2) For questions A-E below, each of which describe a Page Fault situation, describe in detail how the page fault is resolved, keeping in mind our fundamental outcome categories of major fault, minor fault, or delivery of a specific signal (tell me which one). We will use the X86-32 Linux reference model, and all of these Page Faults are from user-mode processes.

A) The faulted address is a write access to 0xBFFF2FFC. The current beginning of the stack region is 0xBFFF3000.

The minor fault occurs as the target address of the write is one lower than the current beginning of the stack region. This effectively would add a page to the stack region via mremap if the region before the faulted address has the GROWSDOWN property because of the target address's close proximity to the current beginning.

B) The faulted address is a read access and falls within a file-mapped region. Neither our process nor anything else on the system has ever read that part of the file.

Because that part of the file has never been read, it could result in a null pointer. This would be considered a major fault as it would invoke demand paging. If you read into a part of a file that has never been read, a PTE will be created to compensate.

C) As above, but the file in question resides on a USB stick that was mounted as a volume into our filesystem. The user has accidentally unplugged the stick.

Since it involves a USB stick, this might seem like a major fault because of the I/O error, but it is a minor fault as it would result in just a SIGBUS. When the USB stick is removed, the data in the file is not accessible, the I/O operation is not required and does not occur.

D) The faulted address is an instruction fetch. The VA corresponds to the BSS region of the process.

The protections of the BSS region are PROT_READ|PROT_WRITE. It cannot be executed, resulting in a SIGSEGV.

E) The faulted address is a read and falls within an mmap region with PROT_READ|PROT_WRITE,MAP_SHARED|MAP_ANONYMOUS Our process is the only process with this region (it is not actually shared with anyone else) and we have never accessed this virtual page before.

Given that this virtual page has never been accessed and MAP_SHARED is on, this would suggest that the address we are attempting to read from corresponds to a shared, zeroed out page frame, a minor page fault and a new page frame is assigned.

2F) We have mmap'd a file which is currently 65536 bytes long with
char *p=mmap(NULL,32768,PROT_READ|PROT_WRITE,MAP_SHARED,0,fd).

Another process has recently written 4096 bytes to offset 16384 of this file. Then we look at the memory region, e.g. with p[16384]. Describe the data structures which allow the kernel to present a consistent view between the memory mapping and the file and ensure that what we see in memory is what was written to the file and vice-versa.

The data structures that allow the kernel to have a consistent view between memory mapping and the corresponding file is through the address_space structure. The file's inode struct directs us to the address_space. In page_tree of the address_space, two relevant flags are types of trees, one of them being the flexible radix-64 tree (radix_tree_root struct) which, given an offset into the file,m finds the struct page corresponding to that part of the file, and the other being the prio_tree_root struct responsible for private and shared mappings. In the struct page is the mem_map referring to the page frame which is shared between processes.