**SOLUTIONS TO CHAPTER 3 PROBLEMS**

1. First, special hardware is needed to do the comparisons, and it must be fast, since it is used on every memory reference. Second, with 4-bit keys, only 16 programs can be in memory at once (one of which is the operating system).
2. It is an accident. The base register is 16,384 because the program happened to be loaded at address 16,384. It could have been loaded anywhere. The limit register is 16,384 because the program contains 16,384 bytes. It could have been any length. That the load address happens to exactly match the program length is pure coincidence.
3. Almost the entire memory has to be copied, which requires each word to be read and then rewritten at a different location. Reading 4 bytes takes 4 nsec, so reading 1 byte takes 1 nsec and writing it takes another 2 nsec, for a total of 2 nsec per byte compacted. This is a rate of 500,000,000 bytes/sec. To copy 4 GB (2232 bytes, which is about 4. 295 × 109 bytes), the computer needs 232/500, 000, 000 sec, which is about 859 msec. This number is slightly pessimistic because if the initial hole at the bottom of memory is *k* bytes, those *k* bytes do not need to be copied. However, if there are many holes and many data segments, the holes will be small, so *k* will be small and the error in the calculation will also be small.
4. First fit takes 20 MB, 10 MB, 18 MB. Best fit takes 12 MB, 10 MB, and 9 MB. Worst fit takes 20 MB, 18 MB, and 15 MB. Next fit takes 20 MB, 18 MB, and 9 MB.
5. Real memory uses physical addresses. These are the numbers that the memory chips react to on the bus. Virtual addresses are the logical addresses that refer to a process’ address space. Thus a machine with a 32-bit word can generate virtual addresses up to 4 GB regardless of whether the machine has more or less memory than 4 GB.
6. For a 4-KB page size the (page, offset) pairs are (4, 3616), (8, 0), and (14, 2656). For an 8-KB page size they are (2, 3616), (4, 0), and (7, 2656).
7. (a) 8212. (b) 4100. (c) 24684.
8. They built an MMU and inserted it between the 8086 and the bus. Thus all 8086 physical addresses went into the MMU as virtual addresses. The MMU then mapped them onto physical addresses, which went to the bus.
9. There needs to be an MMU that can remap virtual pages to physical pages. Also, when a page not currently mapped is referenced, there needs to be a trap to the operating system so it can fetch the page.
10. If the smartphone supports multiprogramming, which the iPhone, Android, and Windows phones all do, then multiple processes are supported. If a process forks and pages are shared between parent and child, copy on write definitely makes sense. A smartphone is smaller than a server, but logically it is not so different.
11. For these sizes
    1. *M* has to be at least 4096 to ensure a TLB miss for every access to an element of *X*. Since *N* affects only how many times *X* is accessed, any value of *N* will do.
    2. *M* should still be at least 4,096 to ensure a TLB miss for every access to an element of *X*. But now *N* should be greater than 64K to thrash the TLB, that is, *X* should exceed 256 KB.
12. The total virtual address space for all the processes combined is *nv*, so this much storage is needed for pages. However, an amount *r* can be in RAM, so the amount of disk storage required is only *nv* −*r*. This amount is far more than is ever needed in practice because rarely will there be *n* processes actually running and even more rarely will all of them need the maximum allowed virtual memory.
13. A page fault every *k* instructions adds an extra overhead of *n*/*k μ* sec to the average, so the average instruction takes 1 +*n*/*k* nsec.
14. The page table contains 232/213 entries, which is 524,288. Loading the page table takes 52 msec. If a process gets 100 msec, this consists of 52 msec for loading the page table and 48 msec for running. Thus 52% of the time is spent loading page tables.
15. Under these circumstances:
    1. We need one entry for each page, or 224 = 16 × 1024 × 1024 entries, since there are 36 = 48 − 12 bits in the page number field.
    2. Instruction addresses will hit 100% in the TLB. The data pages will have a 100 hit rate until the program has moved onto the next data page. Since a 4-KB page contains 1,024 long integers, there will be one TLB miss and one extra memory access for every 1,024 data references.
16. The chance of a hit is 0.99 for the TLB, 0.0099 for the page table, and 0.0001 for a page fault (i.e., only 1 in 10,000 references will cause a page fault). The effective address translation time in nsec is then:

0. 99 × 1 + 0. 0099 × 100 + 0. 0001 × 6 × 106 ≈ 602 clock cycles.

Note that the effective address translation time is quite high because it is dominated by the page replacement time even when page faults only occur once in 10,000 references.

1. Consider,
   1. A multilevel page table reduces the number of actual pages of the page table that need to be in memory because of its hierarchic structure. In fact, in a program with lots of instruction and data locality, we only need the top-level page table (one page), one instruction page, and one data page.
   2. Allocate 12 bits for each of the three page fields. The offset field requires 14 bits to address 16 KB. That leaves 24 bits for the page fields. Since each entry is 4 bytes, one page can hold 212 page table entries and therefore requires 12 bits to index one page. So allocating 12 bits for each of the page fields will address all 238 bytes.
2. The virtual address was changed from (PT1, PT2, Offset) to (PT1, PT2, PT3, Offset). But the virtual address still used only 32 bits. The bit configuration of a virtual address changed from (10, 10, 12) to (2, 9, 9, 12)
3. Twenty bits are used for the virtual page numbers, leaving 12 over for the offset. This yields a 4-KB page. Twenty bits for the virtual page implies 220 pages.
4. For a one-level page table, there are 232/212 or 1M pages needed. Thus the page table must have 1M entries. For two-level paging, the main page table has 1K entries, each of which points to a second page table. Only two of these are used. Thus, in total only three page table entries are needed, one in the top-level table and one in each of the lower-level tables.
5. The code and reference string are as follows

|  |  |
| --- | --- |
| LOAD 6144,R0 | 1(I), 12(D) |
| PUSH R0 | 2(I), 15(D) |
| CALL 5120 | 2(I), 15(D) |
| JEQ 5152 | 10(I) |

The code (I) indicates an instruction reference, whereas (D) indicates a data reference.

1. The effective instruction time is 1*h* + 5(1 −*h*), where *h* is the hit rate. If we equate this formula with 2 and solve for *h*, we find that *h* must be at least 0.75.
2. An associative memory essentially compares a key to the contents of multiple registers simultaneously. For each register there must be a set of comparators that compare each bit in the register contents to the key being searched for. The number of gates (or transistors) needed to implement such a device is a linear function of the number of registers, so expanding the design gets expensive linearly.
3. With 8-KB pages and a 48-bit virtual address space, the number of virtual pages is 248/213, which is 235 (about 34 billion).
4. The main memory has 228/213 = 32,768 pages. A 32K hash table will have a mean chain length of 1. To get under 1, we have to go to the next size, 65,536 entries. Spreading 32,768 entries over 65,536 table slots will give a mean chain length of 0.5, which ensures fast lookup.
5. This is probably not possible except for the unusual and not very useful case of a program whose course of execution is completely predictable at compilation time. If a compiler collects information about the locations in the code of calls to procedures, this information might be used at link time to rearrange the object code so that procedures were located close to the code that calls them. This would make it more likely that a procedure would be on the same page as the calling code. Of course this would not help much for procedures called from many places in the program.
6. Under these circumstances
   1. Every reference will page fault unless the number of page frames is 512, the length of the entire sequence.
   2. If there are 500 frames, map pages 0–498 to fixed frames and vary only one frame.
7. The page frames for FIFO are as follows:

x0172333300

xx017222233

xxx01777722

xxxx0111177

The page frames for LRU are as follows:

x0172327103

xx017232710

xxx01773271

xxxx0111327

FIFO yields six page faults; LRU yields seven.

1. The first page with a 0 bit will be chosen, in this case *D*.
2. The counters are Page 0: 0110110 Page 1: 01001001 Page 2: 00110111 Page 3: 10001011

**31.** The sequence: 0, 1, 2, 1, 2, 0, 3. In LRU, page 1 will be replaced by page 3. In clock, page 1 will be replaced, since all pages will be marked and the cursor is

|  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- |
|  | at page 0. | |  |  |  |  |  |  |
| **32.** | The age of the page is 2204 − 1213 = 991. | | | | | | If*τ* = 400, it is definitely out of the | |
|  | working set and was not recently referenced so it will be evicted. The*τ* = 1000 | | | | | | | |
|  | situation is different. Now the page falls within the working set (barely), so it | | | | | | | |
|  | is not removed. | | |  |  |  |  |  |
| **33.** | Consider, | |  |  |  |  |  |  |
|  | (a) For every *R* bit that is set, set the time-stamp value to 10 and clear all *R* | | | | | | | |
|  |  | bits. You could also change the (0,1) *R-M* entries to (0,0\*). So the entries | | | | | | |
|  |  | for pages 1 and 2 will change to: | | | |  |  |  |
|  |  |  |  |  |  |  |  |  |
|  |  | **Page** | **Time stamp** | **V** | **R** | **M** |  |  |
|  |  |  |  |  |  |  |  |  |
|  |  | 0 | 6 | 1 | 0 | 0\* |  |  |
|  |  |  |  |  |  |  |  |  |
|  |  | 1 | 10 | 1 | 0 | 0 |  |  |
|  |  |  |  |  |  |  |  |  |
|  |  | 2 | 10 | 1 | 0 | 1 |  |  |
|  |  |  |  |  |  |  |  |  |
|  | (b) Evict page 3 (*R* = 0 and *M* = 0) and load page 4: | | | | | | | |
|  |  |  |  |  |  |  |  |  |
|  |  | **Page** | **Time stamp** | **V** | **R** | **M** | **Notes** |  |
|  |  |  |  |  |  |  |  |  |
|  |  | 0 | 6 | 1 | 0 | 1 |  |  |
|  |  |  |  |  |  |  |  |  |
|  |  | 1 | 9 | 1 | 1 | 0 |  |  |
|  |  |  |  |  |  |  |  |  |
|  |  | 2 | 9 | 1 | 1 | 1 |  |  |
|  |  |  |  |  |  |  |  |  |
|  |  | 3 | 7 | 0 | 0 | 0 | Changed from 7 (1,0,0) |  |
|  |  |  |  |  |  |  |  |  |
|  |  | 4 | 10 | 1 | 1 | 0 | Changed from 4 (0,0,0) |  |
|  |  |  |  |  |  |  |  |  |

1. Consider,
   1. The attributes are: (FIFO) load time; (LRU) latest reference time; and (Optimal) nearest reference time in the future.
   2. There is the labeling algorithm and the replacement algorithm. The labeling algorithm labels each page with the attribute given in part a. The replacement algorithm evicts the page with the smallest label.
2. The seek plus rotational latency is 10 msec. For 2-KB pages, the transfer time is about 0.009766 msec, for a total of about 10.009766 msec. Loading 32 of these pages will take about 320.21 msec. For 4-KB pages, the transfer time is doubled to about 0.01953 msec, so the total time per page is 10.01953 msec. Loading 16 of these pages takes about 160.3125 msec. With such fast disks, all that matters is reducing the number of transfers (or putting the pages consecutively on the disk).
3. NRU removes page 2. FIFO removes page 3. LRU removes page 1. Second chance removes page 2.
4. Sharing pages brings up all kinds of complications and options:
   * 1. The page table update should be delayed for process *B* if it will never access the shared page or if it accesses it when the page has been swapped out again. Unfortunately, in the general case, we do not know what process *B* will do in the future.
     2. The cost is that this lazy page fault handling can incur more page faults. The overhead of each page fault plays an important role in determining if this strategy is more efficient. (*Aside*: This cost is similar to that faced by the copy-on-write strategy for supporting some UNIX fork system call implementations.)
5. Fragment *B* since the code has more spatial locality than Fragment *A*. The inner loop causes only one page fault for every other iteration of the outer loop. (There will be only 32 page faults.) [*Aside* (Fragment *A*): Since a frame is 128 words, one row of the *X* array occupies half of a page (i.e., 64 words). The entire array fits into 64 × 32/128 = 16 frames. The inner loop of the code steps through consecutive rows of *X* for a given column. Thus, every other reference to *X*[*i*][ *j*] will cause a page fault. The total number of page faults will be 64 × 64/2 = 2, 048].
6. It can certainly be done.
   1. The approach has similarities to using flash memory as a paging device in smartphones except now the virtual swap area is a RAM located on a remote server. All of the software infrastructure for the virtual swap area would have to be developed.
   2. The approach might be worthwhile by noting that the access time of disk drives is in the millisecond range while the access time of RAM via a network connection is in the microsecond range if the software overhead is not too high. But the approach might make sense only if there is lots of idle RAM in the server farm. And then, there is also the issue of reliability. Since RAM is volatile, the virtual swap area would be lost if the remote server went down.
7. The PDP-1 paging drum had the advantage of no rotational latency. This saved half a rotation each time memory was written to the drum.
8. The text is eight pages, the data are five pages, and the stack is four pages. The program does not fit because it needs 17 4096-byte pages. With a 512-byte page, the situation is different. Here the text is 64 pages, the data are 33 pages, and the stack is 31 pages, for a total of 128 512-byte pages, which fits. With the small page size it is OK, but not with the large one.
9. The program is getting 15,000 page faults, each of which uses 2 msec of extra processing time. Together, the page fault overhead is 30 sec. This means that of the 60 sec used, half was spent on page fault overhead, and half on running the program. If we run the program with twice as much memory, we get half as many memory page faults, and only 15 sec of page fault overhead, so the total run time will be 45 sec.
10. It works for the program if the program cannot be modified. It works for the data if the data cannot be modified. However, it is common that the program cannot be modified and extremely rare that the data cannot be modified. If the data area on the binary file were overwritten with updated pages, the next time the program was started, it would not have the original data.
11. The instruction could lie astride a page boundary, causing two page faults just to fetch the instruction. The word fetched could also span a page boundary, generating two more faults, for a total of four. If words must be aligned in memory, the data word can cause only one fault, but an instruction to load a 32-bit word at address 4094 on a machine with a 4-KB page is legal on some machines (including the x86).
12. Internal fragmentation occurs when the last allocation unit is not full. External fragmentation occurs when space is wasted between two allocation units. In a paging system, the wasted space in the last page is lost to internal fragmentation. In a pure segmentation system, some space is invariably lost between the segments. This is due to external fragmentation.
13. No. The search key uses both the segment number and the virtual page number, so the exact page can be found in a single match.
14. Here are the results:

|  |  |  |
| --- | --- | --- |
|  | **Address** | **Fault?** |
|  |  |  |
| (a) | (14, 3) | No (or 0xD3 or 1110 0011) |
|  |  |  |
| (b) | NA | Protection fault: Write to read/execute segment |
|  |  |  |
| (c) | NA | Page fault |
|  |  |  |
| (d) | NA | Protection fault: Jump to read/write segment |
|  |  |  |

1. General virtual memory support is not needed when the memory requirements of all applications are well known and controlled. Some examples are smart cards, special-purpose processors (e.g., network processors), and embedded processors. In these situations, we should always consider the possibility of using more real memory. If the operating system did not have to support virtual memory, the code would be much simpler and smaller. On the other hand, some ideas from virtual memory may still be profitably exploited, although with different design requirements. For example, program/thread isolation might be paging to flash memory.
2. This question addresses one aspect of virtual machine support. Recent attempts include Denali, Xen, and VMware. The fundamental hurdle is how to achieve near-native performance, that is, as if the executing operating system had memory to itself. The problem is how to quickly switch to another operating system and therefore how to deal with the TLB. Typically, you want to give some number of TLB entries to each kernel and ensure that each kernel operates within its proper virtual memory context. But sometimes the hardware (e.g., some Intel architectures) wants to handle TLB misses without knowledge of what you are trying to do. So, you need to either handle the TLB miss in software or provide hardware support for tagging TLB entries with a context ID.