a) Case 1: On each context switch, loading a new %cr3 value flushes all non-global TLB entries, invalidating user-space mappings like those at 0x08040000. This causes frequent TLB misses as the new process refills its TLB.
   Case 2: Each address space has a unique Process Context ID, allowing TLB entries to be tagged by both virtual page number and PCID. Thus, user-space entries can persist across context switches, avoiding unnecessary flushes and improving performance.
b) When the process accesses address 0xBFFFEFFC, that address is just below the current stack range (0xBFFFF000–0xBFFFFFFF). Because the stack grows downward, this means the program is trying to use more stack space than is currently mapped. Since that address doesn't yet have a valid mapping, the CPU raises a page fault. Control switches from user mode to the kernel, and the processor enters supervisor mode so the kernel can handle the fault. The kernel's page-fault handler checks the faulting address and sees that it's close enough to the current stack to be a normal case of stack growth. In that case, the kernel allocates a new physical page and adds a new page table entry (PTE) for it, extending the stack downward. In BSD, this is done by the vm_fault() function, which updates the vm_map_entry for the stack region to include the new page. Once the mapping is added, the kernel switches back to user mode and retries the instruction that caused the fault. This time, the instruction succeeds because the page now exists. If the address had been too far below the stack limit, the kernel would not allocate a page; instead, it would send the process a SIGSEGV
c) When Counter-Strike was started, only the parts of the program that were immediately needed were loaded into memory. The rest of the executable file remained on the network server and would be loaded later using demand paging through mmap(). After the network cable is unplugged, the game keeps running normally until it tries to use code or data from a part of the executable that hasn't been loaded yet (by allocating a physical page frame and creating a PTE). This is from a backing store. Eventually, the CPU generates a page fault, and the kernel attempts to read a missing page from the server. Because the network connection is gone, the kernel can't fetch the data and cannot resolve the fault. As a result, the process receives a SIGBUS and crashes.
d) When process 6767 runs, the kernel loads the executable file into memory by mapping its text (code) segment into the process's address space using the page tables. These mappings are read/execute only, however, not writable. In the page tables, each mapping has permission bits that specify whether a page can be read, written, or executed. Pages containing executable code have the write bit cleared to secure itself against modification while the program is running (like process 555). Because the file's text pages are currently mapped as executable, the kernel locks the file to prevent any process from opening it for writing. When process 555 is called, the system call fails with the error ETXTBSY (text file busy).
e) 0x0000 0000 0000 0000 to 0x0000 0000 001F FFFF is all unmapped.
   2 X 1024 X 1024 bytes = 2,097,152 bytes = 0x200000 bytes = 2MB
   0x200000 - 0x000001 = 001F FFFF
   Each PMD entry reference 2^21 = 2MB in a X86 4 page structure. This is how much virtual memory 1 PDE controls.