for faster access than NTFS if the file is large enough and its blocks are split up enough to require a linked list of multiple NTFS records, particularly if the number of records in that linked list is greater than the number of indirect blocks you'd access in FFS. After all, each indirect block used for indexing in FFS is effectively a linked list node.

b. *(6 points) We discussed two schemes for free space management in file systems, bitmaps and linked lists. Explain one benefit of each of these schemes.*

Solution: The major benefit of using a bitmap to manage free space is its simplicity. Bitmaps also make it easier to find and allocate consecutive disk blocks.

Linked lists are much more space-efficient than bitmaps for managing free space—each free disk block need only store the address of the next free block.

c. *(6 points) Explain what a transaction is, and also explain how the two main methods we discussed for implementing transactions provide reliability in a file system.*

Solution: A transaction is a group of operations that appear to be atomic (all are completed or none are completed). Each transaction typically ends with a "commit" step that ensures all prior steps are now permanently seen as complete. The two main implementations of transactions we discussed are:

- Shadowing: Changes to the file system are made to a shadow copy of the file data and metadata, and the changes are only "committed" to the file system when the in-place file system is updated to point to the shadowed data.
- Logging: Planned changes are written in order to a log file, with a final sector within the log written to commit those changes. Committed changes can later be "replayed" to actually update the file system itself.

d. *(6 points, **EECE.5730 only**) Explain how, in some cases, one volume can span multiple disks, and how, in other cases, one disk can contain multiple volumes.*

Solution: A volume is a disk partition formatted with a file system. If one volume spans multiple disks, the directory structure tracks which disk contains which files and maps accesses to those files appropriately. If a disk is split into multiple partitions, each partition can contain a separate file system, thus allowing one disk to contain multiple volumes.