1. Every CPU needs two registers: base register and bound register. In address translation, it hardware adds virtual address with the value held in base register, and sends the result which is physical address to the memory. If the physical address exceeds the bound value in the bound register, the CPU will trigger an exception and abort the process. For the operating system, it has to do the following tasks.
   (1) It must allocate space for the address space in memory.
   (2) When the process is terminated, the OS should retrieve its memory for other processes or the OS itself.
   (3) During context switch, the OS should save and restore the base and bound registers.
   (4) It must provide exception handlers to deal with misbehaving processes.

2. (1) size of chunks:

   Segments have unfixed length.

   Page size is fixed and is determined by the operating system.

   (2) management of free space:

   Segmentation uses a free list management algorithm like best fit, worst fit, first fit or buddy algorithm.

   Paging keeps a list of free physical pages and keeps page table to map virtual pages to physical pages.

   (3) context switch overhead:

   In segmentation process, there should be a base address and offset to map the virtual address to the physical address. The overhead of translation is small.

   In the paging process, there should be a page table to map virtual pages to PFN. The overhead of translation is big because the size of a page table may be huge.

   (4) fragmentation:

   Segmentation has external fragments and doesn't have internal fragments.

   Paging has internal fragments but doesn't have external fragments.

   (5) status bit and protection bits:

   In segmentation, there are extra protection bits for each segments to mark whether it can be read, written to or executed.

   In each PTE, there is a valid bit to identify whether the address translation is valid, a present bit shows the location of the page, a reference bit to see whether the page is visited, a protection bit to show if a page can be read, written to or executed. In segmentation, there are extra protection bits for each segments to mark whether it can be read, written to or executed.

3. Page size: 8KB = 2^{13}B, so offset needs 13 bits, and 46-13=33 bits for page numbers

   Max number of entries in a page table: 2^{13}B/4B = 2^{11}, so each level of page table needs 11 bits

   So number of levels is 33/11 = 3

Level 1 (11 bits)	Level 2 (11 bits)	Level 3 (11 bits)	Offset (13bits)

4. (a) Page size: 2^{12}B = 4KB

   Max page table size=entry number * entry size=2^{20}*4B=4MB

(b) 0xC302C302 = $1100001100\ 0000101100\ 001100000010_2$

So first level page number=$1100001100_2=780_{10}$, offset=$001100000010_2=770_{10}$

0xEC6666AB = $1110110001\ 1001100110\ 011010101011_2$

So second level page number=$1001100110_2=614_{10}$, offset=$011010101011_2=1707_{10}$

5.

default_pmm.c × best_fit_pmm.c ×

```c

//----------------------合并空闲块-------------------

//Check whether it can be merged with the previous free blocks
//Let list entry "le" point to the previous list entry of base page
list_entry_t* le = list_prev(&(base->page_link));
//If if is not a free page block
if (le != &free_list) {
    //p is the page address that le points to
    p = le2page(le, page_link);
    //If the base page is after the previous page
    if (p + p->property == base) {
        //Merge the base page to p
        p->property += base->property;
        //Delete the base page
        ClearPageProperty(base);
        list_del(&(base->page_link));
        //Let p be the new base page
        base = p;
    }
}

//Check whether it can be merged with the next free blocks
//Let list entry "le_next" point to the next list entry of base page
list_entry_t *le_next = list_next(&(base->page_link));
//If it is not a free page block
if (le_next != &free_list) {
    //p is the page address that le_next points to
    p = le2page(le_next, page_link);
    //If the base page before the next page
    if (base + base->property == p) {
        //Merge the p page to base page
        base->property += p->property;
        //Delete the p page
        ClearPageProperty(p);
        list_del(&(p->page_link));
    }
}

//---------------------------------------------------
```

ljj11912021@ljj11912021-virtual-machine: ~/Desktop/l...

```
        |_|

Platform Name          : QEMU Virt Machine
Platform HART Features : RV64ACDFIMSU
Platform Max HARTs     : 8
Current Hart           : 0
Firmware Base          : 0x80000000
Firmware Size          : 120 KB
Runtime SBI Version    : 0.2

MIDELEG : 0x0000000000000222
MEDELEG : 0x000000000000b109
PMP0    : 0x0000000080000000-0x000000008001ffff (A)
PMP1    : 0x0000000000000000-0xffffffffffffffff (A,R,W,X)
os is loading ...
memory management: default_pmm_manager
physcial memory map:
  memory: 0x0000000007e00000, [0x0000000080200000, 0x0000000087ffffff].
check_alloc_page() succeeded!
```

6.

default_pmm.c | best_fit_pmm.c | pmm.c

```c
static struct Page *
best_fit_alloc_pages(size_t n)
{
    assert(n > 0);
    //If n > nr_free, memory space is too large, so return null
    if (n > nr_free) {
        return NULL;
    }

    struct Page *page = NULL;
    list_entry_t *le = &free_list;

    //Go through the whole list
    while ((le = list_next(le)) != &free_list) {
        //find the page address that le points to as p
        struct Page *p = le2page(le, page_link);
        //If p page's size is big enough
        if (p->property >= n) {
            //When page is not allocated with space or
            //its size is bigger than the pointed page
            if (page == NULL || p -> property < page -> property) {
                //Since it is best fit, we want the smaller size
                //So make page equals to p
                page = p;
            }
        }
    }

    if (page != NULL) {
        list_entry_t* prev = list_prev(&(page->page_link));
        //Delete the free page allocated just now
        list_del(&(page->page_link));
        //The size of page is too big
        if (page->property > n) {
            //Create a new page p whose address is n more
            struct Page *p = page + n;
            //Adjust its size
            p->property = page->property - n;
            SetPageProperty(p);

            //Insert the extra memory to the space after the original allocated page
            list_add(prev, &(p->page_link));
        }

        //Recalculate free space
        nr_free -= n;
        //Delete page property
        ClearPageProperty(page);
    }
    return page;

}
```

ljj11912021@ljj11912021-virtual-machine: ~/Desktop/lab6/lab6

```
 | |  | | '_ \ / _ \ '_ \ \___ \|  _ < | |
 | |__| | |_) |  __/ | | |____) | |_) || |_
  \____/| .__/ \___|_| |_|_____/|____/_____|
        | |
        |_|

Platform Name          : QEMU Virt Machine
Platform HART Features : RV64ACDFIMSU
Platform Max HARTs     : 8
Current Hart           : 0
Firmware Base          : 0x80000000
Firmware Size          : 120 KB
Runtime SBI Version    : 0.2

MIDELEG : 0x0000000000000222
MEDELEG : 0x000000000000b109
PMP0    : 0x0000000080000000-0x000000008001ffff (A)
PMP1    : 0x0000000000000000-0xffffffffffffffff (A,R,W,X)
os is loading ...
memory management: best_fit_pmm_manager
physcial memory map:
  memory: 0x0000000007e00000, [0x0000000080200000, 0x0000000087ffffff].
check_alloc_page() succeeded!
```