5.1

Q1

Below is the pseudocode for printing out a page table:

Algorithm 1:

```
// Index starts from 1.
define PTE_V (1 << 0);
define PGSHIFT 12 ;
define PXSHIFT(level) (PGSHIFT + (9 * (level)));
define PTE2PA(pte) (((pte) >> 10) << 12);
;
Function vmprint_recur (pagetable_t pagetable, int va, int level)
    for i = 0 $\to$ 512 do
        pte = &(pagetable[i]);
        if !(pte & PTE_V) then
            continue;
        end
        ;
        va = va | (i << PXSHIFT(level));
        pa = PTE2PA(*pte);
        ;
        print(pte, va, pa);
        ;
        if level $>$ 0 then
            vmprint_recur (pa, va, level);
        end
        ;
        va &= $\sim$(i << PXSHIFT(level));
    end
end
Function vmprint (pagetable_t pagetable)
    vmprint (pagetable, 0, 2);
end
```

- `pte`:
  At each level, print all the valid entries. The value of `pte` is simply the address of `pagetable[i]`, where i is the index.

- `pa`:
  We can get `pa` by the built-in macro `PTE2PA`. The reason how this work is:

  - The first 10 bits of `pte` represent the flags.
  - The first 12 bits of `pa` are reserved for offset within a page.

  Hence, `pa = (pte >> 10) << 12`

- `va`:
  According to the docs of `walk()` in xv6, the bits of `va` represent:

  - `0..11`: offset with in the page.
  - `12..20/21..29/30..38`: the index in page table of level 0/1/2, respectively.
  - `39..63`: zero.

  Thus the formula for `va` is `va =` $\texttt{i}_0$ `<< 12 |` $\texttt{i}_1$ `<< 12 + 9 |` $\texttt{i}_2$ `<< 12 + 9*2 | (offset)`, where i_k is the index of level k.

Q2

References

- The xv6 book
- Zhihu article

Answer

Consider the following output:

```
+-- 0: pte=0x0000000087f57000 va=0x0000000000000000 pa=0x0000000087f53000 V
|   +-- 0: pte=0x0000000087f53000 va=0x0000000000000000 pa=0x0000000087f52000 V
|       +-- 0: pte=0x0000000087f52000 va=0x0000000000000000 pa=0x0000000087f54000 V R W X U
|       +-- 1: pte=0x0000000087f52008 va=0x0000000000001000 pa=0x0000000087f51000 V R W X
|       +-- 2: pte=0x0000000087f52010 va=0x0000000000002000 pa=0x0000000087f50000 V R W X U
+-- 255: pte=0x0000000087f577f8 va=0x0000003fc0000000 pa=0x0000000087f56000 V
    +-- 511: pte=0x0000000087f56ff8 va=0x0000003fffe00000 pa=0x0000000087f55000 V
        +-- 510: pte=0x0000000087f55ff0 va=0x0000003fffffe000 pa=0x0000000087f76000 V R W
        +-- 511: pte=0x0000000087f55ff8 va=0x0000003ffffff000 pa=0x0000000080007000 V R X
```

The correspondence from each page to the memory sections should be following:

- `va=0x0000000000000000`: **text** and **data** sections.
- `va=0x0000000000001000`: **guard page**.
- `va=0x0000000000002000`: **stack**.
- `va=0x0000003fffffe000`: **trapframe**.
- `va=0x0000003ffffff000`: **trampoline**.

Reasonales

- Since the **text** section starts at 0, it's reasonable that the `va=0x0000000000000000` corresponds to **text**.
- The `va=0x0000000000001000` has no `U` flag, which implies it corresponds to **guard page**. Also, it has exactly one page size, which is predictable.
- Therefore, the **data** section, which is below **guard page**, should correspond to `0x0000000000000000` since there is no other pages left.
- Likewise, the **stack** corresponds to the page just above **guard page**, `0x0000000000002000`.
- **Trampoline** and **trapframe** are the last two pages of the virtual memory. They should be located at `MAXVA - 2 * PGSIZE = a=0x0000003fffffe`, which matches the result.

Evidence

First, we can add a few lines in `vmprint`:

```
printf("size: %p\n", myproc()->sz);
printf("trapframe: %p\n", myproc()->trapframe);
printf("sp: %p\n\n", myproc()->trapframe->sp);
```

The result is:

```
size: 0x0000000000003000
trapframe: 0x0000000087f76000
sp: 0x0000000000002fd0
```

The address of **trapframe** is the same as the `pa` of `va=0x0000003fffffe`, which we have stated corresponding to **trapframe** section. Also, the `sp` (which is a virtual address) lies in the **stack** section, i.e. `va=0x0000000000002000`.

The second examination we can do is to trace through the `exec()` function in `kernerl/exec.c`. There should be some solid evidence:

- line 38: calling the function `proc_pagetable`, which map **trapframe** to `0x0000003fffffe000` and **trampoline** to `0x0000003ffffff000`.
- line 57: calling `loadseg`, which loads the binary file (i.e. **text** and **data**) into memory, starting from 0.
- line 71-74: calling `uvmalloc(pagetable, sz, sz + 2*PGSIZE))` and `uvmclear(pagetable, sz-2*PGSIZE)`, which allocate **guard page** and then clear the `U` flag, respectively.
- line 79-98: pushing `argv` into the **stack**. **Stack** is one page above **guard page**.

Q3

(a)

First we consider the case in `mp2_1`, namely there are 5 pages needed for each process.

- Multilayer page table (in xv6): a process needs (at least) 5 active page table:
  - Layer 2: 1.
  - Layer 1: 2.
  - Layer 0: 2.

  That should be $5 * 512$ `PTEs`, including those invalid ones.
- Inverted page table: only (at least) 5 `PTEs` required per process.

It seems that the ratio of space usage between the two method is 512, i.e. the size of one page table in `xv6`. However, if the page allocated for a process increase to $512 + 2$ (including trampoline and trapframe), the number of page tables will maintain the same (in the best case, since a layer 0 page table can store at most 512 `PTEs`), whereas the number of `PTEs` increase to $512 + 2$ in the inverted page table. In that case, there is no significant difference in memory usage.

To sum up, **inverted** page tables use less memory in **sparse** case, where the pages used by each process are less. On the contray, as the pages allocated goes up, the cost of **multilayer** page tables can be amortized, so that the two method have the same space efficiency.

(b)

Here we consider no TLBs is working on the hardware. The time complexity for translating virtual memory in **multilayer** page tables is always $O(1)$, since there are merely array indexing. The **inverted** page tables, on the other hand, have to search for the given `pid` and `va` in the table, which results to $O(n)$ time complexity in average cases, where n is the size of the table. Thus **multilayer** page tables have better performance.

1 5.2

Q1

The page table is modified in step 5. The `blockno` previouly stored should be replaced with `pa`, and the `PTE_FLAGS` are modified as following:

- `PTE_V`: set to 1.
- `PTE_S`: set to 0.
- Others: remain the same.

Q2

1. Reference: RISC-V tries to translate and fetch a given `va`. The address of translating page table is stored in `SATP` register.

2. Trap: RISC-V will jump to the address stored in `STVEC` register, which is either `kernelvec` or `uservec`. It also switch into supervisor mode to write to registers. After several steps (such as store the context in `p->trapframe`), the `kernaltrap` or `usertrap` function is called to handle the trap. In the case of user space page fault, `usertrap` will be called and `SCAUSE` register is set to 13 or 15.

3. Page is on backing store: In `usertrap` function, page fault will be handled when ever `r_scause() == 13 || r_scause() == 15`. Then we get the value of `va` from `STVAL` register, which is stored during trap step. Then the handler should call `pte = walk(myproc()->pagetable, va, 0)` to get the corresponding `PTE`. Finally we check the `PTE_S` bit on that `PTE`.

4. Bring in missing page: the procedure can be divide into following parts:

   - Get the blockno by `PTE2BLOCKNO(*pte)`.
   - Allocate a `pa` using `mem = kalloc()`.
   - Copy the data with following code:

     ```
     begin_op();
     read_page_from_disk(ROOTDEV, mem, blockno);
     bfree_page(ROOTDEV, blockno);
     end_op();
     ```

5. Reset page table: calling `mappages(pagetable, PGROUNDDOWN(va), PGSIZE, (uint64)mem, PTE_FLAGS(*pte))`.

6. Restart instruction: after calling `usertrap` (step 3,4), `usertrapret` and `userret` is executed to restore the context and jump back to the previous program counter (if process is not killed), which is stored in `p->trapframe->epc` previously.