Introduction

Your OS will be managing all of the memory. As you have already probably figured out, we do not have the `sbrk` or `mmap` functions to ask for more memory. Your OS will instead be asked by the OS itself as well as user applications for dynamic memory.

Since your OS will be running in virtual memory, it is important to create a **page-grained memory allocator**. This allocator hands out individual **pages** of memory using virtual memory addresses. Recall that RISC-V has three page sizes: (1) 1GB, (2) 2MB, and (3) 4KB.

> Your page-grained allocator will hand out individual or contiguous 4KB pages.

Page Allocator

Edit a file called `page.h` . This file will export the page-grained allocator functions, listed below.

page.h

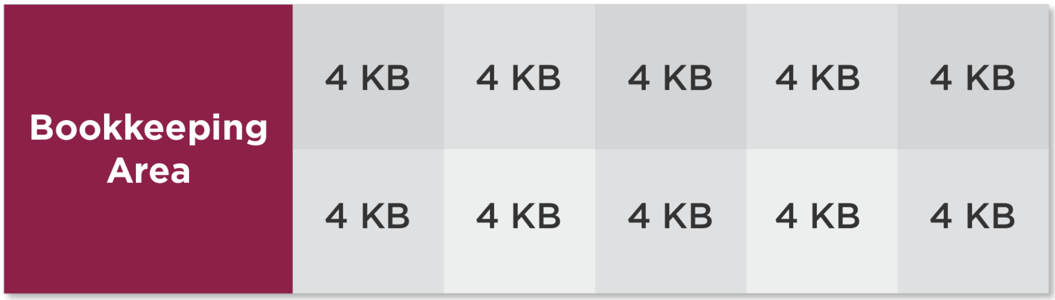
```c
void page_init(void);
void *page_nalloc(unsigned int n);
void *page_znalloc(unsigned int n);
void page_free(void *p);

#define page_alloc()  page_nalloc(1)
#define page_zalloc() page_znalloc(1)
```

Function	Description
`page_init`	Initializes the page allocator system. Usually, this means setting up the bookkeeping area.
`page_nalloc`	Allocate a contiguous set of pages given by the parameter `n`. This function does NOT clear the memory returned.
`page_znalloc`	Allocate a contiguous set of pages given by the parameter `n`. This function clears the memory returned to 0.
`page_free`	Returns the page(s) allocated and returned by `p` . Recall that the page address in `p` may be only a single page or multiple pages.

Write your page_* functions in a file called page.c. The Makefile will automatically compile all C files in src/ and expect all headers be in src/include/

Your page grained allocator will use the top portion of the memory pool (`_heap_start`) to store bookkeeping information, which is two bits: (1) page tak and (2) last page. Therefore, we bookkeep four pages per byte.

Memory Layout for Page-Grained Allocator



- `_heap_start + n` will be for pages $(4 \times n), (4 \times n + 1), (4 \times n + 2)$, and $(4 \times n + 3)$
- `_heap_start + 0` will be for pages 0, 1, 2, and 3.
- `_heap_start + 1` will be for pages 4, 5, 6, and 7.

Finding Things in the Heap

1. $\text{heap\_size} = \text{\_heap\_end} - \text{\_heap\_start}$.
   a. This gives us the total number of bytes in the heap.
2. $\text{num\_pages} = \frac{\text{heap\_size}}{4096}$.
   a. This is the total number of pages in the heap.
3. $\text{num\_pages} \div 4$
   a. Since we bookkeep four pages per byte, we divide by four.
4. $\text{bk\_size} = \frac{(\text{\_heap\_end} - \text{\_heap\_start})}{4096 \times 4}$.
   a. This calculates the number of bookkeeping bytes needed to manage the heap.

> The first page needs to start at a page boundary, so the first page starts at: $\text{page}_0 = (\text{\_heap\_start} + \text{ALIGN\_UP}(\text{bk\_size}, 4096))$.

> Do not forget to set the taken bits for ALL of the bookkeeping bytes. Otherwise, your allocator **may** allocate pages already taken for the book keeping bits.

Bookkeeping Bits

There are two bits per page: (1) taken and (2) last. Since we can allocate multiple, contiguous pages, we need to mark the last page we handed out in a sequence of pages.

Example

We can visualize how to dole out pages.

You can see that the last bits are 0 for every page in a sequence that is not the last in that sequence.

Recall that `page_nalloc` and `page_free` deal with memory addresses, so you will need to take apart the memory address to find the index in the heap pool.

Memory Example

Let's take, for example, that _heap_start is 0xbeef000, and _heap_end is 0xdead000.

> The linker script will ensure that _heap_start and _heap_end are page aligned. It script sets the kernel stack above the heap since the kernel stack is a fixed size.

The full heap is therefore $\text{heap\_size} = \text{\_heap\_end} - \text{\_heap\_start} = 33284096$ heap_size = _heap_end - _heap_start = 33284096 byte

This means we have $\frac{\text{heap\_size}}{4096} = \frac{33284096}{4096} = 8126$ heap_size / 4096 = 33284096 / 4096 = 8126 pages.

To manage 8126 pages, we need $\frac{8126}{4} = 2031.5 = 2032$ 8126 / 4 = 2031.5 = 2032 bytes.

Recall we need to align the bookkeeping bytes to the next page, so we actually need 4096 bytes (one page) to manage 8126 pages.

Therefore, the first page we can allocate is one page after _heap_start, which is $\text{0xbeef000} + 4096 = \text{0xbef0000}$ 0xbeef000 + 4096 = 0xbef000(

> Write short, static functions to test/set/clear the taken and last bits. Also, write short, static functions to calculate the math to map page addresses to the bookkeeping bits.

Virtual Memory

> _heap_start and _heap_end are **physical memory addresses**, since they come from the linker script. After you get the MMU working properly, you need to ensure that your heap is managed **virtually**.

MMU Functions

Your MMU functions need to be aware of the MMU design. For the RISC-V architecture, we are going to use the Sv39 (supervisor, 39-bit virtual addresses system. Therefore, the following defines/macros will be important.

mmu.h

```c
#define MMU_LEVEL_1G        2
#define MMU_LEVEL_2M        1
#define MMU_LEVEL_4K        0

#define PAGE_SIZE_4K        PAGE_SIZE_AT_LVL(MMU_LEVEL_4K)
#define PAGE_SIZE_2M        PAGE_SIZE_AT_LVL(MMU_LEVEL_2M)
#define PAGE_SIZE_1G        PAGE_SIZE_AT_LVL(MMU_LEVEL_1G)

#define PAGE_SIZE_AT_LVL(x) (1 << (x * 9 + 12))

#define PAGE_SIZE           PAGE_SIZE_4K

// PB_* - page bits
#define PB_NONE             0
#define PB_VALID            (1UL << 0)
#define PB_READ             (1UL << 1)
#define PB_WRITE            (1UL << 2)
#define PB_EXECUTE          (1UL << 3)
#define PB_USER             (1UL << 4)
#define PB_GLOBAL           (1UL << 5)
#define PB_ACCESS           (1UL << 6)
#define PB_DIRTY            (1UL << 7)

// MODE[63:60] (4 bits) in the SATP register.
#define SATP_MODE_BIT       60

// SV39 is MODE=8
#define MODE_SV39           8UL
#define SATP_MODE_SV39      (MODE_SV39 << SATP_MODE_BIT)

// ASID[59:44] (16 bits) in the SATP register.
#define SATP_ASID_BIT       44

// PPN[43:0] (44 bits) in the SATP register
#define SATP_PPN_BIT        0
#define SATP_SET_PPN(x)     ((((uint64_t)(x)) >> 12) & 0xFFFFFFFFFFFUL)
#define SATP_SET_ASID(x)    ((((uint64_t)(x)) & 0xFFFF) << SATP_ASID_BIT)

#define SATP(table, asid) (SATP_MODE_SV39 | SATP_SET_PPN(table) |
SATP_SET_ASID(asid))
#define SATP_KERNEL       SATP(kernel_mmu_table, KERNEL_ASID)
```

You will also need to write four functions.

mmu.h

```c
struct page_table {
    uint64_t entries[PAGE_SIZE / 8];
};

bool     mmu_map(struct page_table *tab,
                 uint64_t vaddr,
                 uint64_t paddr,
                 uint8_t lvl,
                 uint64_t bits);

void     mmu_free(struct page_table *tab);
uint64_t mmu_translate(const struct page_table *tab, uint64_t vaddr);

#define  mmu_translate_ptr(tab, ptr)  (void *)mmu_translate(tab, (uint64_t)ptr)
#define  mmu_translate_ptr_to_u64(tab, ptr)  mmu_translate(tab, (uint64_t)ptr)
#define  MMU_TRANSLATE_PAGE_FAULT -1UL

uint64_t mmu_map_range(struct page_table *tab,
                       uint64_t start_virt,
                       uint64_t end_virt,
                       uint64_t start_phys,
                       uint8_t lvl,
                       uint64_t bits);
```

mmu_map

The `mmu_map` function will map the given virtual address to the physical address and create a leaf at the given level. The `bits` needs to be OR'd with the `PB_VALID` bit. This allows the programmer to specify the permission bits, such as `PB_USER` and/or `PB_READ` and so forth. The page table passed wil be the root table. The reason it is passed in is because you will need to create mappings for the kernel and any user space applications, so this same function will be used for all page tables.

This function returns `true` if the mapping was made, or `false` otherwise. A `false` can be returned if the parameters don't make sense (e.g., lvl is not 0 1, or 2) or if there is not enough memory to create branch tables.

> This function will **overwrite** previous mappings if the same virtual address is provided on the same table.

mmu.c

```c
bool mmu_map(struct page_table *tab,
             uint64_t vaddr,
             uint64_t paddr,
             uint8_t lvl,
             uint64_t bits)
{
    int i;
    // Error check tab, lvl, and bits
    if (tab == NULL || lvl > MMU_LVL_1GB || (bits & 0xE) == 0) {
        return false;
    }

    // Get vpn[0], vpn[1], and vpn[2].
    const uint64_t vpn[] = {(vaddr >> ADDR_0_BIT) & 0x1FF, (vaddr >> ADDR_1_BIT) &
0x1FF,
                            (vaddr >> ADDR_2_BIT) & 0x1FF};

    // Get ppn[0], ppn[1], and ppn[2].
    const uint64_t ppn[] = {(paddr >> ADDR_0_BIT) & 0x1FF, (paddr >> ADDR_1_BIT) &
0x1FF,
                            (paddr >> ADDR_2_BIT) & 0x3FFFFFF};


    for (i = MMU_LEVEL_1G; i > lvl; i -= 1) {
        // Go through the branches.
        // NOTE: you may need to create additional tables.
    }

    // After the loop, you're looking at the leaf @ i.

    return true;
}
```

> Recall that C does not have a **bool** data type unless you include `<stdbool.h>` .

The `mmu_map_range` function needs to map a range of addresses, mapping each page to the corresponding physical address. The following code represen an implementation.

mmu.c

```c
uint64_t mmu_map_range(struct page_table *tab,
                       uint64_t start_virt,
                       uint64_t end_virt,
                       uint64_t start_phys,
                       uint8_t lvl,
                       uint64_t bits)
{
    start_virt            = ALIGN_DOWN_POT(start_virt, PAGE_SIZE_AT_LVL(lvl));
    end_virt              = ALIGN_UP_POT(end_virt, PAGE_SIZE_AT_LVL(lvl));
    uint64_t num_bytes    = end_virt - start_virt;
    uint64_t pages_mapped = 0;

    uint64_t i;
    for (i = 0; i < num_bytes; i += PAGE_SIZE_AT_LVL(lvl)) {
        if (!mmu_map(tab, start_virt + i, start_phys + i, lvl, bits)) {
            break;
        }
        pages_mapped += 1;
    }
    return pages_mapped;
}
```

The mmu_map_range function returns the number of pages that were properly mapped. This is because any individual mmu_map may fail, but we don't want to unwind it. Instead, we will let the programmer decide what to do if all pages aren't mapped.

mmu_free

The `mmu_free` function needs to **recursively** free all of the entries of a given table. Recall that each table could be a branch, which means that the memor address stored in the entry is a page you allocated. All of these pages need to be freed.

mmu.c

```c
void mmu_free(struct page_table *tab)
{
    uint64_t entry;
    int i;
    if (tab == NULL) {
        return;
    }
    // Each entry is 8 bytes, so there are PAGE_SIZE / 8 entries.
    for (i = 0; i < (PAGE_SIZE / 8); i += 1) {
        entry = tab->entries[i];
        // Check if this is a branch, if it is, recurse
        // to the branch.

        // ALL entries should be cleared to 0 after branches
        // return back from the recursion.
        tab->entries[i] = 0;
    }
    page_free(tab);
}
```

mmu_translate

The `mmu_translate` function will translate a virtual address to a physical address given a table. This will be helpful getting the physical address when worrying about hardware drivers, etc.s

mmu.c

```c
uint64_t mmu_translate(const struct page_table *tab, uint64_t vaddr)
{
    int i;
    // Can't translate without a table.
    if (tab == NULL) {
        return MMU_TRANSLATE_PAGE_FAULT;
    }
    uint64_t vpn[] = {(vaddr >> ADDR_0_BIT) & 0x1FF, (vaddr >> ADDR_1_BIT) &
0x1FF,
                      (vaddr >> ADDR_2_BIT) & 0x1FF};

    // Translate and return the physical address.
}
```

Copy To/From

We have made things easier by identity mapping some physical pages with the same virtual address. However, we will eventually need to copy data from/t process which is not identity mapped *and* perhaps the memory addresses are virtually contiguous but not necessarily physically contiguous.

Edit the files called uaccess.h (user access) and uaccess.c to support the following two functions.

uaccess.h

```c
#pragma once

#include <stdint.h>

struct page_table;

uint64_t copy_from(void *dst,
                   const struct page_table *from_table,
                   const void *from,
                   uint64_t size);

uint64_t copy_to(void *to,
                 const struct page_table *to_table,
                 const void *src,
                 uint64_t size);
```

uaccess.c

```c
#include <uaccess.h>
#include <util.h>    // for memcpy
#include <mmu.h>     // for struct page_table

uint64_t copy_from(void *dst,
                   const struct page_table *from_table,
                   const void *from,
                   uint64_t size)
{
    uint64_t bytes_copied = 0;

    // ...

    return bytes_copied;
}

uint64_t copy_to(void *to,
                 const struct page_table *to_table,
                 const void *src,
                 uint64_t size)
{
    uint64_t bytes_copied = 0;

    // ...

    return bytes_copied;
}
```

Copy From

```c
uint64_t copy_from(void *dst,
                   const struct page_table *from_table,
                   const void *from,
                   uint64_t size);
```

The `copy_from` function copies data from the virtual address in `from` to the virtual address in `dst` . The destination is translated by the MMU, but the source will be translated with `mmu_translate` using `from_table` as the page table. The number of bytes to copy is provided in `size` . This function returns the number of bytes copied.

You need to make sure that the from addresses are properly mapped. The reason you return the number of bytes copied is because you may successfully translate say the first three pages, but the fourth page faults. Therefore, you only copy the bytes from the first three pages.

> Do not translate every byte. Instead, copy using `memcpy` until you hit a page boundary, then translate, copy another page, and so forth until you hit `size` or a page fault, whichever comes first.

> Recall that mmu_translate produces a physical address, but since you identity mapped the physical pool in the kernel page table, you can treat the physical address as a virtual address. Usually, you would have to do a reverse lookup using some data structure like a map.

Copy To

```
uint64_t copy_to(void *to,
                 const struct page_table *to_table,
                 const void *src,
                 uint64_t size);
```

The `copy_to` function is analogous to `copy_from`, except it will copy bytes from the virtual address in `src` to the destination address in `to`. The sourc address is translated by the MMU, but the to address needs to be translated given the passed table in `to_table`. Just like `copy_from`, the `to` memory address may span multiple pages which are not necessarily contiguous.

Also, like copy_from, this function will return the total number of bytes copied from the `src` address to the `to` address.

> Do not translate every byte. Instead, copy using `memcpy` until you hit a page boundary, then translate, copy another page, and so forth until you hit `size` or a page fault, whichever comes first.

> Recall that mmu_translate produces a physical address, but since you identity mapped the physical pool in the kernel page table, you can treat the physical address as a virtual address. Usually, you would have to do a reverse lookup using some data structure like a map.

Enabling Code in the Template

In `config.h`, uncomment `USE_MMU` and `USE_HEAP`.

These two defines control certain code in `main.c` that call your `page_init()` function as well as the `heap_init()` function. The `heap_init()` will request a certain number of **continuous** pages from `page_nalloc()` to act as the kernel heap.

> You need to get this right. You will be requesting memory for a lot of things, and the heap is the only way to have **persistent** memory. If you don't get this to work properly, many things in your kernel, including the utility library, will not function properly.

You can see in main.c that uncommenting USE_MMU performs the following actions.

```
#ifdef USE_MMU
    page_init();
    struct page_table *pt     = page_zalloc();
    // kernel_mmu_table is global and exported throughout.
    kernel_mmu_table = pt;
    // Map memory segments for our kernel
    mmu_map_range(pt, sym_start(text), sym_end(heap), sym_start(text),
MMU_LEVEL_1G,
                  PB_READ | PB_WRITE | PB_EXECUTE);
    // PLIC
    mmu_map_range(pt, 0x0C000000, 0x0C2FFFFF, 0x0C000000, MMU_LEVEL_2M, PB_READ |
PB_WRITE);
    // PCIe ECAM
    mmu_map_range(pt, 0x30000000, 0x30FFFFFF, 0x30000000, MMU_LEVEL_2M, PB_READ |
PB_WRITE);
    // PCIe MMIO
    mmu_map_range(pt, 0x40000000, 0x4FFFFFFF, 0x40000000, MMU_LEVEL_2M, PB_READ |
PB_WRITE);

    // TODO: turn on the MMU when you've written the src/mmu.c functions
    CSR_WRITE("satp", SATP_KERNEL);
    SFENCE_ALL();
#endif
```

Therefore, you need to make sure you have your page grained allocator as well as the mmu mapping functions working properly before uncommenting `USE_MMU` and `USE_HEAP` .