

MEMORY MANAGEMENT

RETRIEVING MEMORY MAP:

Code from loader file

GetMemInfoStart:

```
mov eax,0xe820
mov edx,0x534d4150
mov ecx,20
mov dword[0x9000],0
mov edi,0x9008
xor ebx,ebx
int 0x15
jc NotSupport
```

GetMemInfo:

```
add edi,20
inc dword[0x9000]
test ebx,ebx
jz GetMemDone
```

The free memory is normally above 1mb. Our kernel is running at 2mb.

But not all the memory above 1mb is available to use. We need to find out the memory map. So in the loader file, we have retrieved the information about memory map and stored it in the address 9000, We didn't print the information on the screen since we hadn't implemented print function back then.

And now we can collect the data and print them on the screen.

Here we make changes. The data we retrieved is stored in the structures. So we stores the count of structures in address 9000 and We initialize 9000 with value 0 and this value is 4-byte data and the structures are stored from address 0x9008. So we move edi 0x9008 each time we get the memory block, we update the count value. So we increment the value in memory address 0x9000

Then we test ebx, if ebx is 0, it means that we reach the end and we jump to get memory done So we use jz instruction, if zero flag is set, we jump to get memory done. if carry flag is not set after executing int instruction, we will jump to get mem info to continue retrieving the memory info, otherwise we reach the end and exit out the loop. So here we use jnc instruction if carry flag is not set,we jump to get memory information

Code from memory.h file

```
#ifndef _MEMORY_H_
#define _MEMORY_H_

#include "stdint.h"

struct E820 {
    uint64_t address;
    uint64_t length;
    uint32_t type;
} __attribute__((packed));

struct FreeMemRegion {
    uint64_t address;
    uint64_t length;
};

void init_memory(void);

#endif
```

The first structure we define is the structure that we have seen in the loader file. when we retrieve the memory info using the BIOS service the data is actually arranged in the structures. The first field in the structure is the base address of the memory region. The second field is the length of the memory region and the last one is the memory type. Here we name the structure E820. Notice that we add attribute packed, so the structure is stored without padding in it. Otherwise the structure will be larger than 20 bytes and we will not interpret the data in memory correctly, the next structure is called free memory region. Because some of memory regions are not available to use. So this structure is used to store the information about the free memory. We have only two fields in it. The base address and the length of the memory region.

Code from memory.c

```

#include "memory.h"
#include "print.h"
#include "debug.h"

static struct FreeMemRegion free_mem_region[50];

void init_memory(void)
{
    int32_t count = *(int32_t*)0x9000;
    uint64_t total_mem = 0;
    struct E820 *mem_map = (struct E820*)0x9008;
    int free_region_count = 0;

    ASSERT(count <= 50);

    for(int32_t i = 0; i < count; i++) {
        if(mem_map[i].type == 1) {
            free_mem_region[free_region_count].address = mem_map[i].address;
            free_mem_region[free_region_count].length = mem_map[i].length;
            total_mem += mem_map[i].length;
            free_region_count++;
        }

        printk("%x  %uKB  %u\n", mem_map[i].address, mem_map[i].length/1024, (uint64_t)mem_map[i].type);
    }

    printk("Total memory is %uMB\n", total_mem/1024/1024);
}

```

EXPLANATION OF CODE:

The free memory region is used to store the information for later use. In this case we assume that we get no more than 50 blocks of memory regions. In the function, variable memory map holds the address of the data retrieved by the BIOS services. The base address is at 0x9008. The number of the memory regions is stored at 0x9000. Each memory region is stored in an e820 structure we just defined. So the variable count indicates how many memory regions we have in the system. The variable count is 4 bytes, here we used fixed width data type. Variable free memory region count is used to store the actual number of free memory regions. Total memory stores the size of free memory we can use in the system. Also, for simplicity, we use assert to make the assumption that the number of memory regions is less than 50. Then we can parse the e820 structures.

First off, we loop through each of the memory structures and print the base address of memory region, the size and type of the memory region. The free memory region we can use is the memory region with type 1. If the type is 1, we copy the data to the free region structure. So the variable free region is actually an array of free memory region structure and variable total memory stores the memory size the operating system can use. So we added the length to the total memory to update the free memory size. Then we increment the free region count after we exit out of the loop, we simply print memory size, here we convert it to the value in megabytes.

OUTPUT:

```

0H 636KB 1
9F000H 4KB 2
E8000H 96KB 2
100000H 1047488KB 1
3FFF0000H 64KB 3
FFFC0000H 256KB 2
Total memory is 1023MB

```

Explanation of output :

The first entry are the free memory we can use in the system The fourth entry shows that free memory starts 1mb and the size of this region is more than 1000mb .The other entries shows the reserved memory and the operating system doesn't use these memory regions. The total memory is 1023 mb that is the size of free memory

CODE OF SETTING UP PAGGING

```

cld
mov edi,0x70000
xor eax,eax
mov ecx,0x10000/4
rep stosd

mov dword[0x70000],0x71003
mov dword[0x71000],10000011b

```

Here first we zero the 10000 bytes of memory region starting 70000 and the value of cr3 is 70000, so the address of the pagemap is 70000 . so here we set up the first entry of the table it takes up 4k space because it includes 512 entries each with 8 bytes .so next table address is set to 71000 and the lower 3 bits are the attribute we need to set. We want the memory to be readable and writeable and accessed by only kernel so in (011)UWP we set p to 1 which means present. So the value is 3 the value we assign to the entry is 71003 , and here in 10000011b the first one means 1g physical page translation

```

mov dword[0x70000],0x71003
mov dword[0x71000],10000011b

mov eax,(0xffff800000000000 >> 39)
and eax,0x1ff // this is for clearing other bits
mov dword[0x70000 + eax*8],0x72003
mov dword[0x72000],10000011b

```

Here in the above code we set the 1g physical page to the same physical page address where the kernel locates

```

[BITS 64]
LMEntry:
    mov rsp,0x7c00

    cld
    mov rdi,0x200000
    mov rsi,0x10000
    mov rcx,51200/8
    rep movsq

    mov rax,0xffff800000200000
    jmp rax

```

Here , since the kernel is reallocated to the new virtual address which is far away from the loader we need to save it to the 64 bit register and jump to kernel ,so we move the virtual address to rax and then jmp rax

In kernel file

```
start:
    mov rax,Gdt64Ptr
    lgdt [rax]
```

```
SetTss:
    mov rax,Tss
    mov rdi,TssDesc
    mov [rdi+2],ax
    shr rax,16
    mov [rdi+4],al
    shr rax,8
    mov [rdi+7],al
    shr rax,8
    mov [rdi+8],eax
    mov ax,0x20
    ltr ax
```

Here the first thing we are going to do is change the load gdt instruction. Since the kernel is running at the high memory location. The memory address of gdt pointer is also at this memory area. We cannot reference it using 32-bit address.

Instead, we move the address in register rax and load rax. So we copy address of gdt pointer to rax and load rax, the same thing happens with the set tss. We move the address of tss descriptor to rdi and access the memory locations using rdi instead.

```
    mov rax,KernelEntry
    push 8
    push rax
    db 0x48
    retf
```

```
KernelEntry:
    mov rsp,0xfffff8000000200000
    call KMain
```

In the above code we move the address of kernel entry to rax then push rax. Next one is stack address. The stack address copied to the rsp register is at the low memory location. We change it to the high memory address which points to the same physical address. In the linker script we specify the linker to link the kernel at the desired address. To do it, we use `__TEXT` which means the current address. And we assign the base address to it. In the loader file, we jump to the new virtual address.

OUTPUT:

```
0H 636KB 1
9F000H 4KB 2
E8000H 96KB 2
100000H 1047488KB 1
3FFF0000H 64KB 3
FFFC0000H 256KB 2
Total memory is 1023MB
```

Code for memory allocation :

```
#define PAGE_SIZE (2*1024*1024)
#define PA_UP(v) (((uint64_t)v+PAGE_SIZE-1)>>21)<<21)
#define PA_DOWN(v) (((uint64_t)v)>>21)<<21)
#define P2V(p) ((uint64_t)(p)+0xffff800000000000)
#define V2P(v) ((uint64_t)(v)-0xffff800000000000)
```

The macro page size which is set 2mb The next two macros align the address to 2m boundary. The page align up will align the address to the next 2m boundary if it is not aligned. The page align down will align the address to the previous 2m boundary

```
for (int i = 0; i < free_region_count; i++) {
    uint64_t vstart = P2V(free_mem_region[i].address);
    uint64_t vend = vstart + free_mem_region[i].length;

    if (vstart > (uint64_t)&end) {
        free_region(vstart, vend);
    }
    else if (vend > (uint64_t)&end) {
        free_region((uint64_t)&end, vend);
    }
}
```

Here we define two variables vstart and vend which represent the beginning and end of the memory region. You can see we use p2v to convert the physical address to virtual address and assign it to the vstart. Then we set vend by adding the length of region to vstart. So vstart and vend hold the virtual addresses instead of physical

address Now we are going to check the location of the memory region .If the start of memory region is larger than the end of kernel, this is the free memory we want and we do the initialization by calling function free region.

NOTE: here we need the address of end because the end is a symbol instead of a variable

If the start address of the memory region is less than the end of kernel we will compare the end of the region with the end of the kernel. If it is larger than the end of kernel, we will initialize the memory from the end of the kernel to the end of the region. Otherwise we ignore the region because it is within the kernel.

```
static void free_region(uint64_t v, uint64_t e)
{
    for (uint64_t start = PA_UP(v); start+PAGE_SIZE <= e; start += PAGE_SIZE) {
        if (start+PAGE_SIZE <= 0xffff800040000000) {
            kfree(start);
        }
    }
}
```

This function is simple, all it does is just divide the region into 2mb pages and call the function kfree to collect the pages. So in the for loop, we start off by aligning the address to the next 2m boundary. Then we compare this page with the end of the region. If it is within the region,we call the free function

NOTE: here we also add another check this value is 1g above the base of kernel.so what it does is just check to see if the page we are about to initialize is beyond the first 1g of ram

```
void kfree(uint64_t v)
{
    ASSERT(v % PAGE_SIZE == 0);
    ASSERT(v >= (uint64_t)&end);
    ASSERT(v+PAGE_SIZE <= 0xffff800040000000);

    struct Page *page_address = (struct Page*)v;
    page_address->next = free_memory.next;
    free_memory.next = page_address;
}
```

This is the function where we do the initialization.

Before we start, we add these simple checks, the first one is to make sure that the virtual address is page aligned.The next one assumes that the virtual address is not within the kernel and the last one checks 1g memory limits

NOW here the bunch of pages are stored in the form in linked list in reverse order In the function,we first convert the virtual address to the pointer to type structure page next we copy the head of the list to the first 8 bytes of the page and then save the virtual address to free memory. At this point, the head of the linked list points to the current page.

```
void* kalloc(void)
{
    struct Page *page_address = free_memory.next;

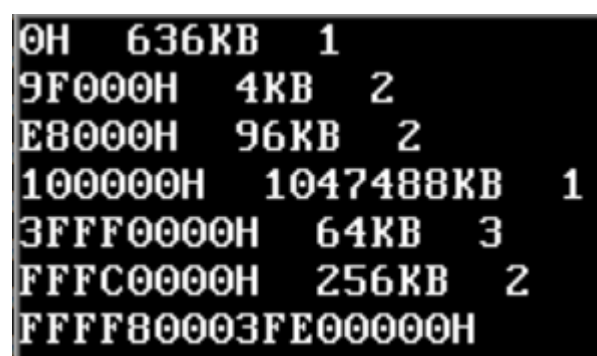
    if (page_address != NULL) {
        ASSERT((uint64_t)page_address % PAGE_SIZE == 0);
        ASSERT((uint64_t)page_address >= (uint64_t)&end);
        ASSERT((uint64_t)page_address+PAGE_SIZE <= 0xffff800040000000);

        free_memory.next = page_address->next;
    }

    return page_address;
}
```

kalloc used in the kernel mode allocation memory is just removing a page form the page list and return to the caller.So we copy the first page in the list to page address and check to see if it is NULL. If the page is not NULL this is a valid page and we will return the page to the caller. As we did in the function kfree, we add the same checks before we allocate the page and then make the head of the list point to the next page.Now, we have removed the page from the list.Return the page

OUTPUT:



```
0H 636KB 1
9F000H 4KB 2
E8000H 96KB 2
100000H 1047488KB 1
3FFF0000H 64KB 3
FFFC0000H 256KB 2
FFFF80003FE00000H
```

EXPLANATION OF THE OUTPUT :

The free memory in the fourth entry is starting from 1mb.we add it to the length of the region which produce the value 3fff0000 and we add the base of the kernel. The result is ffff80003fff0000.

As you can see, the end of the free memory the system collect is different.

Remember the free memory is stored as a list of 2m pages. The memory address we get is not 2m aligned. If we align it to the previous 2m boundary, we will find that we get the exact value showing on the screen and also here we cannot align the address to the next 2m boundary, if we do it, we will include the reserved memory.

MEMORY PAGES:

C CODE:

```
#include "memory.h"

#include "print.h"

#include "debug.h"

#include "lib.h"

#include "stddef.h"

#include "stdbool.h"


static void free_region(uint64_t v, uint64_t e);


static struct FreeMemRegion free_mem_region[50];

static struct Page free_memory;

static uint64_t memory_end;

uint64_t page_map;

extern char end;


void init_memory(void)

{
```

```

int32_t count = *(int32_t*)0x9000;

uint64_t total_mem = 0;

struct E820 *mem_map = (struct E820*)0x9008;

int free_region_count = 0;


ASSERT(count <= 50);


                                for(int32_t i = 0; i < count; i++) {

if(mem_map[i].type == 1) {

    free_mem_region[free_region_count].address = mem_map[i].address;

    free_mem_region[free_region_count].length = mem_map[i].length;

    total_mem += mem_map[i].length;

    free_region_count++;

}

    printk("%x %uKB
%u\n",mem_map[i].address,mem_map[i].length/1024,(uint64_t)mem_map[i].type);

                                }


for (int i = 0; i < free_region_count; i++) {

    uint64_t vstart = P2V(free_mem_region[i].address);

    uint64_t vend = vstart + free_mem_region[i].length;

    if (vstart > (uint64_t)&end) {

        free_region(vstart, vend);

```

```

    }

    else if (vend > (uint64_t)&end) {

        free_region((uint64_t)&end, vend);

    }

}

memory_end = (uint64_t)free_memory.next+PAGE_SIZE;

printk("%x\n",memory_end);

}

static void free_region(uint64_t v, uint64_t e)

{

    for (uint64_t start = PA_UP(v); start+PAGE_SIZE <= e; start += PAGE_SIZE) {

        if (start+PAGE_SIZE <= 0xffff800040000000) {

            kfree(start);

        }

    }

}

void kfree(uint64_t v)

{

    ASSERT(v % PAGE_SIZE == 0);

    ASSERT(v >= (uint64_t) & end);

```

```
ASSERT(v+PAGE_SIZE <= 0xffff800040000000);
```

```
struct Page *page_address = (struct Page*)v;
```

```
page_address->next = free_memory.next;
```

```
free_memory.next = page_address;
```

```
}
```

```
void* kalloc(void)
```

```
{
```

```
struct Page *page_address = free_memory.next;
```

```
if (page_address != NULL) {
```

```
    ASSERT((uint64_t)page_address % PAGE_SIZE == 0);
```

```
    ASSERT((uint64_t)page_address >= (uint64_t)&end);
```

```
    ASSERT((uint64_t)page_address+PAGE_SIZE <= 0xffff800040000000);
```

```
    free_memory.next = page_address->next;
```

```
}
```

```
return page_address;
```

```
}
```

```
static PDPTR find_pml4t_entry(uint64_t map, uint64_t v, int alloc, uint32_t attribute)
```

```

{

    PDPTR *map_entry = (PDPTR*)map;

    PDPTR pdptr = NULL;

    unsigned int index = (v >> 39) & 0x1FF;

    if ((uint64_t)map_entry[index] & PTE_P) {

        pdptr = (PDPTR)P2V(PDE_ADDR(map_entry[index]));

    }

    else if (alloc == 1) {

        pdptr = (PDPTR)kalloc();

        if (pdptr != NULL) {

            memset(pdptr, 0, PAGE_SIZE);

            map_entry[index] = (PDPTR)(V2P(pdptr) | attribute);

        }

    }

    return pdptr;

}

```

```

static PD find_pdpt_entry(uint64_t map, uint64_t v, int alloc, uint32_t attribute)

{

    PDPTR pdptr = NULL;

    PD pd = NULL;

```

```
unsigned int index = (v >> 30) & 0x1FF;
```

```
pdptr = find_pml4t_entry(map, v, alloc, attribute);
```

```
if (pdptr == NULL)
```

```
    return NULL;
```

```
if ((uint64_t)pdptr[index] & PTE_P) {
```

```
    pd = (PD)P2V(PDE_ADDR(pdptr[index]));
```

```
}
```

```
else if (alloc == 1) {
```

```
    pd = (PD)kalloc();
```

```
    if (pd != NULL) {
```

```
        memset(pd, 0, PAGE_SIZE);
```

```
        pdptr[index] = (PD)(V2P(pd) | attribute);
```

```
    }
```

```
}
```

```
return pd;
```

```
}
```

```
bool map_pages(uint64_t map, uint64_t v, uint64_t e, uint64_t pa, uint32_t attribute)
```

```
{
```

```
    uint64_t vstart = PA_DOWN(v);
```

```
uint64_t vend = PA_UP(e);

PD pd = NULL;

unsigned int index;


ASSERT(v < e);

ASSERT(pa % PAGE_SIZE == 0);

ASSERT(pa+vend-vstart <= 1024*1024*1024);


do {

    pd = find_pdpt_entry(map, vstart, 1, attribute);

    if (pd == NULL) {

        return false;

    }


    index = (vstart >> 21) & 0x1FF;

    ASSERT((((uint64_t)pd[index] & PTE_P) == 0));


    pd[index] = (PDE)(pa | attribute | PTE_ENTRY);


    vstart += PAGE_SIZE;

    pa += PAGE_SIZE;

} while (vstart + PAGE_SIZE <= vend);
```



```
    return true;
}

void switch_vm(uint64_t map)
{
    load_cr3(V2P(map));
}

static void setup_kvm(void)
{
    page_map = (uint64_t)kalloc();

    ASSERT(page_map != 0);

    memset((void*)page_map, 0, PAGE_SIZE);

    bool status = map_pages(page_map, KERNEL_BASE, memory_end,
V2P(KERNEL_BASE), PTE_P|PTE_W);

    ASSERT(status == true);
}

void init_kvm(void)
{
    setup_kvm();

    switch_vm(page_map);

    printk("memory manager is working now");
}
```

```
}
```

EXPLANATION OF THE ABOVE CODE:

Function initialize kvm which then call setup kvm. Because we have collected the physical memory when we call this function, we can allocate a new free memory page using function kalloc. This page is actually the new page map level 4 table. Since what we do here is initializing kernel vm, if the allocation failed we simply stop the system. So here we use assert to test the condition. If the check pass, we zero the page using memset function because the page could include random values when they get the page from function kalloc. The function map pages take a few parameters. The first one is pml4 table. Next two parameters are the start and end address of memory region we want to map. The arguments we pass in this example are the base of the kernel and the end of free memory, both of which are virtual addresses. The next one is the start of physical page we want to map into. Because we want to map the kernel to the same physical address, we pass physical address of kernel base in this case. The last one holds the attribute of the page table or physical page. If the mapping failed, it returns false

Here we simply add assert to test the conditions because we used Boolean type, we add the header file stdbool. These are macros we defined in the header file. The p, w, u and entry are the attributes of the table entries. They are in decimal or hexadecimal formats. If we convert them to binary forms, we can see they just set the corresponding bits in the table entries. The pde address and pte address are used to retrieve the address of the next level page or physical page. The entries in page map level 4 table and page directory pointer table have attributes within the lower 12 bits. So we clear the lower 12 bits of the entry to get the correct address. The page table entry has attributes within the 21 bits. So we clear 21 bits of the address to get the physical address. These three customized data types are used when finding the translation tables. The pdptr points to page directory and page directory points to page directory entry, etc. So here we specify that the kernel memory is readable, writeable and not accessible by the user applications.

Let's move to map pages function. In the function, we define two variables vstart and vend which save the aligned virtual addresses.

For example, if we get unaligned start virtual address, we need to map the page where the start address is located. So here we align the address to the previous 2m page so that we can include the start address. The same thing happens with end address. In this case, we align the address to the next 2m page to include the page where the end address is located. Then we define variable pd which is used to set the page directory entry. The index is used to locate the specific entry in the table. The first one checks the start and end of the region. The next one is to make sure

the physical address we want to map into is page aligned. The last one checks to see if the end of physical address is outside the range of 1g memory

Now we do mapping here the first thing we are going to do is we are going to find the page directory pointer table entry which points to page directory table by calling function find pdpt entry. We will see this function in a minute. If it returns null, it means that it failed and we simply return false. If it returns a valid table, we use the index to locate the correct page entry according to the virtual address.

We have learned that the virtual address is divided into several parts to locate the different table entries. Here the table entry we want is page directory entry, the index value is 9 bits in total starting from bits 21. So we shift right 21 bits and clear other bits except the lower 9 bits of the result. The value in the index is used to find the entry in the page directory table.

Here we check the present bit of the entry. If the present bit is set, it means that we remap to the used page. We don't allow it to happen in our system. So we use assert to check that. If the check passes, what we are going to do next is we are going to set the entry with the physical address and attributes and we should not forget to add the attribute page entry to indicate that this is 2m page translation. Now we have mapped one page then we move to the next page by adding the page size to the virtual address and physical address. If the virtual address is still within the memory region, we continue the process until we reach the end of the region.

Now, let's see the function find pdpt entry. The parameters it takes are the page map level 4 table, virtual address and alloc indicating whether or not we will create a page if it does exist. The last one is attribute we can see we pass the value 1 to the function. So we will create a page if it does exist. In the function find pdpt entry, we define the variable pdptr, pd and index. Since we find the pdpt entry in this case, the index value is located from the bits 30 of the virtual address and is also 9 bits in total. So we preserve the low 9 bits of the result. Then we call function find pml4 table entry to get the page directory pointer table and we check if it returns null. If it returns a valid table, we will find the correct entry in the table using index value. If the present attribute is 1, then it means that the value in the entry points to the next level table which is page directory table. We use macro pde address to clear the attribute bits to get the address. Note that here we use p2v to convert the physical address to virtual address because the addresses in the table entries are physical addresses. If the present bit is cleared, then it's an unused entry. We will check the value of alloc. If it's equal to 1, we will allocate a new page and set the entry to make it point to this page. Also, we use v2p to convert the virtual address to physical address and save it to the entry. In the end we return the address of page directory table.

The function find pml4 table entry has the same parameters as find pdpt entry.

The pml4 table holds the entries which point to the page directory pointer tables. So we define the variable map entry which is a pointer to type pdptr and variable pdptr. Since we find the entry in pml4 table, the index is located from bits 39 of the virtual address then we locate the correct entry and check to see if the present bit is set. If it is set, we copy the address to pdpt. otherwise we will check the alloc. If it is 1, we will allocate a new page and set the entry to point to the page. This process is pretty much the same as in function find pdpt entry. At the end of the function, we return pdptR. The last function we implement is switch vm, we call switch vm to use the new paging setup in the kernel

OUTPUT:

```
0H 636KB 1
9F000H 4KB 2
E8000H 96KB 2
100000H 1047488KB 1
3FFF0000H 64KB 3
FFFC0000H 256KB 2
FFFF80003FE00000H
memory manager is working now
```