The x537

CS 537 Fall '22 Midterm

The **x537** is a new processor, introduced just for the purpose of examining students on the inner workings of operating systems. How cruel, to make a new processor just for an exam!

The hardware can be configured to run in a number of different ways. For example, in base-and-bounds mode, x537 uses base-and-bounds relocation. However, it can also be configured to use paging, segmentation, and other virtual-memory mechanisms. It can even use multi-level paging, which for some reason, the designers call "mode 3000". Why do they call it that? It is a mystery, one which may never be solved.

The operating system that runs on x537 is called **xOS**. xOS has a lot of features too, which you will learn about during this exam. How unusual, to make a new operating system, just for an exam!

Cheat sheet freebies:

- 1 KB is 1024 bytes; you need 10 bits to refer to 1 KB of address space
- 1 MB is 1024 * 1024 bytes; you need 20 bits to refer to 1 MB of address space

The exam has **16 total pages**, and **48 total questions (each worth the same amount)**. Please **fill out a single answer for each** one!

On the answer sheet, please fill out your **name** and **student ID** number.

Remember, use an **Easiest Problem First** scheduler (EPF) to maximize your time and score.

And, from the x537/xOS design team, a hearty **good luck!**

Finally, please hand back the questions when you are finished, as some people are taking the exam at a later date. You will receive an electronic version later.

Answer Key!

Question 1: Base and Bounds

Assume for this question that the x537 in question has **a single CPU**, and uses **base and bounds** (also called **dynamic relocation**) to implement virtual memory. Assume a (small) **virtual address space** of size **1 KB** (1024 bytes).

1. With one CPU, how many **base registers** are in the system (total)?
   a. 0
   [✓ b. 1]
   c. 4
   d. 8
   e. None of the above

there is one base register per CPU

2. Assume the base register is set to 1000, and the bounds to 100. How many accesses to memory will the following instruction sequence generate, when fetched and then executed upon that CPU?

```
load 10, r1
```

   a. 0
   b. 1
   [✓ c. 2]
   d. 3
   e. None of the above

fetch instruction: one access

load data into r_1: one access

thus, two total

3. With base=1000, and bounds=100, what **physical address** will virtual address 10 be translated to when it is accessed?
   a. 100
   b. 1000
   [✓ c. 1010]
   d. 110
   e. None of the above

$PA = VA + base$

$= 10 + 1000 = 1010$

(within bounds)

4. With the bounds register set to 100, how many **different legal addresses** can a process access?
   a. 0
   b. 1
   [✓ c. 100]
   d. 1000
   e. None of the above

bounds set to 100 means 0 ... 99 are legal

=> 100 legal addresses

5. Assume you have the following process table (list of active processes):

```
[ Process A  base:100  bounds:10 ]
[ Process B  base:1000 bounds:20 ]
[ Process C  base:500  bounds:50 ]
```

Assume process A is running on a CPU. After the switch to process B, which of the following is a physical address that process B might legally refer to?

a. 0
b. 20
c. 500
[✓ d. 1015]
e. None of the above

B's base is 1000, bounds 20
⇒ 1000 ... 1019 are legal
⇒ 1015

6. Given the process table above (with processes A, B, C), we observe the following **physical** memory accesses over time:

```
        A                         B                          C
100, 101, 109, 103, 1010, 1011, 1012, 1001, 540, 539, 538, 537,
102, 103, 104, 105, 1000, 1002, 1004, 1008, 500, 501, 502, 503
        A                         B                          C
```

What could you conclude about the scheduling policy of the system?

a. It is possibly Shortest Job First (SJF)
b. It is possibly Shortest Time to Completion First (STCF)
[✓ c. It is possibly Round Robin (RR)]
d. It is possibly Shortest Job Last (SJL)
e. It is possibly Least Recently Used (LRU)

Looks like it's rotating between A, B, C
⇒ maybe RR

7. When switching between process A and process B, what would happen if xOS updated the base register correctly (changing it from 100 to 1000) but **forgot** to update the bounds register (thus leaving it at 10).

a. Everything would work as expected
[✓ b. Process B might fault unexpectedly]
c. Process B might be able to access memory it shouldn't be able to access
d. Process A might be mad
e. None of the above

Bounds should be 20, but is 10

B might generate legal address (like 15), but fault!

8. Finally, assume you configure a system to have **four CPUs** (not just one). How many **total base registers** are in the system now?

a. 1
b. 2
[✓ c. 4]
d. 8
e. None of the above

one per CPU,
as each is running
a different process
and each is virtualized

Question 2: xOS Scheduling

xOS uses a multi-level feedback queue (**MLFQ**) by default to schedule processes. The queue has **four levels** and uses a quantum (time slice) length of **10ms**. Processes **start at high priority** (4) and then change priority as MLFQ dictates.

Unless stated otherwise, you can assume there is only a single CPU, which means that only one process can be scheduled at a time. You can also assume that context switches do not take any time or resources.

You also do not have to take more advanced MLFQ modifications into account, such as priority boosting or variable time-slice length, unless otherwise mentioned.

9. The xOS developers first considered using a **Shortest-Job-First** scheduler. Which of these problems does a SJF scheduler **not** have?
   a. Long jobs may never run
   b. A job's runtime is difficult to predict
   [✓ c. Average turnaround time may be greater than with other policies even if all jobs arrive at the same time] (SJF: really good at turnaround!)
   d. Does not support preemptive scheduling
   e. It does have all of these problems

10. The developers then also considered using a **Shortest Time-to-Completion First** scheduler. From class you might remember that it is similar to SJF, but supports preempting processes.

    Consider a job A arrives at time T=0 and job length of 500ms. Job B arrives at time T=100ms with a length of 100ms. What does the schedule for our CPU look like?
    a. A runs for 500ms, then B runs for 100ms
    b. B runs for 100ms, then A runs for 500ms
    [✓ c. A runs for 100ms, then B runs for 100ms, and then A runs again for 400ms]
    d. A runs for 100ms, then A and B run in round-robin for 200ms, and then A runs for another 300ms
    e. A runs for 100ms, then B runs for 100ms, then A runs for 100ms, then B runs for 100ms, then A runs for 300ms

    B preempts because shorter than rest of A

11. In what regard is MLFQ better compared to a scheduler like SJF or STCF?
    [✓ a. Average Response Time]
    b. Average Turnaround Time
    c. Fewer I/Os issued by processes
    d. CPU Utilization
    e. None of the above

    SJF, STCF both are bad at response

12. Let's focus on MLFQ now. Consider we run a process **P1** that uses the CPU for 15ms, then waits for I/O for 5ms, and then keeps using the CPU for another 100ms before it finishes. It starts at high priority (4). What priority level does P1 have after the first three time slices (30ms total)? Assume there are no other processes in the system.
    - a. 1
    - [✓ b. 2]
    - c. 3
    - d. 4
    - e. None of the above

start at 4,
move to 3, then 2

13. Now consider that 5ms after P1 was started, we start another process **P2.** P2 alternates between first using the CPU for 5ms and then waiting on I/O for another 5ms. P2 runs for 100ms total. What does the CPU schedule look like for the **first five time slices**?
    - a. P1, P2, P2, P2, P2
    - b. P1, P2, P2, P1, P2
    - c. P1, P1, P1, P2, P2
    - d. P2, P2, P2, P2, P2
    - [✓ e. None of the above]

P_1 starts, demotes
P_2 starts, runs 5ms, sleeps
P_1 runs again
thus P_1, P_2, P_1, thus (e)

14. The xOS developers decided **not to add boosting** to their MLFQ scheduler to reduce code complexity. What problems could this cause?
    - a. Interactive jobs might not be scheduled at all
    - [✓ b. CPU-heavy (batch) jobs might get starved]
    - c. Batch jobs will stay at the highest priority level
    - d. There can only be one job at the highest priority level
    - e. None of the above

the boost ensures
long-running jobs
will run again

15. Now let us consider what happens if there are **four CPUs** available to the scheduler. What is the maximum number of processes that can be **READY** at the same time?
    - a. 2
    - b. 4
    - c. 8
    - d. 16
    - [✓ e. None of the above]

CPUs don't limit the
number of processes,
which can be very high
on typical systems

16. Reconsider the scenario from above, with just a single process **P1**. **P1** uses the CPU for 15ms, then waits for I/O for 5ms, and then keeps using the CPU for another 100ms before it finishes. With **four CPUs**, what priority level does P1 have after the first three time slices (30ms)? Again, assume there are no other processes in the system.
    - a. 1
    - b. 2
    - c. 3
    - d. 4
    - e. None of the above

should be same as #12,

just accepted
all.

Question 3: Advanced Paging

xOS/x537 has some advanced paging features, which we will now explore.

One of the features is the support for **large pages**. Instead of just a standard 4KB page, the system also allows the use of **4MB pages**.

17. First, let's do some basic questions. Assuming a **32-bit virtual address space**, and **4KB pages**, how many page table entries (PTEs) would a **linear page table** have?
    a. Roughly 1,000
    b. Roughly 10,000
    c. Roughly 100,000
    [✓ d. Roughly 1,000,000]
    e. None of the above

VA	VPN	offset
	20 bits	12 bits
	$2^{20} \cong$ 1 million	$2^{12} = 4K$

18. Now imagine we have a **4MB page size**. Assuming the same **32-bit address space**, how many page table entries would a **linear page table** have?
    [✓ a. Roughly 1,000]
    b. Roughly 10,000
    c. Roughly 100,000
    d. Roughly 1,000,000
    e. None of the above

VPN	offset
10 bits	22 bits

$2^{10} \cong 1000$

19. Because large pages make page tables smaller, they are attractive to use. However, they introduce new problems as well. What is the main problem with using large pages?
    a. Large pages are hard to describe to people
    [✓ b. Large pages can lead to wasted space] →
    c. Large pages make page table access slower
    d. Large pages use more physical memory
    e. None of the above

larger pages means some of the pages may go unused ⇒ waste (internal fragmentation)

20. Large pages also can affect TLB usage. Which one of the following is true?
    a. Large pages can increase the size of the TLB
    [✓ b. Large pages can reduce TLB miss rates]
    c. Large pages affect whether processes can share the TLB
    d. Large pages change the color of the TLB
    e. None of the above

Because you can map more of your address space in the (fixed sized) TLB

21. We now investigate a hybrid system, in which xOS/x537 uses both 4KB and 4MB pages at the same time. In this version of the system, we use a **multi-level page table.** We again assume a 32-bit virtual address space.

The system uses a **page directory** at the top of the tree. However, it is a little different than what you have already learned about. The contents of each **page directory entry (PDE)** are as follows:

```
[ valid bit | large-page bit | physical frame number ]
```

In all cases, **page directory entries** are **4 bytes** in size, as are **page table entries**.

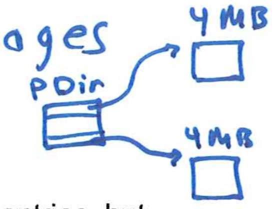
The top-level page directory has 1024 entries (hence it is 4KB in size). If an entry is valid, and the **large-page bit** is set to **zero**, then the physical frame number (PFN) refers to a 4KB page of the page table for the relevant portion of the address space (this is what you learned about already). However, if an entry is valid, and the **large-page bit** is set to **one**, then the PFN refers directly to a large page (size 4MB). Make sense?

And now, finally, some questions. First: if the first and last entries in the page directory are set to one (valid), and no other entries are set to one (the rest are not valid), and the first and last entry large-page bit is set to one for each of the valid entries, how much memory can the process access (total) legally?

a. 0 MB
b. 1 MB
c. 2 MB
d. 4 MB
[✓ e. None of the above]

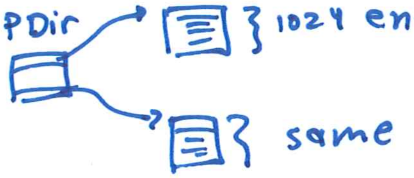
This page table points to two valid large pages

2 × 4 MB = 8 MB



22. Now assume the first and last page-directory entries are again the only valid entries, but both large-page bits are set to zero. Now, how much memory can the process access legally, assuming that any valid page of the page table consists of all valid entries?

a. 0 KB
b. 8 KB
c. 4 MB
[✓ d. 8 MB]
e. None of the above

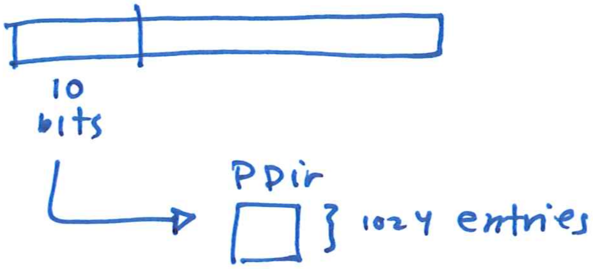


=> 2 × 1024 × 4KB

= 8 MB

23. Given a 32-bit virtual address, which bits are used to index into the page directory?

a. The first 8 bits (i.e., the 8 high-order bits)
[✓ b. The first 10 bits (i.e., the 10 high-order bits)]
c. The first 12 bits (i.e., the 12 high-order bits)
d. The last 10 bits (i.e., the 10 low-order bits)
e. None of the above



24. If we added one more bit to the virtual address (making it a 33-bit virtual address space), which of the following is true?

    a. The page size would have to be bigger
    [✓ b. The page directory would have to be bigger]
    c. You couldn't use large pages
    d. You could only use large pages
    e. None of the above

would need 2x the page dir space

(The rest of this page is intentionally blank - don't worry!)

Question 4: Advanced Page Replacement in xOS

xOS uses a novel approach to page replacement, based on a new policy called **2Q**. With 2Q, there are two queues that are used to manage memory. The **top** queue manages one chunk of memory, while the **bottom** queue manages the rest. When a page is first accessed, it is brought into the **bottom** queue (and placed at the "first-in" side). This queue is managed in **FIFO** manner (first-in first-out). If a page on the bottom queue is accessed (before being evicted from the bottom queue), it is moved to the **top** queue (at the "most recent" side). The top queue is managed in an **LRU** manner. When a page is evicted from the top queue, it moves back into the bottom queue (at the "first-in" side).

25. Assuming there are 4 total pages with **2Q** replacement (2 for the top, 2 for the bottom), how many misses will be generated by the following access pattern: 1, 2, 3, 4, 1, 2, 3, 4

    a. 0
    b. 1
    c. 4
    [✓ d. 8]
    e. none of the above

    1,2 => bottom Q (miss, miss)
    3,4 => bottom Q (replacing 1,2) (miss, miss)
    repeat => 8 misses

26. If we had instead configured the 4 page cache to simply use **LRU**, how many misses would have been generated by that same pattern? (1, 2, 3, 4, 1, 2, 3, 4)

    a. 0
    b. 1
    [✓ c. 4]
    d. 8
    e. none of the above

    first four : misses
    then all in cache : hits!

27. Assume again we have 4 pages with **2Q** replacement (2 for the top, 2 for the bottom). Now analyze this access pattern: 1, 2, 1, 2, 3, 4, 5, 6, 1, 2, 1, 2, 1, 2. How many misses are generated by this pattern?

    a. 0
    b. 1
    c. 4
    d. 8
    [✓ e. none of the above]

    6 misses

    first four : miss, miss, hit, hit (promote to upper Queue)
    next four : all misses (4×) (but top queue remains w/ 1,2 in it)
    then all hits

28. Assume the same pattern: 1, 2, 1, 2, 3, 4, 5, 6, 1, 2, 1, 2, 1, 2. How many misses would have been generated using just **LRU** replacement?

    a. 0
    b. 1
    c. 4
    [✓ d. 8]
    e. none of the above

    miss, miss, then hit, hit
    3,4,5,6 all miss but fill cache
    then miss, miss, hit, hit, hit, hit

27, 28 show the benefit of 2Q (!)

29. One issue with 2Q is the sizing of each queue; making the top queue bigger makes the bottom one smaller (and vice versa). Assume we again have four pages in memory. Assume also this access sequence:
1, 2, 3, 1, 2, 3, 4, 5, 6, 7, 1, 2, 3, 1, 2, 3

    What allocation of memory between the top queue (LRU) and bottom queue (FIFO) leads to the **most hits** for this access sequence?

    all misses {
    a. 4 in top, 0 in bottom
    b. 3 in top, 1 in bottom
    c. 2 in top, 2 in bottom

    7 hits [✓ d. 1 in top, 3 in bottom]
    6 hits e. 0 in top, 4 in bottom

30. Another issue with 2Q is exactly which replacement policies to use in each queue. Assume now that for both queues, we use FIFO replacement (instead of LRU for the top queue). Assuming 2 pages in the top queue and 2 in the bottom, how many misses occur during this access sequence? 1, 2, 3, 4, 1, 2, 3, 4
    a. 0
    b. 1
    c. 2
    d. 4
    [✓ e. None of the above]

31. Again assume a FIFO/FIFO 2Q, with 2 pages in each queue. How many misses occur during this sequence? 1, 2, 1, 2, 3, 4, 5, 6, 5, 6, 1, 2

    m m $\underbrace{h\ h}_{\text{promote}}$ m m m m $\underbrace{h\ h}_{\text{promote}}$ h h

    a. 2
    b. 4
    [✓ c. 6]
    d. 8
    e. None of the above

32. Overall, what could you say about the 2Q approach, as compared to usual single-queue approaches?
    a. 2Q is always better than a single-queue approach
    b. 2Q is always worse than a single-queue approach
    c. 2Q is identical to a single-queue approach
    d. 2Q is simpler than a single-queue approach
    [✓ e. None of the above]

    2Q is more complex, but can be better or worse depending on workload

Question 5: Multi-level Page Tables

x537 can be configured to use multi-level page tables; this is called, by the designers, “mode 3000” (the reason for this name is lost to history). This is bad news for both users and exam-takers, because no one really likes multi-level page tables.

In this question, we assume the following:

- The page size is an unrealistically-small 32 bytes
- The virtual address space for the process in question is 1024 pages, or 32 KB
- Physical memory consists of 128 pages

Thus, a virtual address needs 15 bits (5 for the offset, 10 for the VPN).
Thus, a physical address requires 12 bits (5 offset, 7 for the PFN).

The system assumes a multi-level page table. Thus, the upper five bits of a virtual address are used to index into a page directory; the single-byte page directory entry (PDE), if valid, points to a page of the page table; the upper-bit is the valid bit. Each page-table page holds 32 single-byte page-table entries (PTEs). Each PTE, if valid, holds the desired translation (physical frame number, or PFN) of the virtual page in question; the upper bit is the valid bit.

The format of a PTE is thus a valid bit followed by seven bits of the PFN:
VALID | PFN6 ... PFN0

The format of a PDE is essentially identical:
VALID | PT6 ... PT0

The full contents of physical memory are omitted for brevity; relevant portions are shown as need be.

33. First, assume you were instead using a **linear page table** (not multi-level). For the virtual address space described above, how many **pages** of memory would such a table use? Remember PTEs are 1 byte in size; pages are 32 bytes large.

a. 1

[✓ b. 32]

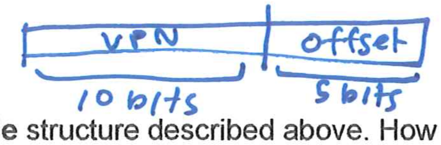
c. 64

d. 1024

e. none of the above

32 KB virtual address space

15 bits virt addr. ($2^{15} = 32$ KB)



2^{10} entries

$\frac{1024 \text{ bytes}}{32 \text{ bytes/page}} = 32$ pages

34. Now assume the **multi-level page table** structure described above. How many pages of memory would a multi-level structure use, assuming the address space has just one valid page in it?

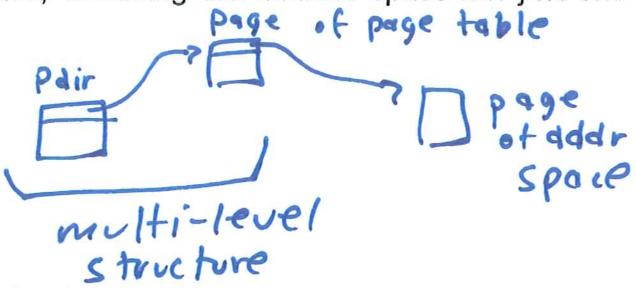
a. 1

[✓ b. 2]

c. 32

d. 1024

e. none of the above



35. Which of the following is true about multi-level page tables?

a. They are always smaller than linear page tables

[✓ b. They can be bigger than linear page tables]

c. Lookups are faster than in linear page tables

d. They can only be two levels deep

e. None of the above

if fully allocated, extra page dir => can be bigger

36. The **page directory** is found in **page 81** of physical memory. Its contents are:

```
84 fb 8b b7 e4 fe 7f ec b9 d2 d9 ca 90 b4 f0 86
e3 7f 99 a4 a2 7f aa 83 d3 e7 7f bb eb a5 82 7f
```

How many **invalid** entries does the page directory contain?

a. 0

b. 1

[✓ c. 5]

d. 21

e. None of the above

invalid => first bit is zero

e.g. 0x7f

0111 1111

37. The first entry of the page directory is **0x84**. Which physical frame (in decimal) does this refer to?

- a. 0
- [✓ b. 4]
- c. 8
- d. 84
- e. None of the above

$\underbrace{\boxed{1}}_{\text{valid}}\underbrace{0000100}_{\text{PFrame}}$

38. Let's examine the contents of physical page 2 of the system:

```
7f 7f 7f 7f 7f 7f 7f 7f 7f 7f 7f 7f 7f 7f ab 7f
7f 7f 7f 7f 7f c6 7f 7f 7f 7f 7f 7f 7f 7f 7f 7f
```

It is a page of the page table (i.e., it contains page table entries). How many **valid** entries does this page have on it?

not valid : 0x7f

- a. 0
- b. 1
- [✓ c. 2]
- d. 31
- e. None of the above

39. If page 2 (above) is a valid page of the page table, the page directory must refer to it. Which of the following entries in the **page directory** refers to page 2?

- a. 0x84
- b. 0x8b
- c. 0xd2
- [✓ d. 0x82]
- e. 0x7f

$0x82 \Rightarrow \underbrace{1}_{\text{valid}}\ \underbrace{000\ 0010}_{\text{page frame } \boxed{2}}$

40. Finally, let us examine the contents of physical page 93 of the system:

```
0  1  2  3  4  5  6  7  8  9  10 11
15 08 08 00 11 15 07 02 12 12 0c 03 1c 1a 13 11
12 00 1a 16 15 05 17 11 10 0b 10 18 1e 09 09 15
```

Should have been virtual

Assume this is simply a page of the address space. If the [physical address] that refers to a byte on this page is **0x48cb**, which byte value would be returned by a load to that address?

$\overset{4}{100}\ \overset{8}{1000}\ \overset{c}{11\,0}\boxed{0\ \overset{b}{1011}}$

- a. 0x00
- b. 0x12
- [✓ c. 0x03]
- d. 0x15
- e. None of the above

offset is last five bits

=> entry 11, starting at 0

Question 6: x537 TLBs

The x537 has TLBs (translation look-aside buffers) in each CPU's MMU to speed up memory access. The system is configured to use a **16-bit virtual address space**, with **4-KB pages**. Physical memory is **128 KB**. x537 is configured to use a **linear page table**.

Assume these are the contents of the (linear) page table of a currently running process:

entry	
0	0x8018
1	0x0000
2	0x0000
3	0x800c
4	0x8009
5	0x0000
6	0x801d
7	0x8013
8	0x0000
9	0x801f
a	0x801c
b	0x0000
c	0x800f
d	0x0000
e	0x0000
f	0x8008

VPN	offset
4 bits	12 bits ($2^{12} = 4KB$)

$\Rightarrow 2^4 = 16$
virtual pages

Each line represents a **page table entry** (**PTE**). The topmost bit is the valid bit; the rest, if valid, is the physical frame number (**PFN**) of the translation.

41. To begin, let's start with a simple translation. The process accesses virtual address **0xce16 (ce16 in hex notation)**. What physical address does this translate to?

   a. 0xce16
   b. 0x18e16
   [✓ c. 0xfe16]
   d. 0xcece
   e. Fault (not a legal virtual address)

$\underbrace{c}_{VPN}$ e16

c th entry in page table:

$0x\underbrace{8}_{\text{top bit is valid}}\underbrace{00F}_{\text{rest if PFN}}$

42. Let's do one more translation, but in reverse. Assume that physical address **0x$\boxed{18}$398** has been accessed. What virtual address was issued to result in this physical address access?

   a. 0x18398
   [✓ b. 0x0398]
   c. 0x98
   d. 0xff98
   e. None of the above

table above has one entry w/ PFN 0x18 ⇒ 0th entry

thus, $0x\underbrace{0}_{VPN}\underbrace{398}_{offset}$

43. The next few questions deal with the TLB. Assume we have a four-entry TLB. Its entries are replaced in LRU fashion, i.e., when the TLB is full, the least-recently-used entry is the one that gets removed, to make room for the new entry.

We now have the following stream of four virtual address references from the process above (you know, the one whose page table is on the previous page):

```
0xce16, 0xce17, 0xce18, 0xce19
```

How many **TLB hits** will this reference stream generate, assuming the TLB was empty at the start?

all refer to Virt Page 0x C

thus miss, hit, hit, hit

a. 0
b. 1
c. 2
[✓ d. 3]
e. None of the above

44. Now assume the following series of virtual address accesses, again from the same process and again using the page table above:

```
0x0000, 0x3001, 0x4002, 0x6003, 0x0001, 0x3002, 0x4003, 0x6004
```

How many **TLB hits** will this reference stream generate, again assuming the TLB was empty at the start?

Virt Page 0, 3, 4, 6
m m m m

0, 3, 4, 6 } 4 hits
h h h h

a. 0
b. 2
[✓ c. 4]
d. 6
e. None of the above

45. Each TLB entry, in x537, has the following contents:

```
[ valid bit | virtual page number | physical frame number ]
```

Which of the following is false?

accepted either →

[✓ a.] The TLB can never be empty (i.e., it can never have zero valid translations)
b. The TLB should be flushed (i.e., valid bits set to zero) when xOS switches between processes
[✓ c.] The TLB valid bit is equivalent to the valid bit in the page table
d. The TLB is located in the MMU of the x537 processor
e. None of the above (the above statements are all true)

46. Assume the following contents of the 4-entry TLB, with the 1st entry at the top, and the last (4th entry) at the bottom:

```
[ 1 | 0x3 | 0x0c ]
[ 1 | 0x0 | 0x08 ]
[ 1 | 0x4 | 0x09 ]
[ 1 | 0x6 | 0x1d ]
```

Unfortunately, there may be a bug in the TLB! Which entry, assuming the page table above, contains a bad entry?

a. The 1st

[✓ b. The 2nd]

c. The 3rd

d. The 4th

e. None of the above (they all are good)

0th v Page => 0x 18 (in first entry of page table)

47. Assume now the following contents:

```
[ 0 | 0x7 | 0x13 ]
[ 1 | 0x9 | 0x1f ]
[ 1 | 0x3 | 0x0c ]
[ 0 | 0xf | 0x08 ]
```

] in TLB, valid!

With the following virtual address accesses, how many **TLB hits** will be generated?

```
0x9000, 0x9001, 0x9002, 0x9003
```

all ref virt page 0x9

all hits!

a. 1

b. 2

c. 3

[✓ d. 4]

e. None of the above

48. Finally, the xOS designers are trying to convince the x537 processor team to switch to a **software-managed TLB.** Which of the following is true about software-managed TLBs? (choose the best answer)

[✓ a. It allows more flexible page table structures]

b. It makes the TLB miss lookup faster

c. It makes the MMU more complex

d. It adds more bits to each TLB entry

e. None of the above (they are all false)

hardware doesn't need to do lookup, so OS controls structure of table