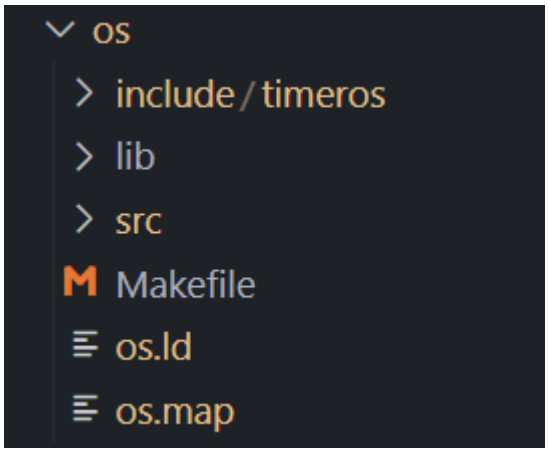
用户态printf以及物理内存管理 | TimerのBlog

yanglianoo.github.io/2023/08/30/用户态printf以及物理内存管理

2023年8月30日

1. os文件架构修改

在开始启用mmu之前我对os的文件编译体系进行了修改，如下：



将头文件全部放在了include/timeros下，内核的源码放在了src目录下，lib目录下放的是一些通用的函数库，比如之前的string.c，以及马上要新增的和printf相关的代码。

所有的头文件都被修改成了这种形式：添加了一个TOS的前缀

c

```
#ifndef
TOS_OS_H__
#define
TOS_OS_H__
#endif
```

然后修改了makefile

```
INCLUDE:=-I./include
SRC:=./src
LIB:=./lib
# 将源文件替换为 .o 文件
OBJS = $(SRCS_ASM:.S=.o)
OBJS += $(SRCS_C:.c=.o)


os.elf: ${OBJS}
    ${CC} ${CFLAGS} -T os.ld -Wl,-Map=os.map -o os.elf $^
    ${OBJCOPY} -O binary os.elf os.bin

%.o : $(SRC)/%.c
    ${CC} ${CFLAGS} $(INCLUDE) -c -o $@ $<

%.o : $(LIB)/%.c
    ${CC} ${CFLAGS} $(INCLUDE) -c -o $@ $<

%.o : $(SRC)/%.S
    ${CC} ${CFLAGS} $(INCLUDE) -c -o $@ $<
```

修改的地方用红色方框圈了起来，主要就是用于调整文件结构后的编译。

2. 用户态的printf实现

在之前的应用程序中，我们一直使用sys_write的系统调用来向串口输出数据，这样及其不方便，因此需要实现一个用户态的printf函数。实现方式和之前的内核实现的基本一样。

```
static int _vprintf(const char* s, va_list vl)
{
    int res = _vsnprintf(NULL, -1, s, vl);
    if (res+1 >= sizeof(out_buf)) {
        uart_puts("error: output string size overflow
        while(1) {}
    }
    _vsnprintf(out_buf, res + 1, s, vl);
    uart_puts(out_buf);
    return res;
}
```

差别就在于需要把上面这个uart_puts函数换成sys_write就行了。

我首先把之前实现的printf函数名改成了printk，这是内核专属的，在S态用到的printf都要更改成printk，

```c
int printk(const char* s, ...)
{
    int res = 0;
    va_list vl;
    va_start(vl, s);
    res = _vprintf(s, vl);
    va_end(vl);
    return res;
}
```

然后在lib目录下新增printf.c和vsprintf.c,其中vsprintf.c的内容就是把之前的_vsnprintf函数移动到了此文件而已，然后printf.c如下：

c

```c
static char out_buf[1000]; // buffer for
vprintf()
static int vprintf(const char* s, va_list
vl)
{
        int res = _vsnprintf(NULL, -1, s,
vl);
        _vsnprintf(out_buf, res + 1, s, vl);
        sys_write(stdout,out_buf,res + 1);
        return res;
}

int printf(const char* s, ...)
{
        int res = 0;
        va_list vl;
        va_start(vl, s);
        res = vprintf(s, vl);
        va_end(vl);
        return res;
}
```

可以看见和printk的实现一模一样，只是使用了sys_write(stdout,out_buf,res + 1);来输出数据，_vsnprintf函数作为printf和printk公用函数。同时新增了一个头文件stdio.h

c

```
int _vsnprintf(char * out, size_t n, const char* s,
va_list vl);
void panic(char *s);
int printk(const char* s, ...);
int printf(const char* s, ...);

/* 文件描述符 */
typedef enum std_fd_t
{
    stdin,
    stdout,
    stderr,
} std_fd_t;
```

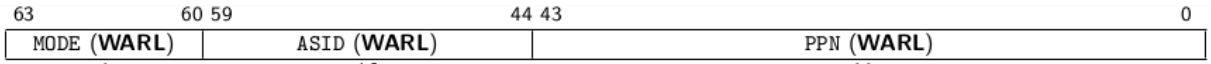
3. riscv的分页机制

在riscv体系中，有三种地址转换机制：

- Sv32：仅支持32位riscv处理器，是一个二级页表结构，支持32位虚拟地址转换
- Sv39：支持64位riscv处理器，是一个三级页表结构，支持39位虚拟地址转换
- Sv48：支持64位riscv处理器，是一个四级页表结构，支持48位虚拟地址转换

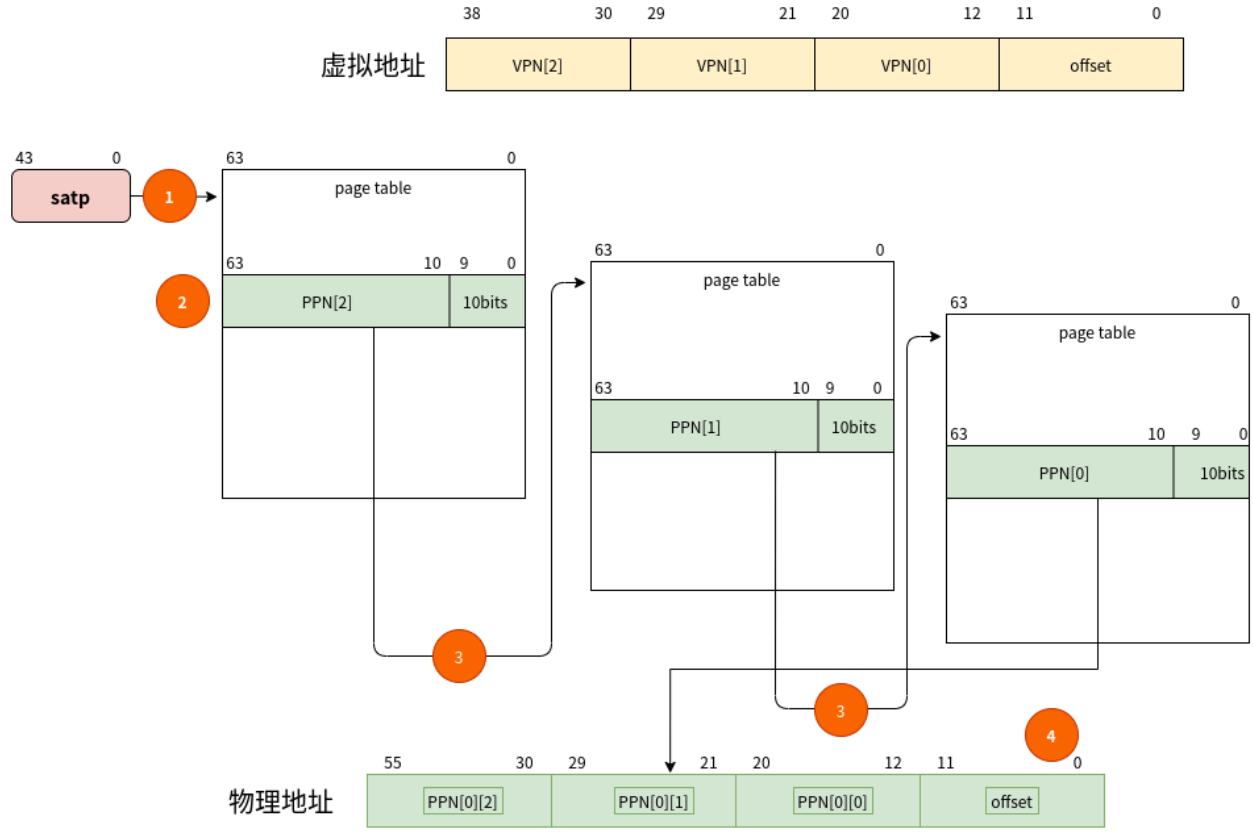
目前RISCV体系通常支持4KB大小的页面粒度，也支持2MB，1GB大小的块粒度。在我们的操作系统中用到的是Sv39页表映射，关于Sv39的具体内容本博客不做解释，主要是写起来他麻烦了哈哈，网上有很多资料：

首先先来看一下satp寄存器，这是一个S特权级的控制寄存器



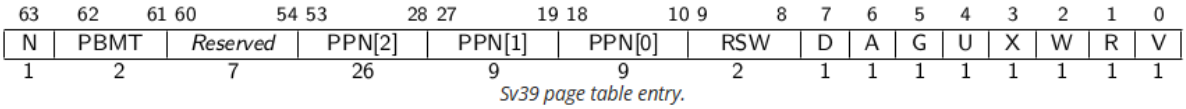
Supervisor address translation and protection register satp when SXLEN=64, for MODE values Bare, Sv39, Sv48, and Sv57.

- MODE 控制 CPU 使用哪种页表实现；当 MODE 设置为 0 的时候，代表所有访存都被视为物理地址；而设置为 8 的时候，SV39 分页机制被启用，所有 S/U 特权级的访存被视为一个 39 位的虚拟地址，它们需要先经过 MMU 的地址转换流程，如果顺利的话，则会变成一个 56 位的物理地址来访问物理内存；否则则会触发异常。
- ASID 表示地址空间标识符，这里还没有涉及到进程的概念，我们不需要管这个地方；
- PPN 存的是根页表所在的物理页号。这样，给定一个虚拟页号，CPU 就可以从三级页表的根页表开始一步步的将其映射到一个物理页号。



- 首先从satp寄存器的低44位取出一级页表的物理页号，乘以 PAGESIZE (4 KiB) 后，得到 3 级页表地址，取出虚拟地址的VPN[2],去3级页表中寻找对应的页表项。
- 然后从三级页表中得到的页表项中存储了二级页表的物理页号，乘以 PAGESIZE (4 KiB) 后，得到 2级页表地址，取出虚拟地址的VPN[1],去2级页表中寻找对应的页表项。
- 然后从二级页表中得到的页表项中存储了1级页表的物理页号，乘以 PAGESIZE (4 KiB) 后，得到 1级页表地址，取出虚拟地址的VPN[0],去1级页表中寻找对应的页表项。
- 将1级页表中得到的页表项中存储了虚拟地址的物理页号，乘以 PAGESIZE (4 KiB) 后，得到实际的物理页号，然后再加上虚拟地址的最后12位offset就最终得到了物理地址。

页表项的最后10bit代表的是页表项的属性：



Sv39 page table entry.

- V(Valid)：仅当位 V 为 1 时，页表项才是合法的；
- R(Read)/W(Write)/X(eXecute)：分别控制索引到这个页表项的对应虚拟页面是否允许读/写/执行；
- U(User)：控制索引到这个页表项的对应虚拟页面是否在 CPU 处于 U 特权级的情况下是否被允许访问；
- G：暂且不理会；
- A(Accessed)：处理器记录自从页表项上的这一位被清零之后，页表项的对应虚拟页面是否被访问过；

- D(Dirty)：处理器记录自从页表项上的这一位被清零之后，页表项的对应虚拟页面是否被修改过
- RSW：预留位

X	W	R	Meaning
0	0	0	Pointer to next level of page table.
0	0	1	Read-only page.
0	1	0	*Reserved for future use.*
0	1	1	Read-write page.
1	0	0	Execute-only page.
1	0	1	Read-execute page.
1	1	0	*Reserved for future use.*
1	1	1	Read-write-execute page.

4. 物理内存管理

要开启mmu之前，内核需要对拥有的物理内存进行分配管理，对物理内存一页为单位进行分配和释放。

4.1 xv6-riscv的物理内存管理

首先先来看，xv6是如何管理的，代码在这里：xv6-riscv/kernel/kalloc.c at riscv · mit-pdos/xv6-riscv (github.com)

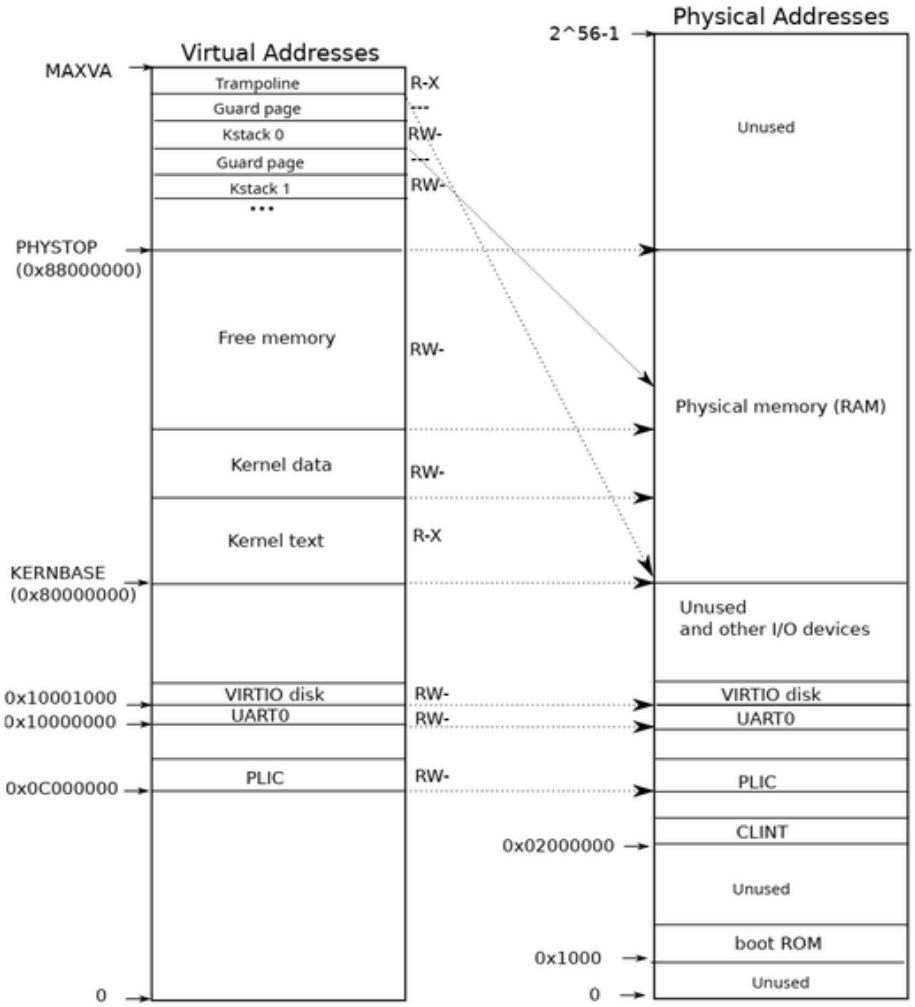


Figure 3.3: On the left, xv6's kernel address space. RWX refer to PTE read, write, and execute permissions. On the right, the RISC-V physical address space that xv6 expects to see.

xv6内核的起始地址是从`KERNBASE=0x80000000`开始，结束的地方是`PHYSTOP=0x88000000`，这之间128M的地方，其中内核的代码段是可读可执行的，内核数据段代码是可读可写的。所以我们实际可以分配和管理的地址是从内核段的代码结束的地方开始的，在xv6的链接文件中指明了空闲物理内存开始的地方：

```
SECTIONS
{
  /*
   * ensure that entry.S / _entry is at 0x80000000,
   * where qemu's -kernel jumps.
   */
  . = 0x80000000;

  .text : {
    *(.text .text.*)
    . = ALIGN(0x1000);
    _trampoline = .;
    *(trampsec)
    . = ALIGN(0x1000);
    ASSERT(. - _trampoline == 0x1000, "error: trampoline larger than one page");
    PROVIDE(etext = .);
  }

  .rodata : {
    . = ALIGN(16);
    *(.srodata .srodata.*) /* do not need to distinguish this from .rodata */
    . = ALIGN(16);
    *(.rodata .rodata.*)
  }

  .data : {
    . = ALIGN(16);
    *(.sdata .sdata.*) /* do not need to distinguish this from .data */
    . = ALIGN(16);
    *(.data .data.*)
  }

  .bss : {
    . = ALIGN(16);
    *(.sbss .sbss.*) /* do not need to distinguish this from .bss */
    . = ALIGN(16);
    *(.bss .bss.*)
  }

  PROVIDE(end = .);
}
```

内核段代码开始位置

内核段代码结束位置

所以现在来看xv6的kalloc.c:

首先就是拿到了end所代表的地址

c

```
extern char end[]; // first address after
kernel.
                   // defined by kernel.ld.
```

然后定义了一个数据结构kmem，这个数据结构包含一个锁和一个链表，这就是xv6管理物理内存的核心数据结构，我们可以先忽略掉锁

C

```
struct run {
  struct run
*next;
};

struct {
  struct spinlock
lock;
  struct run
*freelist;
} kmem;
```

然后定义了kinit()函数，这个函数用来扫描从end~PHYSTOP之间可用的物理内存页，将可用的物理内存页通过kmem维护起来，同时将可用的空闲物理页中的每个字节的数据进行填充，这是用于初始化的时候

C

```
void
kinit()
{
  initlock(&kmem.lock,
"kmem");
  freerange(end,
(void*)PHYSTOP);
}
```

freerange就是用来扫描内存的，可以看见从物理内存开始位置到结束，freerange会以页为单位来释放内存，其中会去调用kfree这个函数

C

```
void
freerange(void *pa_start, void *pa_end)
{
  char *p;
  p = (char*)PGROUNDUP((uint64)pa_start);
  for(; p + PGSIZE <= (char*)pa_end; p +=
PGSIZE)
    kfree(p);
}
```

在kfree中，对此页内存进行数据填充，然后将指向此页内存的指针放到kmem的链表头部，然后再让链表头指针前移

C

```
void
kfree(void *pa)
{
  struct run *r;

  if(((uint64)pa % PGSIZE) != 0 || (char*)pa < end || (uint64)pa >=
PHYSTOP)
    panic("kfree");

  // Fill with junk to catch dangling refs.
  memset(pa, 1, PGSIZE);

  r = (struct run*)pa;

  acquire(&kmem.lock);
  r->next = kmem.freelist;
  kmem.freelist = r;
  release(&kmem.lock);
}
```

与kfree与之对应的就是分配一页内存的函数kalloc函数，此函数将返回分配的物理内存页的指针。发配的逻辑就是从kmemd的链表头部取出一个空闲页块的指针，然后将链表头指针后移

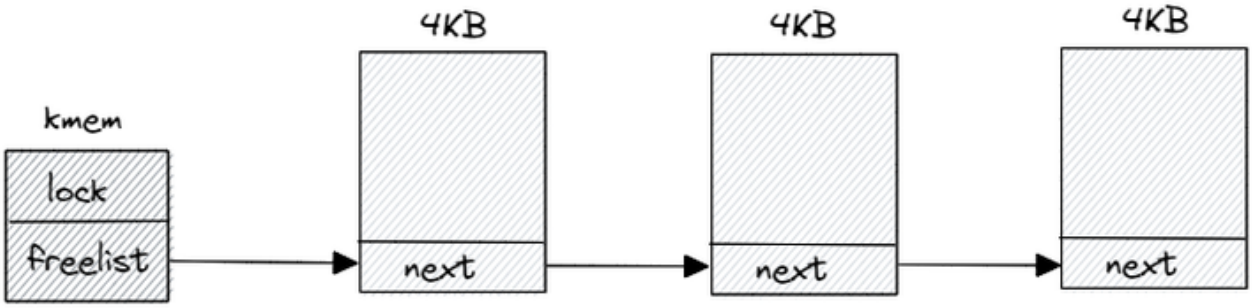
C

```
// Allocate one 4096-byte page of physical
memory.
// Returns a pointer that the kernel can
use.
// Returns 0 if the memory cannot be
allocated.
void *
kalloc(void)
{
  struct run *r;

  acquire(&kmem.lock);
  r = kmem.freelist;
  if(r)
    kmem.freelist = r->next;
  release(&kmem.lock);

  if(r)
    memset((char*)r, 5, PGSIZE); // fill
with junk
  return (void*)r;
}
```

所以最后空闲链表就把空闲的物理内存串了起来：



可以看见xv6对空闲物理内存的管理十分的简洁明了。

4.2 rCore的物理内存管理

接下来我们再来看看操蛋的rCore，不知道是不是因为为了使用rust的语法特性，`rCore`的物理内存管理搞得我迷迷糊糊，绕过来绕过去。

`rCore`采用的是栈式物理页帧管理策略，核心的数据结构如下：

rust

```
pub struct StackFrameAllocator {
    current: usize,  //空闲内存的起始物
理页号
    end: usize,      //空闲内存的结束物
理页号
    recycled: Vec<usize>,
}
```

其中各字段的含义是：物理页号区间 [`current` , `end`) 此前均 *从未* 被分配出去过，而向量 `recycled` 以后入先出的方式保存了被回收的物理页号。

因此为了实现这种栈式的管理，我们先得来实现一个栈的数据结构，在rust/c++中倒是有方便的vector可以使用，但是我们使用的是c，所以有点麻烦，在`lib`目录下新建了一个`stack.c`的文件，在`timeros`目录下新建了`stack.h`的头文件

C

```c
#include <timeros/stack.h>

// 初始化栈
void initStack(Stack *stack) {
    stack->top = -1;
}

// 判断栈是否为空
bool isEmpty(Stack *stack) {
    return stack->top == -1;
}

// 判断栈是否已满
bool isFull(Stack *stack) {
    return stack->top == MAX_SIZE -
1;
}

// 入栈操作
void push(Stack *stack, u64 value) {
    if (isFull(stack)) {
        printk("Stack overflow\n");
        return;
    }
    stack->data[++stack->top] =
value;
}

// 出栈操作
u64 pop(Stack *stack) {
    if (isEmpty(stack)) {
        printk("Stack underflow\n");
        return -1; // 表示栈为空或操
作失败
    }
    return stack->data[stack->top-
-];
}

// 获取栈顶元素
u64 top(Stack *stack) {
    if (isEmpty(stack)) {
        printk("Stack is empty\n");
        return -1; // 表示栈为空
    }
    return stack->data[stack->top];
}
```

c

```
#ifndef TOS_STACK_H__
#define TOS_STACK_H__

#include "os.h"

#define MAX_SIZE 10000

typedef struct {
    u64 data[MAX_SIZE];
    int top;    // 不能定义成无符号类型，不然会导致
-1 > 0
} Stack;

bool isEmpty(Stack *stack);
bool isFull(Stack *stack);
void push(Stack *stack, u64 value);
u64 pop(Stack *stack);
u64 top(Stack *stack);

#endif
```

stack的数据结构代码还是比较简单的，这里有一点说明：

c

```
typedef struct {
    u64 data[MAX_SIZE];
    int top;    // 不能定义成无符号类型，不然会导致
-1 > 0
} Stack;
```

栈顶需要定义成int类型，不能定义成无符号类型，因为对栈初始化时，top的值被设置为-1，但是后面会让top和0进行大小比较，如果设置成无符号会导致结果出错。第二个就是data是一个u64的数组，因为栈中维护的是物理页号，物理页号是u64类型的。

在`src`目录下新建一个`address.c`，`timeros`目录下新建了`address.h`的头文件

c

```c
#ifndef TOS_ADDRESS_H
#define TOS_ADDRESS_H

#include <timeros/os.h>
#include <timeros/stack.h>
#include <timeros/string.h>
#include <timeros/assert.h>

#define PAGE_SIZE 0x1000      // 4kb  一页的大小
#define PAGE_SIZE_BITS   0xc  // 12   页内偏移地址长度

#define PA_WIDTH_SV39 56      //物理地址长度
#define VA_WIDTH_SV39 39      //虚拟地址长度
#define PPN_WIDTH_SV39 (PA_WIDTH_SV39 - PAGE_SIZE_BITS)  // 物理页号 44位
[55:12]
#define VPN_WIDTH_SV39 (VA_WIDTH_SV39 - PAGE_SIZE_BITS)  // 虚拟页号 27位
[38:12]

#define MEMORY_END 0x80800000    // 0x80200000 ~ 0x80800000
#define MEMORY_START 0x80400000

/* 物理地址 */
typedef struct {
    uint64_t value;
} PhysAddr;

/* 虚拟地址 */
typedef struct {
    uint64_t value;
} VirtAddr;

/* 物理页号 */
typedef struct {
    uint64_t value;
} PhysPageNum;

/* 虚拟页号 */
typedef struct {
    uint64_t value;
} VirtPageNum;
```

address.h中定义了PhysAddr、VirtAddr、PhysPageNum、VirtPageNum，全部定义成结构体的类型，在rCore中可以为这些结构体实现操作函数，但是c语言没有面向对象的特性，因此就只有一个手动实现：

c

```
/* 给定一个u64 转换为PhysAddr */
PhysAddr phys_addr_from_size_t(uint64_t v) {
    PhysAddr addr;
    addr.value = v & ((1ULL << PA_WIDTH_SV39) - 1);
    return addr;
}

/*给定一个u64 转换为PhysPageNum */
PhysPageNum phys_page_num_from_size_t(uint64_t v) {
    PhysPageNum pageNum;
    pageNum.value = v & ((1ULL << PPN_WIDTH_SV39) - 1);
    return pageNum;
}

/* 给定一个PhysAddr转换为u64 */
uint64_t size_t_from_phys_addr(PhysAddr v) {
    return v.value;
}

/* 给定一个PhysPageNum 转换为u64 */
uint64_t size_t_from_phys_page_num(PhysPageNum v) {
    return v.value;
}

/* 从物理页号转换为实际物理地址 */
PhysAddr phys_addr_from_phys_page_num(PhysPageNum ppn)
{
    PhysAddr addr;
    addr.value = ppn.value << PAGE_SIZE_BITS ;
    return addr;
}

/* 给定一个u64 转换为VirtAddr  */
VirtAddr virt_addr_from_size_t(uint64_t v) {
    VirtAddr addr;
    addr.value = v & ((1ULL << VA_WIDTH_SV39) - 1);
    return addr;
}

/* 给定一个u64 转换为VirtPageNum  */
VirtPageNum virt_page_num_from_size_t(uint64_t v) {
    VirtPageNum pageNum;
    pageNum.value = v & ((1ULL << VPN_WIDTH_SV39) - 1);
    return pageNum;
}

/*给定一个VirtAddr 转换为一个u64 */
uint64_t size_t_from_virt_addr(VirtAddr v) {
    if (v.value >= (1ULL << (VA_WIDTH_SV39 - 1))) {
        return v.value | ~((1ULL << VA_WIDTH_SV39) - 1);
    } else {
        return v.value;
    }
}

/* 给定一个VirtPageNum 转换为 u64*/
uint64_t size_t_from_virt_page_num(VirtPageNum v) {
    return v.value;
```

```
}

/* 物理地址向下取整 */
PhysPageNum floor_phys(PhysAddr phys_addr) {
    PhysPageNum phys_page_num;
    phys_page_num.value = phys_addr.value / PAGE_SIZE;
    return phys_page_num;
}

/* 物理地址向上取整 */
PhysPageNum ceil_phys(PhysAddr phys_addr) {
    PhysPageNum phys_page_num;
    phys_page_num.value = (phys_addr.value + PAGE_SIZE - 1) /
PAGE_SIZE;
    return phys_page_num;
}

/* 把虚拟地址转换为虚拟页号 */
VirtPageNum virt_page_num_from_virt_addr(VirtAddr virt_addr)
{
    VirtPageNum vpn;
    vpn.value =  virt_addr.value / PAGE_SIZE;
    return vpn;
}
```

上面的函数都是做一些转换工作，然后我们来定义栈式的内存管理数据结构：

c

```
typedef struct
{
    uint64_t current;   //空闲内存的起始物
理页号
    uint64_t  end;      //空闲内存的结束物
理页号
    Stack recycled;     //
}StackFrameAllocator;
```

首先是第一个new函数，用于创建`FrameAllocator`的实例：只需将区间两端均设为 0 ，然后创建一个初始化栈；

c

```
void StackFrameAllocator_new(StackFrameAllocator*
allocator) {
    allocator->current = 0;
    allocator->end = 0;
    initStack(&allocator->recycled);
}
```

然后是init函数，用于将自身的 [current,end) 初始化为可用物理页号区间：

c

```
void StackFrameAllocator_init(StackFrameAllocator *allocator, PhysPageNum l,
PhysPageNum r) {
    allocator->current = l.value;
    allocator->end = r.value;
}
```

接下来就是物理页帧的分配和实现：

c

```
PhysPageNum StackFrameAllocator_alloc(StackFrameAllocator
*allocator) {
    PhysPageNum ppn;
    if (allocator->recycled.top >= 0) {
        ppn.value = pop(&(allocator->recycled));
    } else {
        if (allocator->current == allocator->end) {
            ppn.value = 0; // Return 0 as None
        } else {
            ppn.value = allocator->current++;
        }
    }
    /* 清空此页内存 ： 注意不能覆盖内核代码区，分配的内存只能是未使用
部分*/
    PhysAddr addr = phys_addr_from_phys_page_num(ppn);
    memset(addr.value,0,PAGE_SIZE);
    return ppn;
}
```

在分配 alloc 的时候，首先会检查栈 recycled 内有没有之前回收的物理页号，如果有的话直接弹出栈顶并返回；否则的话我们只能从之前从未分配过的物理页号区间 [current , end) 上进行分配，我们分配它的左端点 current ，同时将管理器内部维护的 current 加 1 代表 current 已被分配了。然后清空此页内存，全部初始化为0，最后返回分配的页的物理页号。

C

```
void StackFrameAllocator_dealloc(StackFrameAllocator *allocator, PhysPageNum
ppn) {
    uint64_t ppnValue = ppn.value;
    // 检查回收的页面之前一定被分配出去过
    if (ppnValue >= allocator->current) {
        printk("Frame ppn=%lx has not been allocated!\n", ppnValue);
        return;
    }
    // 检查未在回收列表中
    if(allocator->recycled.top>=0)
    {
        for (size_t i = 0; i <= allocator->recycled.top; i++)
        {
            if(ppnValue ==allocator->recycled.data[i] )
            return;
        }
    }
    // 回收物理内存页号
    push(&(allocator->recycled), ppnValue);
}
```

在回收 `dealloc` 的时候，我们需要检查回收页面的合法性，然后将其压入 `recycled` 栈中。回收页面合法有两个条件：

- 该页面之前一定被分配出去过，因此它的物理页号一定 < `current` ；
- 该页面没有正处在回收状态，即它的物理页号不能在栈 `recycled` 中找到。

在上面的代码中用到了一个`memset`函数，这个函数实现在`string.c`中:

C

```
//复制字符 ch（一个无符号字符）到参数 dest 所指向的字符串的前 n 个
字符。
void* memset(void *dest, int ch, size_t count)
{
    char *ptr = dest;
    while (count--)
    {
        *ptr++ = ch;
    }
    return dest;
}
```

我们来编写测试代码测试一下：

c

```
static StackFrameAllocator FrameAllocatorImpl;
void frame_allocator_test()
{
    StackFrameAllocator_new(&FrameAllocatorImpl);
    StackFrameAllocator_init(&FrameAllocatorImpl, \
            floor_phys(phys_addr_from_size_t(MEMORY_START)), \
            ceil_phys(phys_addr_from_size_t(MEMORY_END)));
    printk("Memoery start:%d\n",floor_phys(phys_addr_from_size_t(MEMORY_START)));
    printk("Memoery end:%d\n",ceil_phys(phys_addr_from_size_t(MEMORY_END)));
    PhysPageNum frame[10];
    for (size_t i = 0; i < 5; i++)
    {
         frame[i] = StackFrameAllocator_alloc(&FrameAllocatorImpl);
         printk("frame id:%d\n",frame[i].value);
    }
    for (size_t i = 0; i < 5; i++)
    {
        StackFrameAllocator_dealloc(&FrameAllocatorImpl,frame[i]);
        printk("allocator-
>recycled.data.value:%d\n",FrameAllocatorImpl.recycled.data[i]);
        printk("frame id:%d\n",frame[i].value);
    }
    PhysPageNum frame_test[10];
    for (size_t i = 0; i < 5; i++)
    {
         frame[i] = StackFrameAllocator_alloc(&FrameAllocatorImpl);
        printk("frame id:%d\n",frame[i].value);
    }
}
```

上面的测试函数一次调用了new，init，然后尝试分配五页内存，并打印五页内存的物理页号，然后将分配的五页内存释放掉，此时这五页内存应该会全部被压入recycled栈中，然后再次分配五页内存，此时分配的话就是从recycled中pop的内存了。强调一下在StackFrameAllocator_init函数中传入的起始物理内存的地址和上面xv6的一样，必须在内核代码段之上，在头文件中进行了定义：

c

```
#define MEMORY_END 0x80800000    // 0x80200000 ~
0x80800000
#define MEMORY_START 0x80400000
```

来编译测试一下：修改一下main函数：

c

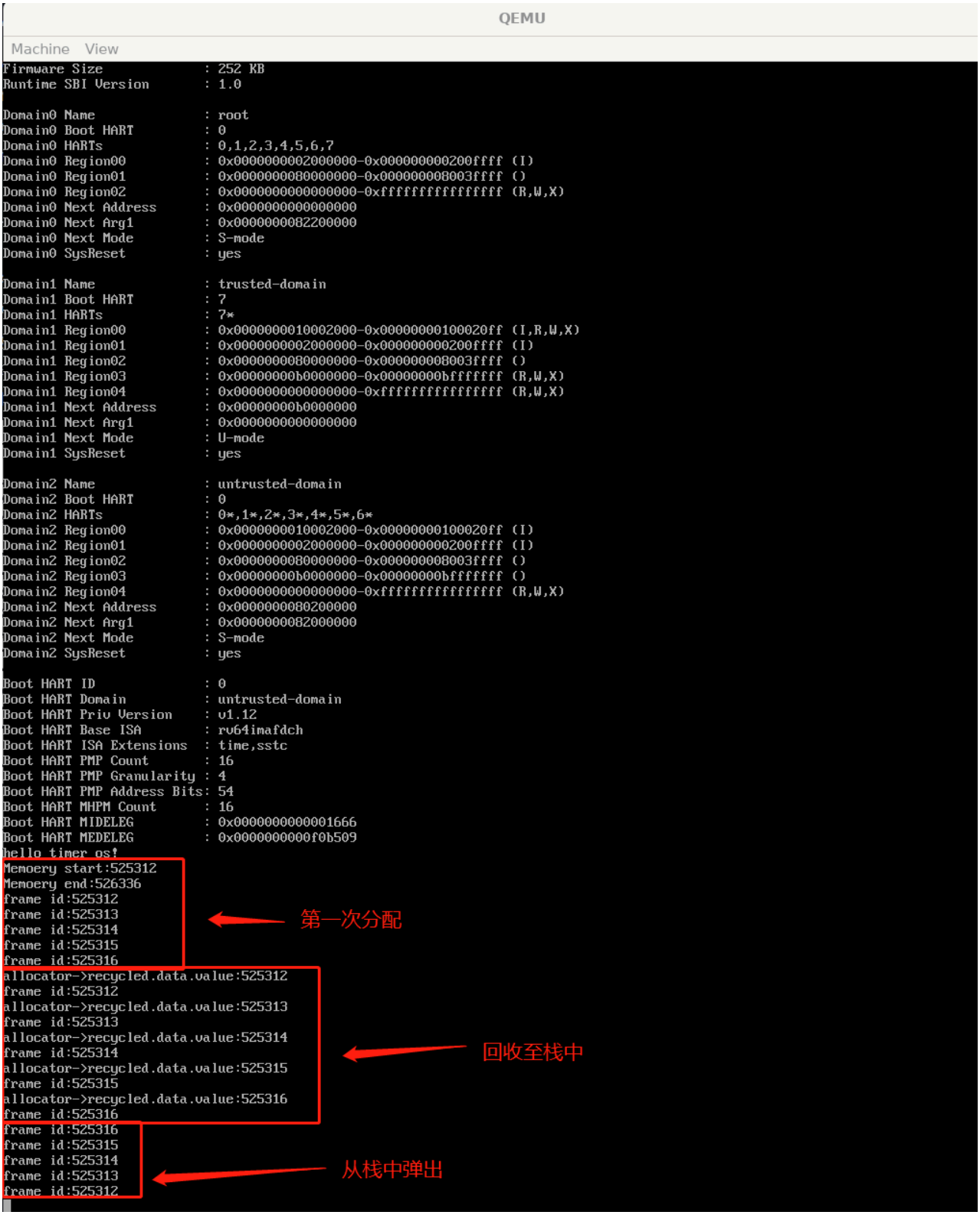
```
extern void
frame_allocator_test();
void os_main()
{
   printk("hello timer
os!\n");
   frame_allocator_test();
   while (1)
   {
      /* code */
   }
   // trap_init();

   // task_init();

   // timer_init();

   // run_first_task();
}
```

编译运行，结果如下



参考链接

- 管理 SV39 多级页表 - rCore-Tutorial-Book-v3 3.6.0-alpha.1 文档 (rcore-os.cn)
- SV39 多级页表的硬件机制 - rCore-Tutorial-Book-v3 3.6.0-alpha.1 文档 (rcore-os.cn)
- xv6 物理内存管理 - 知乎 (zhihu.com)
- The RISC-V Instruction Set Manual, Volume II: Privileged Architecture | Five EmbedDev (five-embeddev.com)
- RISC-V from Scratch 7 - 峰子的乐园 (dingfen.github.io)

相关推荐