4.2.1

1. PTE can be get from taking the address of pagetable[i] (where i is from 0 to 512), and its physical address can be got by the macro `PTE2PA()` with parameter "the pointer point to the address of pagetable[i]" .
   As for virtual address va, there are 3 levels of page table, for every PTE we can get an entry number for each level, and va can be get by concatenate the 3 entry numbers of PTE. Give the level of page table to the macro `PXSHIFT()` getting how many bit we have to shift for the current level, shift the entry for each level, and repeat it until it reaches the PTE, then the sum is va.
2. The page table entries with higher va corresponds to the stack section, while with the lower va corresponds to the text section.
   The text section is usually loaded at virtual address = 0, so the page table entries with lower va corresponds to the text section, and the stack section is the only section used in mp2_1 besides the text section, so the page table entries with higher virtual address corresponds to the stack section.
   From the output, we can observe that those with lower virtual address has the flag "PTE_X", which means it is executable. So it corresponds to the text section.
3. (a) inverted page table use less memory space
   (b) inverted page table usually takes longer time to query the page table

4.3

1. In step 5 the page table is changed. Before being modified, the page table entry stores the block number where the "backing store" saves the page, as for the flags, PTE_S is set and PTE_V is unset. After being modified, the page table entry stores the physical address about the "brought in" page, and for the flags, PTE_S is unset and PTE_V is set.
2. "step 1": Check the reference is valid or not by PTE_V in the page table.

   "step 2": If the reference is invalid then terminate the process. If it is valid but still in the "backing store" then move the page in. The page fault will invoke `handle_pgfault()` in paging.c.

   "step 3": Ask for a free frame by `kalloc()`

   "step 4": Read the page from disk by calling `read_page_from_disk(ROOTDEV, (char *)pa, blockno)`, where ROOTDEV means the backing store, pa is the free frame we asked in step 3, and blockno is the block number where the missing page is saved. To get blockno, I

call the macro PTE2BLOCKNO() with parameter PTE, where PTE is the page table entry. After reading, free the storage block by calling `bfree_page(ROOTDEV, blockno)`.

“step 5”: Update the PTE with the physical address, unset PTE_V, and set PTE_S.

“step 6”: Return to the instruction that causes the page fault.