CS302 OS Week6 Assignment - Report

Name: 刘仁杰
SID: 11911808

1. Read Chapter 15 of "Three Easy Pieces"

- The hardware provides base and bound register as part of MMU of the CPU.
- When a user program is running, the hardware will translate each address, by adding the base register value to the virtual address generated by the user program.
- Also, the hardware will check if the address is legal, by using the bound register. If not, the hardware will throw an exception which will be caught by the OS and dispatch corresponding exception handler.
- In OS kernel mode, there are some hardware APIs for the kernel to change the value of base and bound register, which will enable OS to handle multiprocessing.
- In user mode, if the user program tries to change the value of base and bound register, the CPU hardware will also raise an exception and the OS will trigger corresponding handler.

2. Read Chapter 16 and 18 of "Three Easy Pieces"

1. Segmentation

- size of chunk: Variable-sized.
- management of free space: Use a free-list management algorithm that tries to keep large extents of memory available for allocation, such as best-fit, worst-fit and first-fit.
- context switch overhead: The segment registers must be saved and restored.
- fragmentation: Free of internal fragmentation while suffers from external fragmentation.
- status bits: There is no status bit in segmentation.
- protection bits: To support sharing, the basic implementation adds a few bits per segment, indicating whether or not a program can read or write a segment, or perhaps execute code that lies within the segment. Then the hardware will not only check if the want-to-access address is within bound, also should be permissible.

1. Paging

- size of chunk: Divide the process space into fixed-sized units.
- management of free space: More simple, OS keeps a free list of all free pages for this, and just grabs the first several pages that satisfy the demanded size off of this list.
- context switch overhead: On context switch, just simply changes the pointer to the page table.
- fragmentation: No external fragmentation, have internal fragmentation.
- status bits: There are 9 status bits, indicate D (dirty), A (accessed), V (valid), G (Global), U (User) and RSW
- protection bits: There are also protection bits, indicating whether the page could be read from, written to, or executed from. Again, accessing a page in a way not allowed by these bits will generate a trap to the OS.

3. How many levels of page tables would be required to map the entire virtual address space

There should be at least 3 levels of page tables.

Since Page size is of 8 KBytes, namely 2^{13} Bytes, which will take 13 bits of virtual address space. Now, 33 bits are left.

Then if every page table is required to fit into a single page with each size of 4 Bytes, there should at most be 2^{11} entry in a page table, which will take at most 11 bits of virtual address space.

So, there should be at least 3 levels of page tables with each occupy at most 11 bits of virtual address space.

Finally, the format of a virtual address should be:

- bit 0~12 for offset
- bit 13~23 for L3 index
- bit 24~34 for L2 index
- bit 35~45 for L1 index

4. Problem 4

(a) What is the page size? What is the maximum page table size?

page size: $2^{12}\ Bytes = 4\ KBytes.$

maximum page table size: $2^{20} \times 4\ Bytes = 4\ MBytes$

(b) Suppose it uses 2-level page table

- virtual address 0xC302C302

1-st level page number: 1100 0011 00(2) = 780(10)

offset: 0011 0000 0010(2) = 770(10)

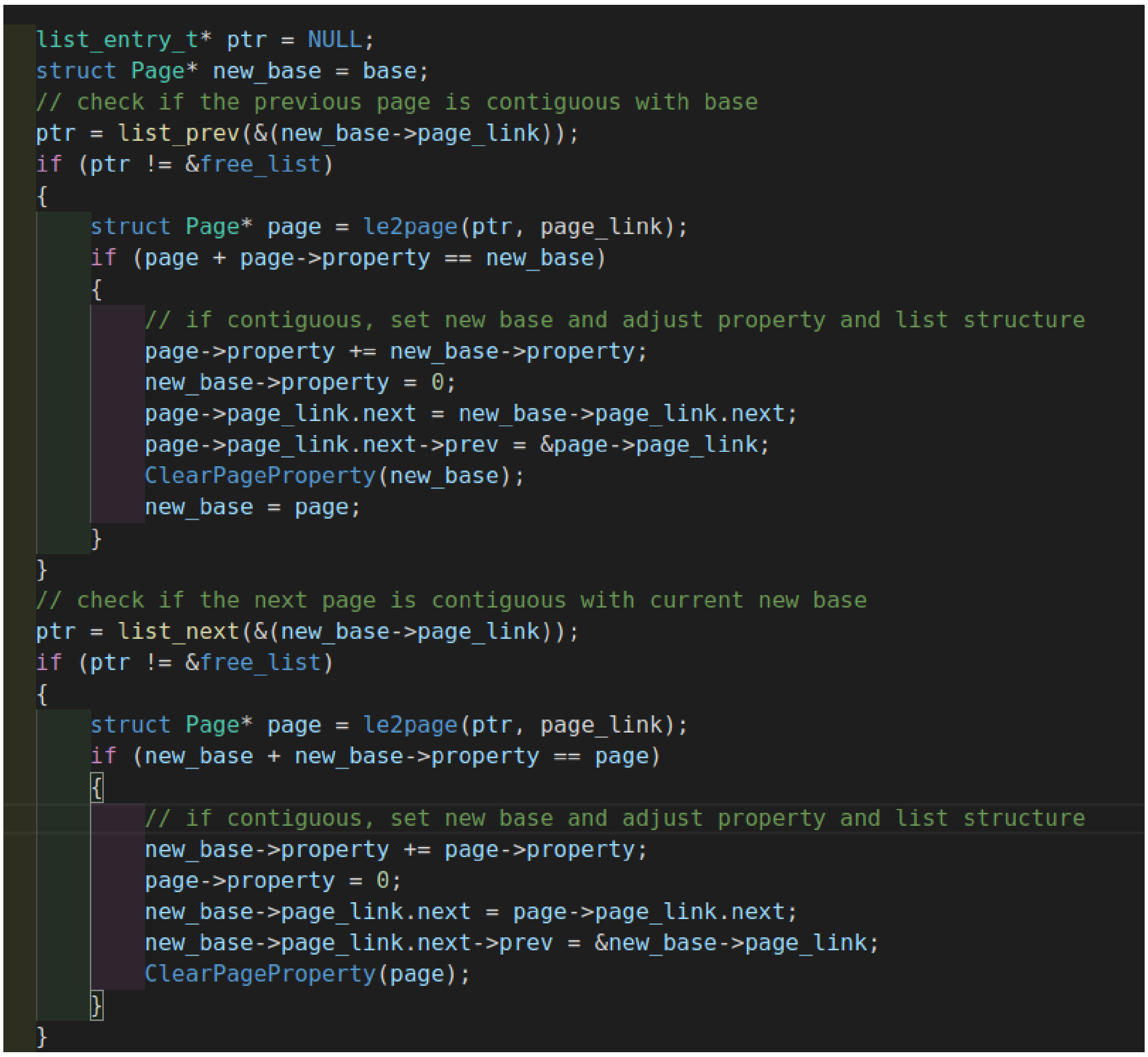
- virtual address 0xEC6666AB

2-nd level page number: 10 0110 0110(2) = 614(10)

offset: 0110 1010 1011(2) = 1707(10)

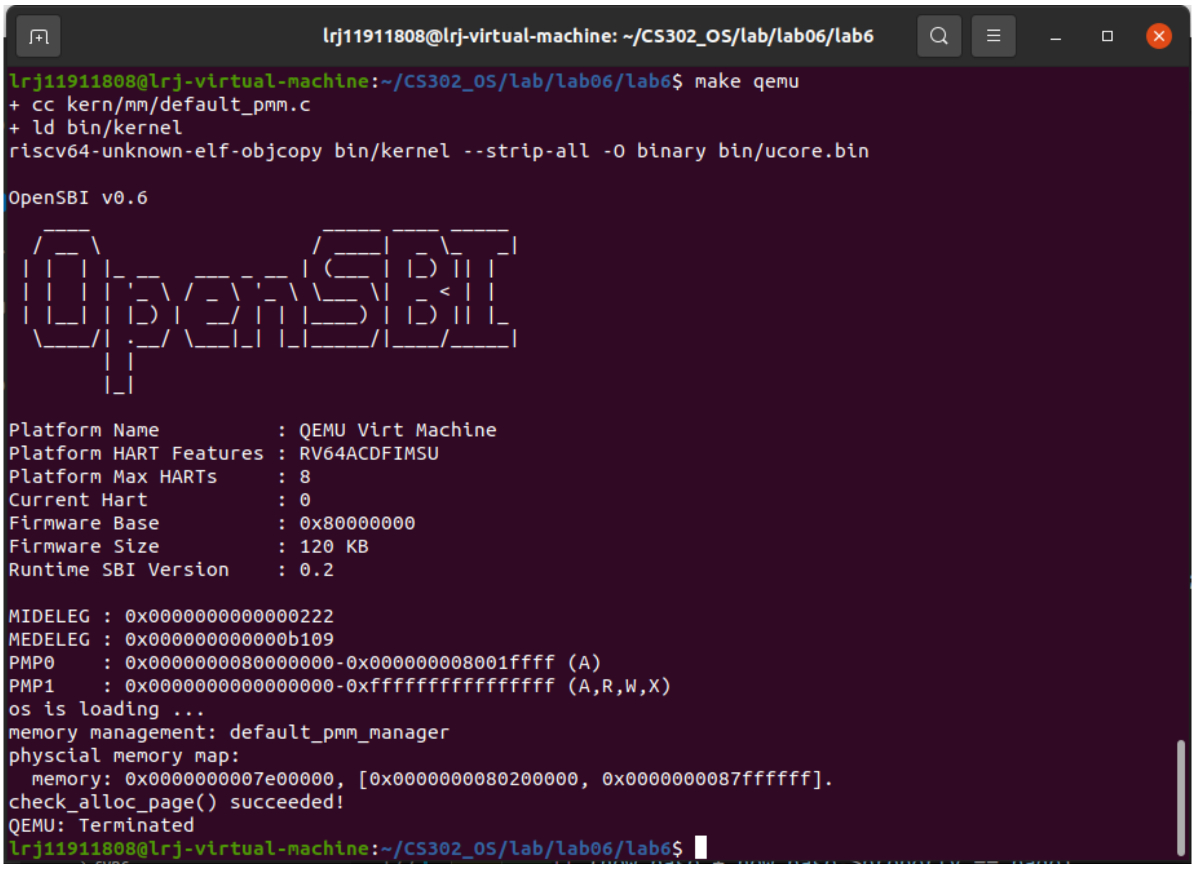
5. Please realize merging free blocks in default_free_pages()

Codes:



```c
list_entry_t* ptr = NULL;
struct Page* new_base = base;
// check if the previous page is contiguous with base
ptr = list_prev(&(new_base->page_link));
if (ptr != &free_list)
{
    struct Page* page = le2page(ptr, page_link);
    if (page + page->property == new_base)
    {
        // if contiguous, set new base and adjust property and list structure
        page->property += new_base->property;
        new_base->property = 0;
        page->page_link.next = new_base->page_link.next;
        page->page_link.next->prev = &page->page_link;
        ClearPageProperty(new_base);
        new_base = page;
    }
}
// check if the next page is contiguous with current new base
ptr = list_next(&(new_base->page_link));
if (ptr != &free_list)
{
    struct Page* page = le2page(ptr, page_link);
    if (new_base + new_base->property == page)
    {
        // if contiguous, set new base and adjust property and list structure
        new_base->property += page->property;
        page->property = 0;
        new_base->page_link.next = page->page_link.next;
        new_base->page_link.next->prev = &new_base->page_link;
        ClearPageProperty(page);
    }
}
```

Result Display:



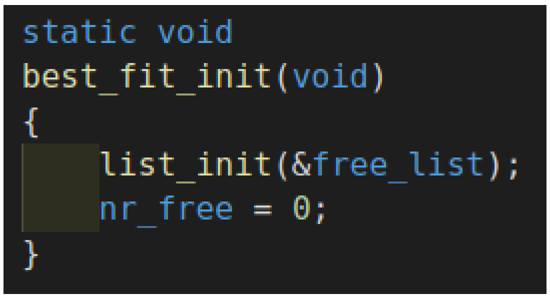
lrj11911808@lrj-virtual-machine: ~/CS302_OS/lab/lab06/lab6

```
lrj11911808@lrj-virtual-machine:~/CS302_OS/lab/lab06/lab6$ make qemu
+ cc kern/mm/default_pmm.c
+ ld bin/kernel
riscv64-unknown-elf-objcopy bin/kernel --strip-all -O binary bin/ucore.bin

OpenSBI v0.6
   ____                    _____ ____ _____
  / __ \                  / ____|  _ \_   _|
 | |  | |_ __   ___ _ __ | (___ | |_) || |
 | |  | | '_ \ / _ \ '_ \ \___ \|  _ < | |
 | |__| | |_) |  __/ | | |____) | |_) || |_
  \____/| .__/ \___|_| |_|_____/|____/_____|
        | |
        |_|

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!
QEMU: Terminated
lrj11911808@lrj-virtual-machine:~/CS302_OS/lab/lab06/lab6$ 
```

6. Realize bestfit in best_fit_pmm.c

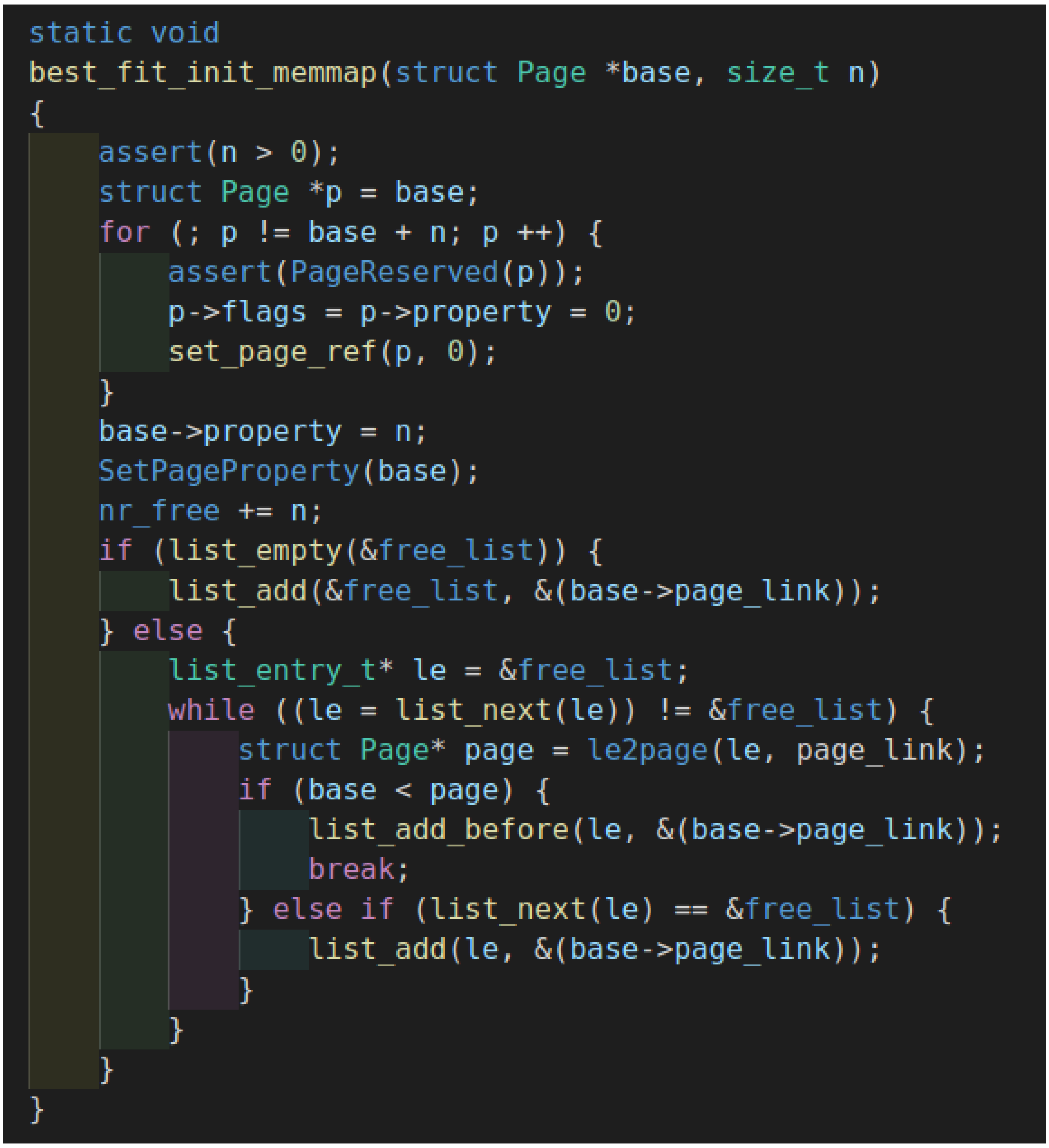
Codes:

- init



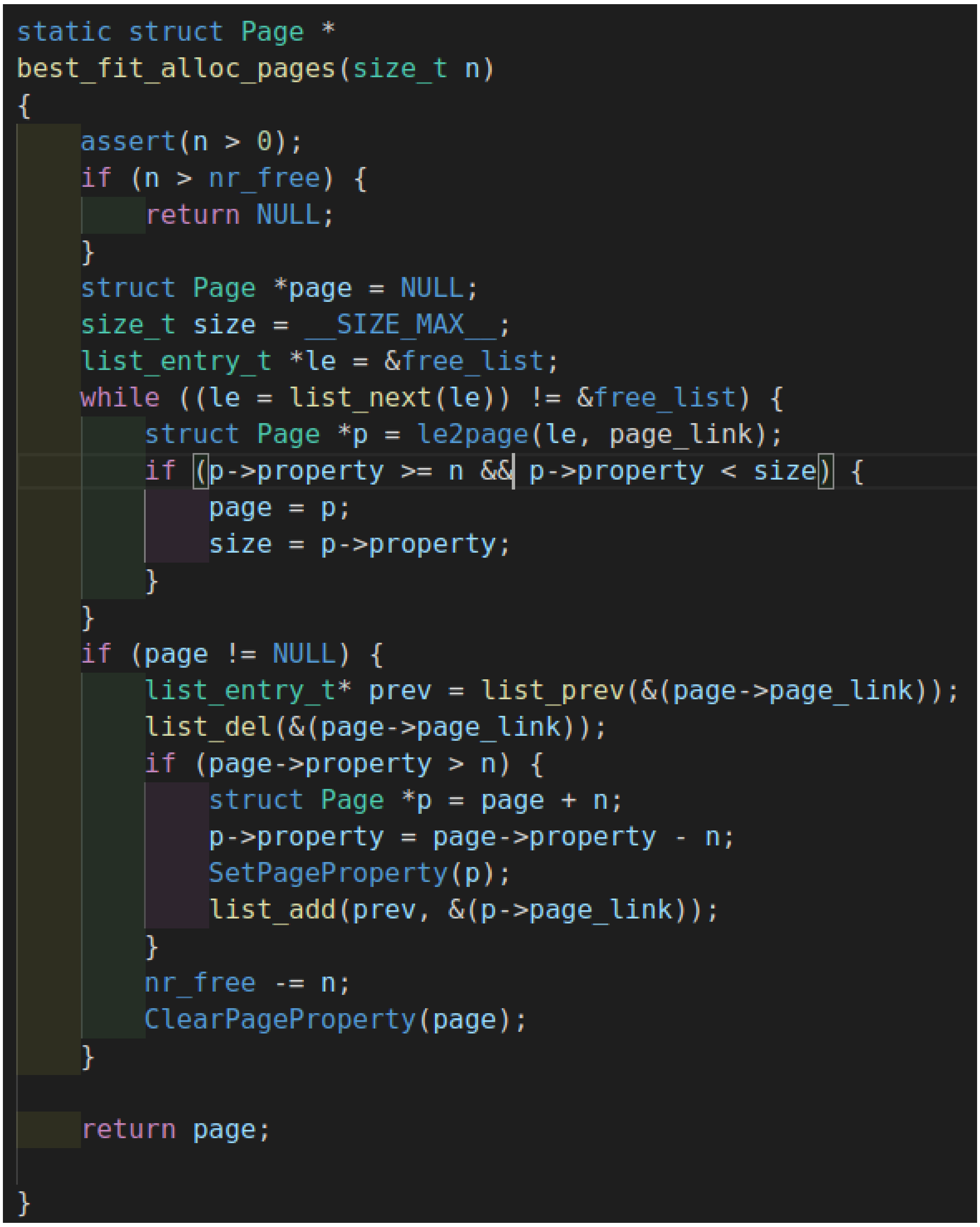
```
static void
best_fit_init(void)
{
    list_init(&free_list);
    nr_free = 0;
}
```

- init_memmap



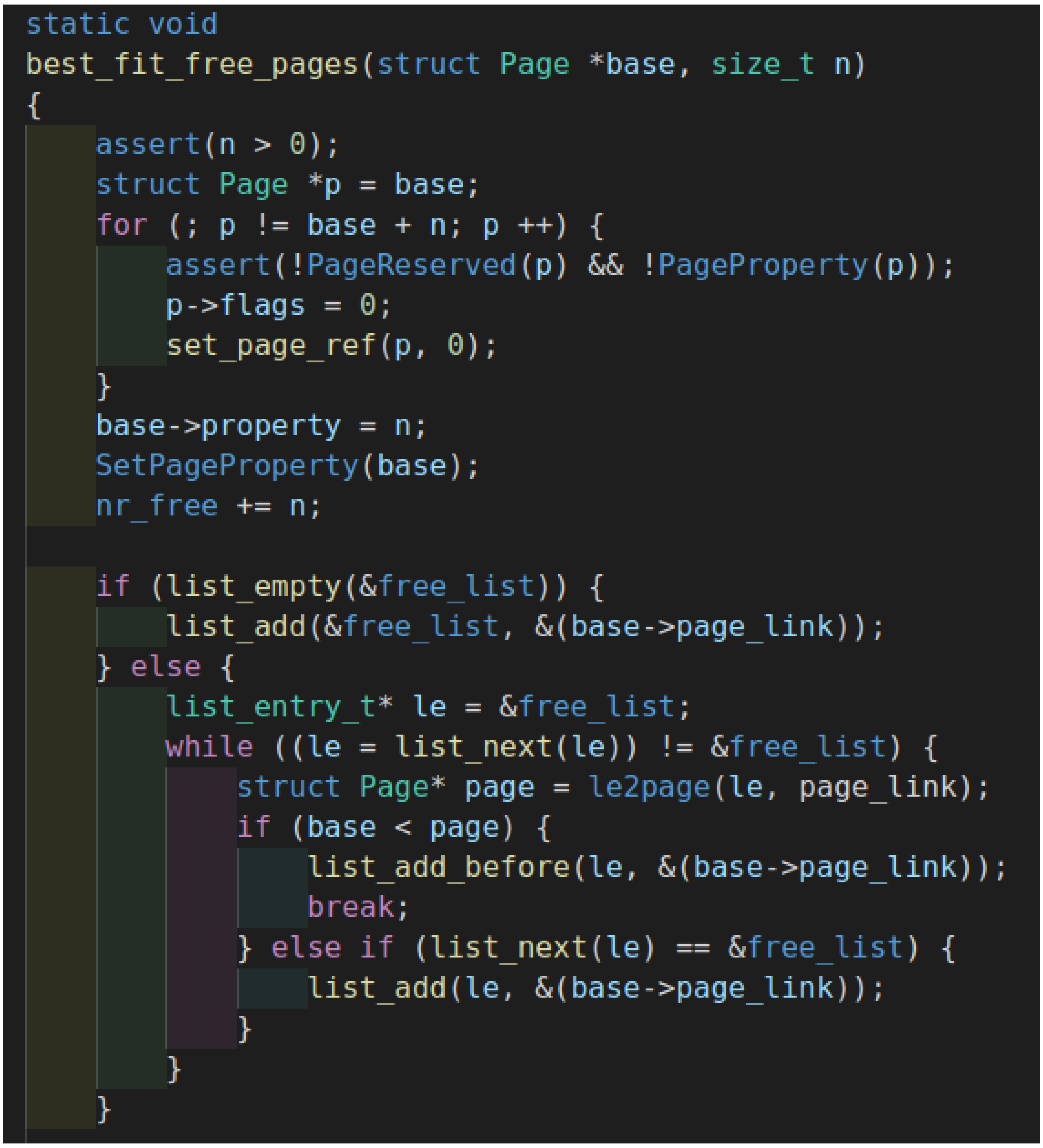
```
static void
best_fit_init_memmap(struct Page *base, size_t n)
{
    assert(n > 0);
    struct Page *p = base;
    for (; p != base + n; p ++) {
        assert(PageReserved(p));
        p->flags = p->property = 0;
        set_page_ref(p, 0);
    }
    base->property = n;
    SetPageProperty(base);
    nr_free += n;
    if (list_empty(&free_list)) {
        list_add(&free_list, &(base->page_link));
    } else {
        list_entry_t* le = &free_list;
        while ((le = list_next(le)) != &free_list) {
            struct Page* page = le2page(le, page_link);
            if (base < page) {
                list_add_before(le, &(base->page_link));
                break;
            } else if (list_next(le) == &free_list) {
                list_add(le, &(base->page_link));
            }
        }
    }
}
```

- alloc



```c
static struct Page *
best_fit_alloc_pages(size_t n)
{
    assert(n > 0);
    if (n > nr_free) {
        return NULL;
    }
    struct Page *page = NULL;
    size_t size = __SIZE_MAX__;
    list_entry_t *le = &free_list;
    while ((le = list_next(le)) != &free_list) {
        struct Page *p = le2page(le, page_link);
        if (p->property >= n && p->property < size) {
            page = p;
            size = p->property;
        }
    }
    if (page != NULL) {
        list_entry_t* prev = list_prev(&(page->page_link));
        list_del(&(page->page_link));
        if (page->property > n) {
            struct Page *p = page + n;
            p->property = page->property - n;
            SetPageProperty(p);
            list_add(prev, &(p->page_link));
        }
        nr_free -= n;
        ClearPageProperty(page);
    }

    return page;

}
```

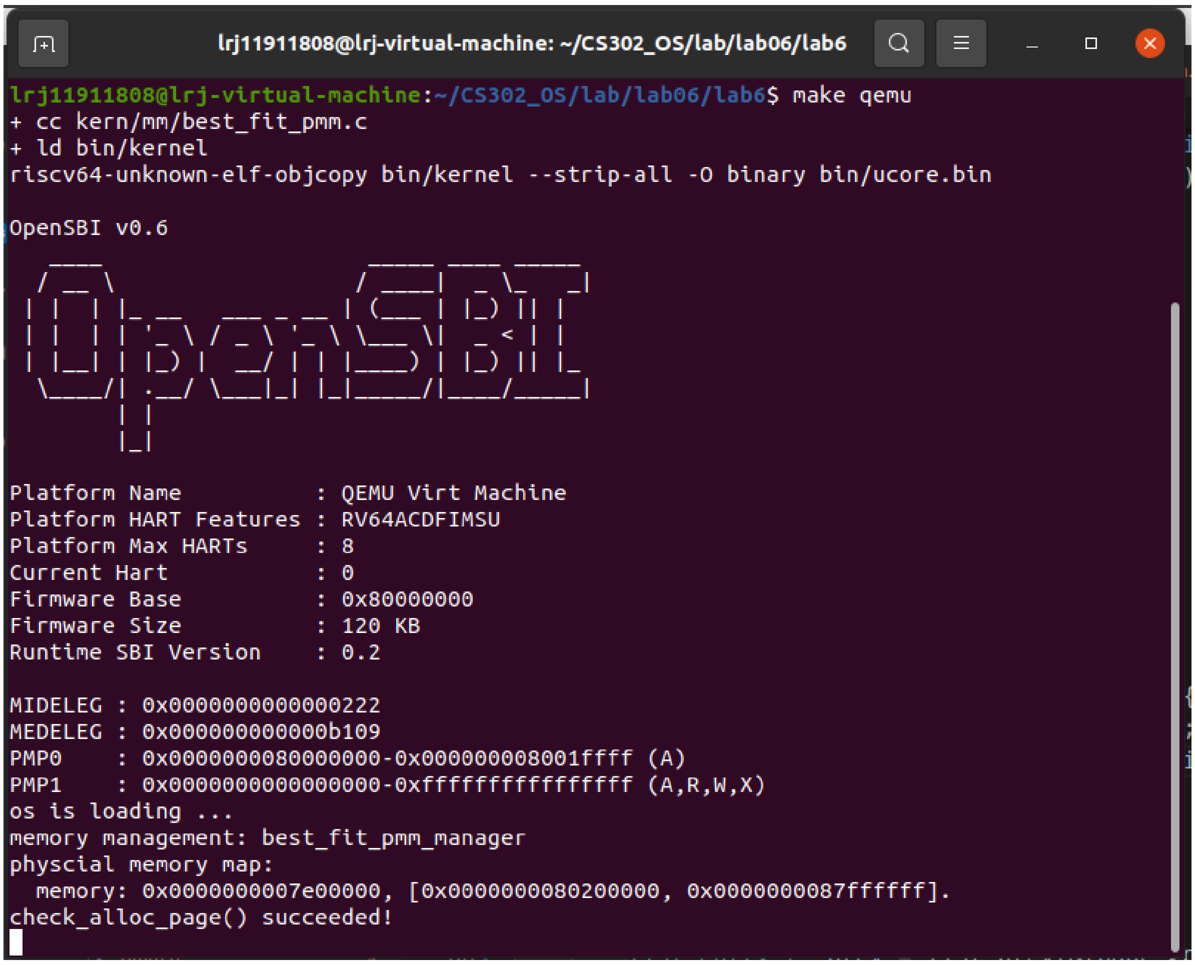
- free



```c
static void
best_fit_free_pages(struct Page *base, size_t n)
{
    assert(n > 0);
    struct Page *p = base;
    for (; p != base + n; p ++) {
        assert(!PageReserved(p) && !PageProperty(p));
        p->flags = 0;
        set_page_ref(p, 0);
    }
    base->property = n;
    SetPageProperty(base);
    nr_free += n;

    if (list_empty(&free_list)) {
        list_add(&free_list, &(base->page_link));
    } else {
        list_entry_t* le = &free_list;
        while ((le = list_next(le)) != &free_list) {
            struct Page* page = le2page(le, page_link);
            if (base < page) {
                list_add_before(le, &(base->page_link));
                break;
            } else if (list_next(le) == &free_list) {
                list_add(le, &(base->page_link));
            }
        }
    }
```

```c
list_entry_t* ptr = NULL;
struct Page* new_base = base;
// check if the previous page is contiguous with base
ptr = list_prev(&(new_base->page_link));
if (ptr != &free_list)
{
    struct Page* page = le2page(ptr, page_link);
    if (page + page->property == new_base)
    {
        // if contiguous, set new base and adjust property and list structure
        page->property += new_base->property;
        new_base->property = 0;
        page->page_link.next = new_base->page_link.next;
        page->page_link.next->prev = &page->page_link;
        ClearPageProperty(new_base);
        new_base = page;
    }
}
// check if the next page is contiguous with current new base
ptr = list_next(&(new_base->page_link));
if (ptr != &free_list)
{
    struct Page* page = le2page(ptr, page_link);
    if (new_base + new_base->property == page)
    {
        // if contiguous, set new base and adjust property and list structure
        new_base->property += page->property;
        page->property = 0;
        new_base->page_link.next = page->page_link.next;
        new_base->page_link.next->prev = &new_base->page_link;
        ClearPageProperty(page);
    }
}
```

Result Display:



```
lrj11911808@lrj-virtual-machine:~/CS302_OS/lab/lab06/lab6$ make qemu
+ cc kern/mm/best_fit_pmm.c
+ ld bin/kernel
riscv64-unknown-elf-objcopy bin/kernel --strip-all -O binary bin/ucore.bin

OpenSBI v0.6
   ____                    _____ ____ _____
  / __ \                  / ____|  _ \_   _|
 | |  | |_ __   ___ _ __ | (___ | |_) || |
 | |  | | '_ \ / _ \ '_ \ \___ \|  _ < | |
 | |__| | |_) |  __/ | | |____) | |_) || |_
  \____/| .__/ \___|_| |_|_____/|____/_____|
        | |
        |_|

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!
```