Sheet8
VIRTUAL MEMORY

1) What is the difference between simple paging and virtual memory paging?

   Simple paging: all the pages of a process must be in main memory for process to run, unless overlays are used. **Virtual memory paging:** not all pages of a process need be in main memory frames for the process to run.; pages may be read in as needed

2) Explain thrashing.

   Thrashing is a phenomenon in virtual memory schemes, in which the processor spends most of its time swapping pieces rather than executing instructions.

3) Why is the principle of locality crucial to the use of virtual memory?

   Algorithms can be designed to exploit the principle of locality to avoid thrashing. In general, the principle of locality allows the algorithm to predict which resident pages are least likely to be referenced in the near future and are therefore good candidates for being swapped out.

4) What elements are typically found in a page table entry? Briefly define each element.

   Frame number: the sequential number that identifies a page in main memory; **present bit:** indicates whether this page is currently in main memory; **modify bit:** indicates whether this page has been modified since being brought into main memory.

5) What is the purpose of a translation lookaside buffer?

   The TLB is a cache that contains those page table entries that have been most recently used. Its purpose is to avoid, most of the time, having to go to disk to retrieve a page table entry.

6) Suppose the page table for the process currently executing on the processor looks like the following. All numbers are decimal, everything is numbered starting from zero, and all addresses are memory byte addresses. The page size is 1024 bytes.

Virtual page number	Valid bit	Reference bit	Modify bit	Page frame number
0	1	1	0	4
1	1	1	1	7
2	0	0	0	—
3	1	0	0	2
4	0	0	0	—
5	1	0	1	0

   a) Describe exactly how, in general, a virtual address generated by the CPU is translated into a physical main memory address.

      Split binary address into virtual page number and offset; use VPN as index into page table; extract page frame number; concatenate offset to get physical memory address

b) What physical address, if any, would each of the following virtual addresses correspond to? (Do not try to handle any page faults, if any.)
   i. 1052
   ii. 2221
   iii. 5499

   i. 1052 = 1024 + 28 maps to VPN 1 in PFN 7, (7 x 1024+28 = 7196)
   ii. 2221 = 2 * 1024 + 173 maps to VPN 2, page fault
   iii. 5499 = 5 * 1024 + 379 maps to VPN 5 in PFN 0, (0 x 1024+379 = 379)

7) Consider the following program.

```
#define Size 64
int A[Size; Size], B[Size; Size], C[Size; Size];
int register i, j;
for (j = 0; j< Size; j ++)
for (i = 0; i< Size; i++)
C[i; j] = A[i; j] + B[i; j];
```

Assume that the program is running on a system using demand paging and the page size is 1 Kilobyte. Each integer is 4 bytes long. It is clear that each array requires a 16-page space. As an example, A[0, 0]-A[0, 63], A[1, 0]-A[1, 63], A[2, 0]-A[2, 63], and A[3, 0]-A[3, 63] will be stored in the first data page. A similar storage pattern can be derived for the rest of array A and for arrays B and C. Assume that the system allocates a 4-page working set for this process. One of the pages will be used by the program and three pages can be used for the data. Also, two index registers are assigned for i and j (so, no memory accesses are needed for references to these two variables).

a) Discuss how frequently the page fault would occur (in terms of number of times C[i, j] = A[i, j] + B[i, j] are executed).

   3 page faults for every 4 executions of C[i, j] = A[i, j] +B[i, j].

b) Can you modify the program to minimize the page fault frequency?

   Yes. The page fault frequency can be minimized by switching the inner and outer loops.

c) What will be the frequency of page faults after your modification?

   After modification, there are 3 page faults for every 256 executions.

8) Two questions:

a) How much memory space is needed for the user page table of Figure 8.3?

   4 MByte

b) Assume you want to implement a hashed inverted page table for the same addressing scheme as depicted in Figure 8.3, using a hash function that maps the 20-bit page number into a 6-bit hash value. The table entry contains the page number, the frame number, and a chain pointer. If the page table allocates space for up to 3 overflow entries per hashed entry, how much memory space does the hashed inverted page table take?

   the hash value is 6 bits so there can be 2^6 chains each of which consist of an entry plus 3 overflow entries.

   Number of rows: $2^6 \times (1 + 3) = 2^8 = 256$ entries.

   Each entry consists of: 20 (page number) + 20 (frame number) + 8 bits (chain index) = 48 bits = 6 bytes.

   Total: 256 x 6= 1536 bytes

9) Assuming a page size of 4 Kbytes and that a page table entry takes 4 bytes, how many levels of page tables would be required to map a 64-bit address space, if the top level page table fits into a single page?

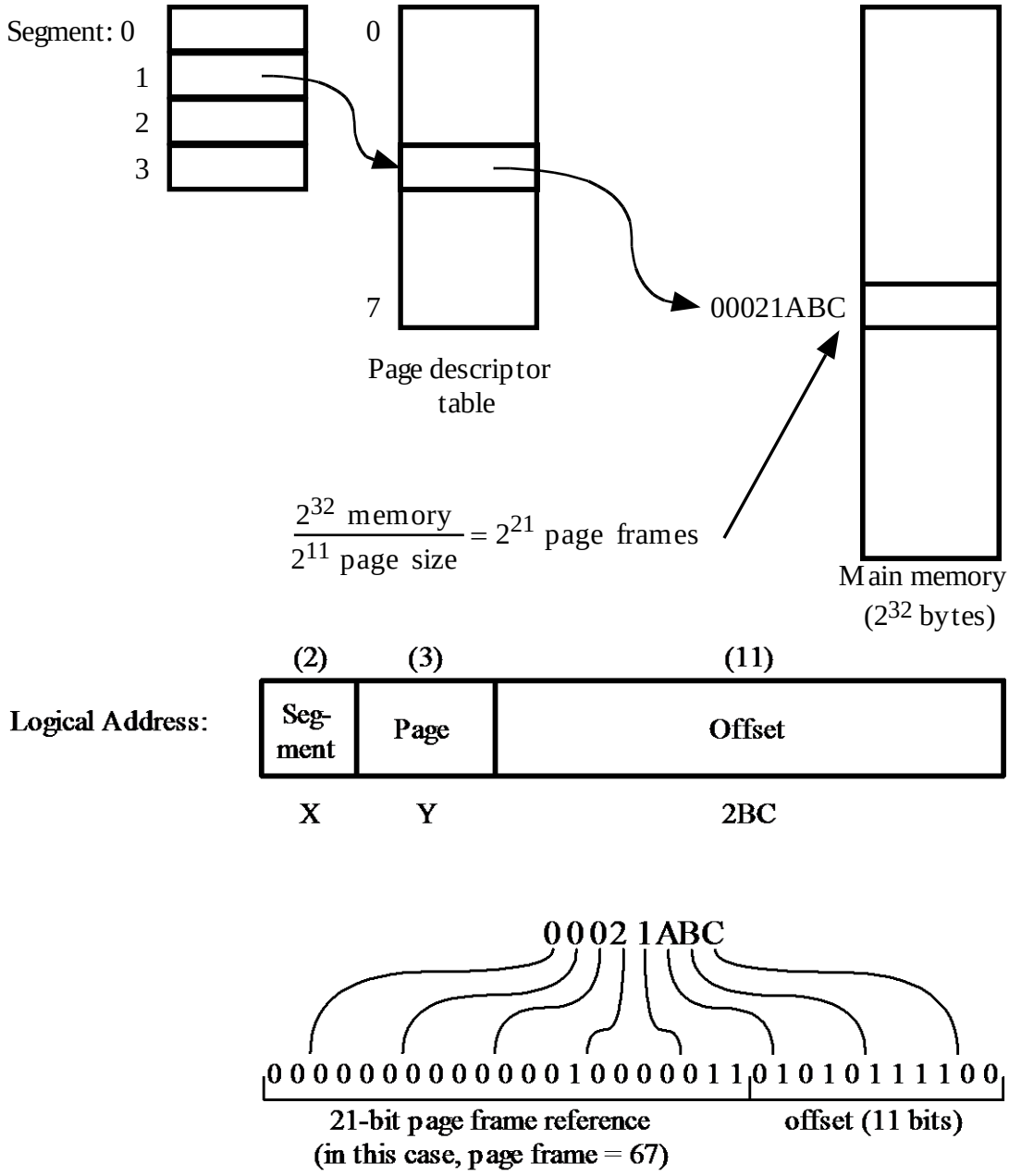
Since each page table entry is 4 bytes and each page contains 4 Kbytes, then a one-page page table would point to 1024 = 2^{10} pages, addressing a total of 2^{10} x 2^{12} = 2^{22} bytes. The address space however is 2^{64} bytes. Adding a second layer of page tables, the top page table would point to 2^{10} page tables, addressing a total of 2^{32} bytes. Continuing this process,

Depth	Address Space
1	2^{22} bytes
2	2^{32} bytes
3	2^{42} bytes
4	2^{52} bytes
5	2^{62} bytes
6	2^{72} bytes (> 2^{64} bytes)

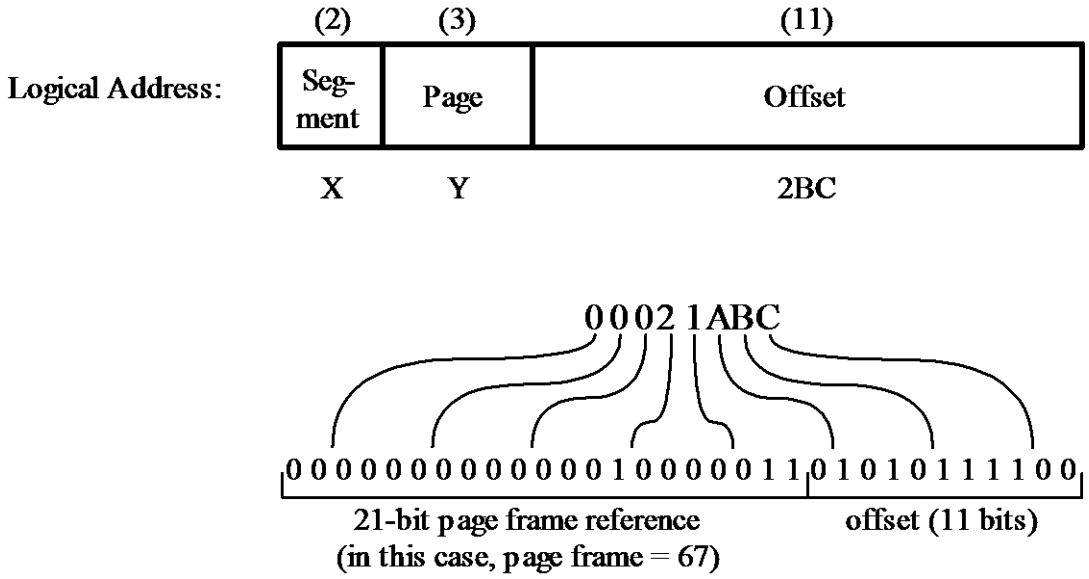
we can see that 5 levels do not address the full 64-bit address space, so a 6th level is required. But only 2 bits of the 6th level are required, not the entire 10 bits. So instead of requiring your virtual addresses be 72 bits long, you could mask out and ignore all but the 2 lowest order bits of the 6th level. This would give you a 64-bit address. Your top-level page table then would have only 4 entries. Yet another option is to revise the criteria that the top-level page table fit into a single physical page and instead make it fit into 4 pages. This would save a physical page, which is not much.

10) Assume that a task is divided into four equal-sized segments and that the system builds an eight-entry page descriptor table for each segment. Thus, the system has a combination of segmentation and paging. Assume also that the page size is 2 Kbytes.
   a) What is the maximum size of each segment?
   b) What is the maximum logical address space for the task?
   c) Assume that an element in physical location 00021ABC is accessed by this task. What is the format of the logical address that the task generates for it? What is the maximum physical address space for the system?

$$\frac{2^{32}\ \text{memory}}{2^{11}\ \text{page size}} = 2^{21}\ \text{page frames}$$



(a) $8 \times 2K = 16K$
(b) $16K \times 4 = 64K$
(c) $2^{32} = 4$ GBytes



11) Consider a paged logical address space (composed of 32 pages of 2 Kbytes each) mapped into a 1-Mbyte physical memory space.

a) What is the format of the processor's logical address?

page number (5)	offset (11)

b) What is the length and width of the page table (disregarding the "access rights" bits)?

32 entries, each entry is 9 bits wide.

c) What is the effect on the page table if the physical memory space is reduced by half?

If total number of entries stays at 32 and the page size does not change, then each entry becomes 8 bits wide.
