Virtualization

Q1. The minimum number of page faults required is **2**.

(Any idea why we say minimum here?)

1. Fault #1: Loads the Leaf Page Table

- **Hardware Action:** The MMU misses in the TLB, starts at the Page Directory (which is already in memory). It successfully locates the Page Directory Entry (PDE) for the address. However, that PDE points to a **Leaf Page Table** that is currently in swap storage (`Present = 0`).
- **Exception:** The MMU triggers a Page Fault.
- **OS Action:** The OS Page Fault Handler identifies the missing Leaf Page Table and loads it from swap into physical memory. It then updates the Page Directory Entry.
- **Restart:** The OS returns from the interrupt, and the CPU **restarts** the `MOV` instruction from the very beginning.

2. Fault #2: Load the Data Page

- **Hardware Action:** The instruction restarts. The MMU misses in the TLB, looks up the Page Directory (Hit) and now finds the Leaf Page Table in memory (Hit). It then looks up the specific Page Table Entry (PTE) for the data address, but finds that the **actual Data Page** is in swap storage (`Present = 0`).
- **Exception:** The MMU triggers a second Page Fault.
- **OS Action:** The OS loads the **Data Page** containing the value at `[0x7FF00123]` from swap into physical memory and updates the PTE.
- **Restart:** The OS returns from the interrupt, and the CPU **restarts** the `MOV` instruction for a third time.

3. TLB insert

- **Hardware Action:** The instruction restarts. The MMU misses in the TLB, looks up the Page Directory (Hit) and now finds the Leaf Page Table in memory (Hit). It then looks up the specific Page Table Entry (PTE) for the data address, and finds it (`Present = 0`), inserts that entry into the TLB and **restarts** the `MOV` instruction for a third time.

4. Actual Load

- **Hardware Action:** The instruction restarts. The MMU hits in the TLB, computes the physical address, and loads the actual value at `[0x7FF00123]` into the register.

Q2.

Attempt 1: Discovery of Missing Leaf Page Table

1. **TLB Lookup: Miss** 5ns
2. **Page Directory Access:** 100ns
3. **Page Fault 1:** 20ms

- **Time for Attempt 1:** 20ms+105ns

Attempt 2: Discovery of Missing Data Page

1. **TLB Lookup: Miss** 5ns
2. **Page Directory Access:** 100ns
3. **Leaf Page Table Access:** Look up the PTE 100ns
4. **Page Fault 2:** 20ms for data page

- **Time for Attempt 2:** 20ms + 205ns

Attempt 3: TLB insert

1. **TLB Lookup:** 5ns
2. **Full Page Walk:** 100ns + 100ns; insert into TLB

- **Time for Attempt 3:** 205ns

Attempt 4: Actual load

1. **TLB Lookup:** 5ns
2. **Data Load:** 100ns.

- **Time for Attempt 4:** 105ns

Total: 40ms + 620ns = 40,000,620ns

Q3.

Expected/average page fault service time = 0.6*20 + 0.4*70 ms = 40ms
Attempt1: 40ms + 105ns
Attempt2: 40ms + 205ns
Attempt3: 205ns
Attempt4: 105ns
Total: 80ms + 620ns = 80,000,620ns

Q4.

Even a 40% probability of the victim blocks being dirty doubles the access time. Therefore, the use of a page/swap daemon can be hugely beneficial to remove that work from the critical path of the page fault handler.

Concurrency

1. The `my_node` is allocated on the stack of the waiting threadA. Once cond_wait exits, that memory is invalid. If cond_signal or another cond_wait attempts to dereference that address, it will either return a use-after-free error or result in memory corruption. How can that happen? A regular signal will use that my_node before it wakes up threadA, so that race condition needs an extra condition: the spurious wakeup. ThreadA waits, gets a spurious wake, exits the wait, the stack entry for that cond_wait is freed now. If cond_wait was called in a while loop (as it should be), it checks the condition and maybe moves on OR creates another my_node and waits again. The memory is now either free or reallocated for the stack of threadA. When cond_signal is called, it accesses bad memory. Of course, other threads may have also called cond_wait on that cv by now and inserted their my_nodes after this stale memory from threadA.
2.
```
cond_wait():
      node_t my_node = malloc(sizeof(node_t));
      my_node->tid = gettid();
      my_node->next = NULL;

cond_signal():
      unpark(to_wake->tid);
      free(to_wake);
```
3. If threadA is preempted just after mutex_unlock(user_lock) and before it can call park(), and threadB calls cond_signal() and unpark on threadA, then the signal is lost. Eventually threadA parks and never wakes up
4. Insert a setpark() earlier in the cond_wait()
5. The key idea is to reduce lock contention (on c->queue_lock), i.e., minimize the time we hold it. So, it's best to not walk the entire linked list while holding onto the lock.

```
void cond_broadcast(my_cond_t *c) {
    mutex_lock(&c->queue_lock);
    node_t *curr = c->head;
    c->head = NULL;
    c->tail = NULL;
    mutex_unlock(&c->queue_lock);
    while (curr != NULL) {
        node_t *to_wake = curr;
        curr = curr->next;
        unpark(to_wake->tid);
        free(to_wake);
    }
}
```

MPs

1. Code running in user-mode is fully isolated from other processes. The only real “damage” a bug in that code can do is corrupt another thread’s memory (within the process), for eg., its stack. OTOH, code running in the kernel can mostly do whatever it wants…it can corrupt memory, hold onto the CPU for any length of time by disabling interrupts, etc. Also, when it does something bad, it can result in a crash or hang. Or worse, silent data corruption!
2. The copy_to/from_user() api: (1) verifies that the user-provided pointer is a valid user-space address, the memory is accessible to that current process. (2) handles invalid addresses correctly–returns fault if the page is swapped out, zero/unmapped, etc. (3) handles CPU arch specific stuff (4) performs size checks to ensure no buffer overflow.
    a. IOW, memcpy() in the kernel should be used only for copying between two trusted kernel-space pointers.
3. **Pro**: Fast because a cache of objects of that exact size is pre-created. It has a few “free” objects in that cache, which it can (re)allocate quickly. This performance benefit is more pronounced in multi-core systems, because it creates a per-CPU cache. **Con**: Memory is wasted–the “free” objects in a cache multiplied by the #cores.