CS372: Solutions for Homework 9

Problem 1

In a 32-bit machine we subdivide the virtual address into 4 segments as follows:

10-bit	8-bit	6-bit	8 bit

We use a 3-level page table, such that the first 10-bit are for the first level and so on.

1. What is the page size in such a system?
2. What is the size of a page table for a process that has 256K of memory starting at address 0?
3. What is the size of a page table for a process that has a code segment of 48K starting at address 0x1000000, a data segment of 600K starting at address 0x80000000 and a stack segment of 64K starting at address 0xf0000000 and growing upward (like in the PA-RISC of HP)?

Solution:

1. The page field is 8-bit wide, then the page size is 256 bytes.

2. Using the subdivision above, the first level page table points to 1024 2nd level page tables, each pointing to 256 3rd page tables, each containing 64 pages. The program's address space consists of 1024 pages, thus we need we need 16 third-level page tables. Therefore we need 16 entries in a 2nd level page table, and one entry in the first level page table. Therefore the size is: 1024 entries for the first table, 256 entries for the 2nd level page table, and 16 3rd level page table containing 64 entries each. Assuming 2 bytes per entry, the space required is 1024 * 2 + 256 * 2 (one second-level paget table) + 16 * 64 * 2 (16 third-level page tables) = 4608 bytes.

3. First, the stack, data and code segments are at addresses that require having 3 page tables entries active in the first level page table. For 64K, you need 256 pages, or 4 third-level page tables. For 600K, you need 2400 pages, or 38 third-level page tables and for 48K you need 192 pages or 3 third-level page tables. Assuming 2 bytes per entry, the space required is 1024 * 2 + 256 * 3 * 2 (3 second-level page tables) + 64 * (38+4+3)* 2 (38 third-level page tables for data segment, 4 for stack and 3 for code segment) = 9344 bytes.

Problem 2

A computer system has a 36-bit virtual address space with a page size of 8K, and 4 bytes per page table entry.

1. How many pages are in the virtual address space?
2. What is the maximum size of addressable physical memory in this system?
3. If the average process size is 8GB, would you use a one-level, two-level, or three-level page table? Why?
4. Compute the average size of a page table in question 3 above.

Solution:

1. A 36 bit address can address 2^36 bytes in a byte addressable machine. Since the size of a page 8K

bytes (2^13), the number of addressable pages is 2^36 / >2^13 = 2^23

2. With 4 byte entries in the page table we can reference 2^32 pages. Since each page is 2^13 B long, the maximum addressable physical memory size is 2^32 * 2^13 = 2^45 B (assuming no protection bits are used).

3. 8 GB = 2^33 B

   We need to analyze memory and time requirements of paging schemes in order to make a decision. Average process size is considered in the calculations below.

   1 Level Paging
   Since we have 2^23 pages in each virtual address space, and we use 4 bytes per page table entry, the size of the page table will be 2^23 * 2^2 = 2^25. This is 1/256 of the process' own memory space, so it is quite costly. (32 MB)

   2 Level Paging
   The address would be divided up as 12 | 11 | 13 since we want page table pages to fit into one page and we also want to divide the bits roughly equally.

   Since the process' size is 8GB = 2^33 B, I assume what this means is that the total size of all the distinct pages that the process accesses is 2^33 B. Hence, this process accesses 2^33 / 2^13 = 2^20 pages. The bottom level of the page table then holds 2^20 references. We know the size of each bottom level chunk of the page table is 2^11 entries. So we need 2^20 / 2^11 = 2^9 of those bottom level chunks.

   The total size of the page table is then:

//size of the outer page table		//total size of the inner pages		
1 * 2^12 * 4	+	2^9 * 2^11 * 4	=	2^20 * (2^-6 + 4) ~4MB

   3 Level Paging
   For 3 level paging we can divide up the address as follows:
   8 | 8 | 7 | 13

   Again using the same reasoning as above we need 2^20/2^7 = 2^13 level 3 page table chunks. Each level 2 page table chunk references 2^8 level 3 page table chunks. So we need 2^13/2^8 = 2^5 level-2 tables. And, of course, one level-1 table.

   The total size of the page table is then:

//size of the outer page table	//total size of the level 2 tables	//total size of innermost tables	
1 * 2^8 * 4	2^5 * 2^8 *4	2^13 * 2^7 * 4	~4MB

   As easily seen, 2-level and 3-level paging require much less space then level 1 paging scheme. And since our address space is not large enough, 3-level paging does not perform any better than 2 level paging. Due to the cost of memory accesses, choosing a 2 level paging scheme for this process is much more logical.

4. Calculations are done in answer no. 3.

Problem 3

In a 32-bit machine we subdivide the virtual address into 4 pieces as follows:

8-bit 4-bit 8-bit 12-bit

We use a 3-level page table, such that the first 8 bits are for the first level and so on. Physical addresses are 44 bits and there are 4 protection bits per page. Answer the following questions, showing all the steps you take to reach the answer. A simple number will not receive any credit.

1. What is the page size in such a system? Explain your answer (a number without justification will not get any credit).

2. How much memory is consumed by the page table and wasted by internal fragmentation for a process that has 64K of memory starting at address 0?

3. How much memory is consumed by the page table and wasted by internal fragmentation for a process that has a code segment of 48K starting at address 0x1000000, a data segment of 600K starting at address 0x80000000 and a stack segment of 64K starting at address 0xf0000000 and growing upward (towards higher addresses)?

Solution:

1. 4K. The last 12 bits of the virtual address are the offset in a page, varying from 0 to 4095. So the page size is 4096, that is, 4K.

2. 2912 or 4224 bytes for page tables, 0 bytes for internal fragmentation.
   Using the subdivision above, the 1st level page table contains 256 entries, each entry pointing to a 2nd level page table. A 2nd level page table contains 16 entries, each entry pointing to a 3rd page table. A 3rd page table contains 256 entries, each entry pointing to a page. The process's address space consists of 16 pages, thus we need 1 third-level page table. Therefore we need 1 entry in a 2nd level page table, and one entry in the first level page table. Therefore the size is: 256 entries for the first table, 16 entries for the 2nd level page table, and 1 3rd level page table containing 256 entries.

   Since physical addresses are 44 bits and page size is 4K, the page frame number occupies 32 bits. Taking the 4 protection bits into account, each entry of the level-3 page table takes (32+4) = 36 bits. Rounding up to make entries byte (word) aligned would make each entry consume 40 (64) bits or 5 (8) bytes. For a 256 entry table, we need 1280 (2048) bytes.

   The top-level page table should not assume that 2^{nd} level page tables are page-aligned. So, we store full physical addresses there. Fortunately, we do not need control bits. So, each entry is at least 44 bits (6 bytes for byte-aligned, 8 bytes for word-aligned). Each top-level page table is therefore 256*6 = 1536 bytes (256 * 8 = 2048 bytes).

   Trying to take advantage of the 256-entry alignment to reduce entry size is probably not worth the trouble. Doing so would be complex; you would need to write a new memory allocator that guarantees such alignment. Further, we cannot quite fit a table into a 1024-byte aligned region (44-10 = 34 bits per address, which would require more than 4 bytes per entry), and rounding the size up to the next power of 2 would not save us any size over just storing pointers and using the regular allocator.

   Similarly, each entry in the 2^{nd} level page table is a 44-bit physical pointer, 6 bytes (8 bytes) when aligned to byte (word) alignment. A 16 entry table is therefore 96 (128) bytes. So the space

required is 1536 (2048) bytes for the top-level page table + 96 (128) bytes for one second-level page table + 1280 (2048) bytes for one third-level page table = 2912 (4224) bytes. Since the process can fit exactly into 16 pages, there is no memory wasted by internal fragmentation.

3. 5664 or 8576 bytes for page tables, 0 bytes.
   First, the stack, data and code segments are at addresses that require having 3 page table entries active in the first level page table, so we have 3 second-level page tables. For 48K, you need 12 pages or 1 third-level page table; for 600K, you need 150 pages, or 1 third-level page table and for 64K you need 16 pages or 1 third-level page table.

   So the space required is 1536 (2048) bytes for the top level page table + 3 * 96 (3 * 128) bytes for 3 second-level page tables + 3 * 1280 (3 * 2048) for 3 third-level page table = 5664 (8576) bytes.

   As the code, data, stack segment of the process fits exactly into 12, 150, 16 pages respectively, there is no memory wasted by internal fragmentation.

Problem 4

In keping with the RISC processor design philosophy of moving hardware functionality to software, you see a proposal that processor designers remove the MMU (memory management unit) from the hardware. To replace the MMU, the proposal has compilers generate what is known as position independent code (PIC). PIC can be loaded and run at any adress without any relocation being performed. Assuming that PIC code runs just as fast as the non-PIC code, what would be the disadvantaqge of this scheme compared to the page MMU used on modern microprocessors?

Solution:
Need solution.

Problem 5

Describe the advantages of using a MMU that incorporates segmentation and paging over ones that are either pure paging or pure segmentation. Present your answer as separate lists of advantages over each of the pure schemes.

Solution:
Need solution.

Problem 6

Consider the following piece of code which multiplies two matrices:

```
int a[1024][1024], b[1024][1024], c[1024][1024];
multiply()
{
   unsigned i, j, k;
   for(i = 0; i < 1024; i++)
       for(j = 0; j < 1024; j++)
           for(k = 0; k < 1024; k++)
               c[i][j] += a[i,k] * b[k,j];
}
```

Assume that the binary for executing this function fits in one page, and the stack also fits in one page. Assume further that an integer requires 4 bytes for storage. Compute the number of TLB misses if the page size is 4096 and the TLB has 8 entries with a replacement policy consisting of LRU.

Solution:
1024*(2+1024*1024) = 1073743872
The binary and the stack each fit in one page, thus each takes one entry in the TLB. While the function is running, it is accessing the binary page and the stack page all the time. So the two TLB entries for these two pages would reside in the TLB all the time and the data can only take the remaining 6 TLB entries.

We assume the two entries are already in TLB when the function begins to run. Then we need only consider those data pages.

Since an integer requires 4 bytes for storage and the page size is 4096 bytes, each array requires 1024 pages. Suppose each row of an array is stored in one page. Then these pages can be represented as a[0..1023], b[0..1023], c[0..1023]: Page a[0] contains the elements a[0][0..1023], page a[1] contains the elements a[1][0..1023], etc.

For a fixed value of i, say 0, the function loops over j and k, we have the following reference string:

a[0], b[0], c[0], a[0], b[1], c[0], ¡ a[0], b[1023], c[0]
¡
a[0], b[0], c[0], a[0], b[1], c[0], ¡ a[0], b[1023], c[0]

For the reference string (1024 rows in total), a[0], c[0] will contribute two TLB misses. Since a[0] and b[0] each will be accessed every four memory references, the two pages will not be replaced by the LRU algorithm. For each page in b[0..1023], it will incur one TLB miss every time it is accessed. So the number of TLB misses for the second inner loop is
2+1024*1024 = 1048578
So the total number of TLB misses is 1024*1048578 = 1073743872

Problem 7

1. A computer system has a page size of 1,024 bytes and maintains the page table for each process in main memory. The overhead required for doing a lookup in the page table is 500 ns. To recude this overhead, the comnputer has a TLB that caches 32 virtual pages to physical frame mappings. A TLB lookup requires 100ns. What TLB hit-rate is required to ensure an average virtual address translation time of 200ns?

   Solution:
   Need solution.

2. Discuss the issues involved in picking a page size for a virtual memory system.
   a. Name one issue that argues for small page sizes? Name two that argue for large page sizes?
   b. How do the characteristics of disks influence the selection of a page size?

   Solution:

   a. Need solution
   b. Need solution

Problem 8

Consider a system with a virtual address size of 64MB (2^26), a physical memory of size 2GB (2^31), and a page size of 1K (2^10). Under the target workload, 32 processes (2^5) are running; half of the processes are smaller than 8K (2^13) and half use the full 64MB virtual address space. Each page has 4 control bits.

1. What is the size of a single top-level page table entry (and why)?
   Solution:

2. What is the size of a single bottom-level page table entry (and why)?
   Solution:

3. If you had to choose between two arrangements of page table: 2-level and 3-level, which would you choose and why? Compute the expected space overhead for each variation: State the space overhead for each small process and each large process. Then compute the total space overhead for the entire system.
   Solution: