南开大学
Nankai University

南　开　大　学

计 算 机 学 院

计算机系统设计

PA **实验四报告**

朱浩泽 1911530

年级：2019 级

专业：计算机科学与技术

指导教师：卢冶

2022 年 6 月 7 日

目录

一、 概述

(一) 实验目的

- 学习虚拟内存映射，并实现分页机制
- 学习上下文切换的基本原理并实现上下文切换、进程调度与分时多任务
- 学习硬件中断并实现时钟中断

(二) 实验内容

- 虚拟地址空间的作用，实现分页机制，并让用户程序运行在分页机制上
- 实现内核自陷、上下文切换与分时多任务
- 解决阶段二分时多任务的隐藏 bug: 改为使用时钟中断来进行进程调度
- 实现当前运行游戏的切换，使不同的游戏与 hello 程序分时运行

二、 阶段一：实现分页机制

(一) 基础结构

首先修改寄存器结构体，在 nemu/include/cpu/reg.h 当中，添加 CR0 和 CR3 两个控制寄存器：

```
struct IDTR
  {
    /* data */
    uint32_t base;
    uint16_t limit;
  } idtr;

  rtlreg_t cs;
  rtlreg_t es; // 配x64
  rtlreg_t ds;
  uint32_t CR0;
  uint32_t CR3;
  bool INTR;
} CPU_state;
```

然后在 nemu/src/monitor/monitor.c 的 resart 函数中初始化 CR0 的值

```
static inline void restart() {
  /* Set the initial instruction pointer. */
  cpu.eip = ENTRY_START;
  cpu.cs = 8;
  cpu.CR0 = 0x60000011;
  unsigned int origin = 2;
  memcpy(&cpu.eflags, &origin, sizeof(cpu.eflags));

#ifdef DIFF_TEST
  init_qemu_reg();
#endif
}
```

打开 HAS_PTE 宏，让 Nanos-lite 在初始化的时候首先调用 init_mm() 函数。在 nanos-lite 目录下 make run，报错如下：

到目前这个阶段而言，init_mm() 函数的工作主要有两方面，一是将空闲物理页的首地址设置为堆区首地址；二是调用 _pte_init() 函数，填写内核的页目录和页表，设置控制寄存器，开启分页机制。

于是，我们首先需要实现新的指令函数，来完成对两个控制寄存器的操作。相关代码在 nemu/include/cpu/rtl.h 中添加。

首先完成 nemu/include/cpu/rtl.h 中的 rtl_store_cr(int r, const rtlreg_t* src) 函数，实现对控制信息的存储，根据情况把参数保存到 CR0 寄存器或 CR3 寄存器当中。

```
static inline rtl_store_cr(int r, const rtlreg_t* src) {
  switch (r)
  {
  case 0:
    cpu.CR0 = *src;
    return;
  case 3:
    cpu.CR3 = *src;
    return;
  default:
    assert(0);
  }
  return;
}
```

完成 nemu/include/cpu/rtl.h 中的 rtl_load_cr(rtlreg_t* dest, int r) 函数，实现对控制信息的读取，根据情况把 CR0 或 CR3 寄存器当中存储的信息读取到 dest 参数中。

```
static inline rtl_load_cr(rtlreg_t* dest, int r) {
  switch (r)
  {
  case 0:
    *dest = cpu.CR0;
    return;
    break;
  case 3:
    *dest = cpu.CR3;
    return;
  default:
    assert(0);
  }
  return;
}
```

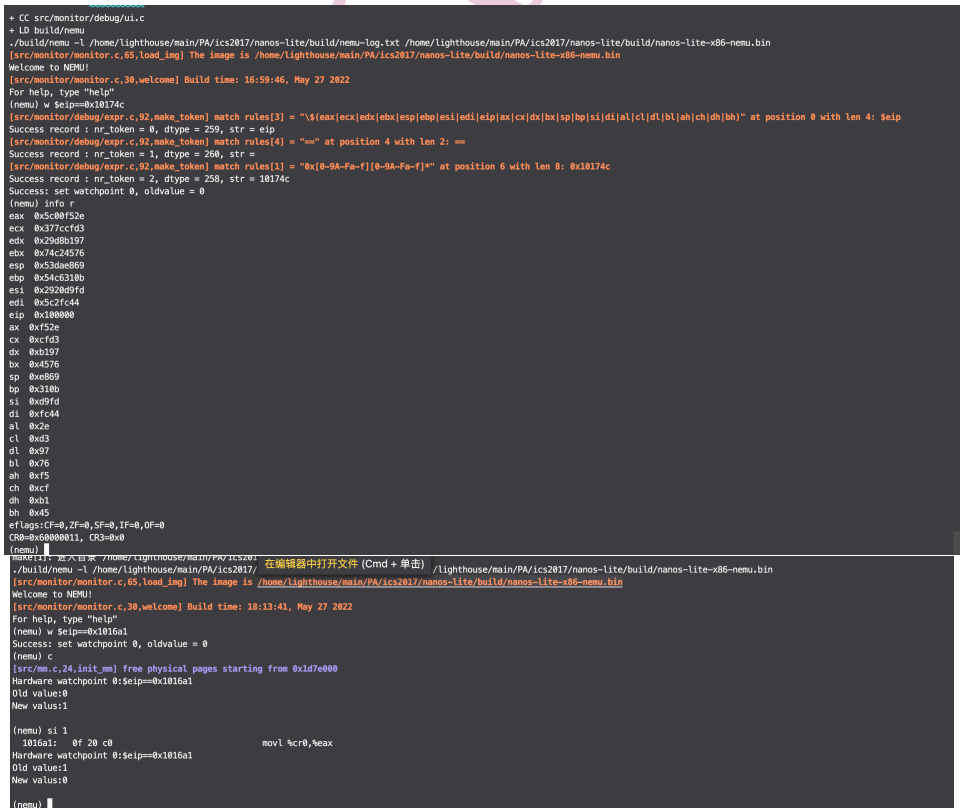
然后完成 init_mm() 函数遇到的指令，即 CR3 或 CR0 的 mov 操作。在 nemu/src/cpu/decode/decode.c 中实现译码函数，其中 make_DHelper(mov_load_cr) 完成的是把控制寄存器中所存储的值进行加载读取的操作，make_DHelper(mov_load_cr) 完成的是把目标寄存器的值保存到控制寄存器的操作。

```
make_DHelper(mov_load_cr) {
  decode_op_rm(eip, id_dest, false, id_src, false);
  rtl_load_cr(&id_src -> val, id_src -> reg);
#ifdef DEBUG
  snprintf(id_src -> str, 5, "%%cr%d", id_dest -> reg);
#endif
}

make_DHelper(mov_store_cr) {
  decode_op_rm(eip, id_src, true, id_dest, false);
#ifdef DEBUG
  snprintf(id_src -> str, 5, "%%cr%d", id_dest -> reg);
#en
```

最后在 nemu/src/cpu/exec/data-mov.c 中添加实际的指令函数 make_EHelper(mov_store_cr)，来完成控制寄存器的 store 操作。

```
make_EHelper(mov_store_cr) {
  rtl_store_cr(id_dest -> reg, &id_src -> val);
  print_asm_template2(mov);
}
```

到此，分页机制的数据结构基础部分我们已经基本完成。通过设置断点，我们可以看到如下内容：



（二） 虚拟地址的访问的转换

接下来需要对 nemu/src/memory/memory.c 中的 vaddr_read() 和 vaddr_write() 函数作少量修改、实现 page_translate() 函数，使得所有虚拟地址的访问都需要经过分页地址的转换。

首先，在 nemu/src/memory/memory.c 中添加我们需要的宏定义。

```
#define PTXSHFT 12 //线性地址偏移量
#define PDXSHFT 22 //线性地址偏移量

#define PTE_ADDR(pte) ((uint32_t)(pte) & ~0xfff)
#define PDX(va) (((uint32_t)(va) >> PDXSHFT) & 0x3ff)
#define PTX(va) (((uint32_t)(va) >> PTXSHFT) & 0x3ff)
#define OFF(va) ((uint32_t)(va) & 0xfff)
```

vaddr_read(vaddr_t addr, int len) 函数实现了读地址时的虚拟地址转换。

```
uint32_t vaddr_read(vaddr_t addr, int len) {
  if(PTE_ADDR(addr) != PTE_ADDR(addr + len -1)) {
    // printf("error: the data pass two pages:addr=0x%x, len=%d!\n", addr, len);
    assert(0);
  }
  else {
    paddr_t paddr = page_translate(addr, false);
    return paddr_read(paddr, len);
  }
  // return paddr_read(addr, len);
}
```

vaddr_write(vaddr_t addr, int len) 函数实现了写地址时的虚拟地址转换。

```
  void vaddr_write(vaddr_t addr, int len, uint32_t data) {
  if(PTE_ADDR(addr) != PTE_ADDR(addr+len-1)) {
    // printf("error: the data pass two pages:addr=0x%x, len=%d!\n", addr, len);
    assert(0);
    }
  }
  else {
    paddr_t paddr = page_translate(addr, true);
    paddr_write(paddr, len, data);
  }
  // paddr_write(addr, len, data);
}
```

其中，如果存在跨页读写的现象，暂且不做处理，直接结束程序。这里的 page_translate() 函数是实验指导书要求实现的页面地址转换函数，接下来我们就需要对这个函数进行编写。

```
paddr_t page_translate(vaddr_t addr, bool iswrite) {
  CR0 cr0 = (CR0)cpu.CR0;
  if(cr0.paging && cr0.protect_enable) {
    CR3 crs = (CR3)cpu.CR3;

    PDE *pgdirs = (PDE*)PTE_ADDR(crs.val);
    PDE pde = (PDE)paddr_read((uint32_t)(pgdirs + PDX(addr)), 4);

    PTE *ptable = (PTE*)PTE_ADDR(pde.val);
    PTE pte = (PTE)paddr_read((uint32_t)(ptable + PTX(addr)), 4);
    //printf("hhahah%x, jhhh%x\n", pte.present, addr);
```

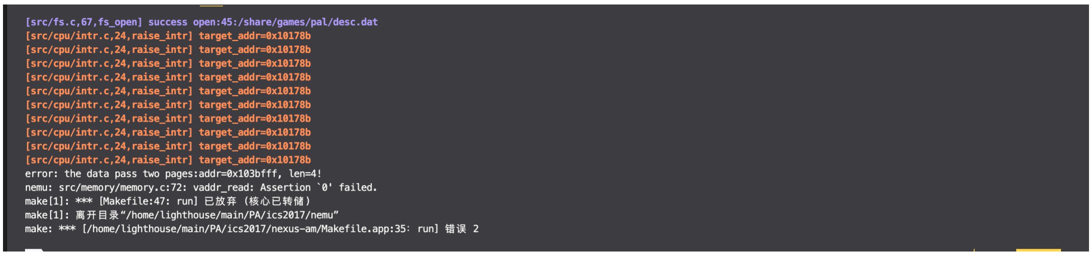
```
    Assert(pte.present, "addr=0x%x", addr);

    pde.accessed=1;
    pte.accessed=1;
    if(iswrite) {
      pte.dirty=1;
    }
    paddr_t paddr = PTE_ADDR(pte.val) | OFF(addr);
    // printf("vaddr=0x%x, paddr=0x%x\n", addr, paddr);
    return paddr;
  }
    return addr;
}
```

在 page_translate(vaddr_t addr, bool iswrite) 函数当中，参数 addr 代表待处理页面的地址，iswrite 是读或写的标志位，若需要进行的是读操作，该参数就被设置为 false，若需要进行写操作，就被设置为 true。

这个函数需要依次判断页目录、页表是否存在，并根据读写情况对页面进行脏位标记。如果页面不存在，需要呈现错误信息并结束程序。

运行程序，可以发现，这时候的虚拟地址和物理地址是相同的，我们的初步分页算法实现成功。

但这个算法只能在单页面内有效，如果出现了跨页的情况，就会按我们在 vaddr_read() 和 vaddr_write() 函数中所设定的一样，结束程序。



为了解决跨页问题，我们还需要进一步修改 nemu/src/memory/memory.c 中的这两个读写函数。

```
uint32_t vaddr_read(vaddr_t addr, int len) {
  if(PTE_ADDR(addr) != PTE_ADDR(addr + len -1)) {
    // printf("error: the data pass two pages:addr=0x%x, len=%d!\n", addr, len);
    // assert(0);
    int num1 = 0x1000 - OFF(addr);
    int num2 = len - num1;
    paddr_t paddr1 = page_translate(addr, false);
    paddr_t paddr2 = page_translate(addr + num1, false);

```

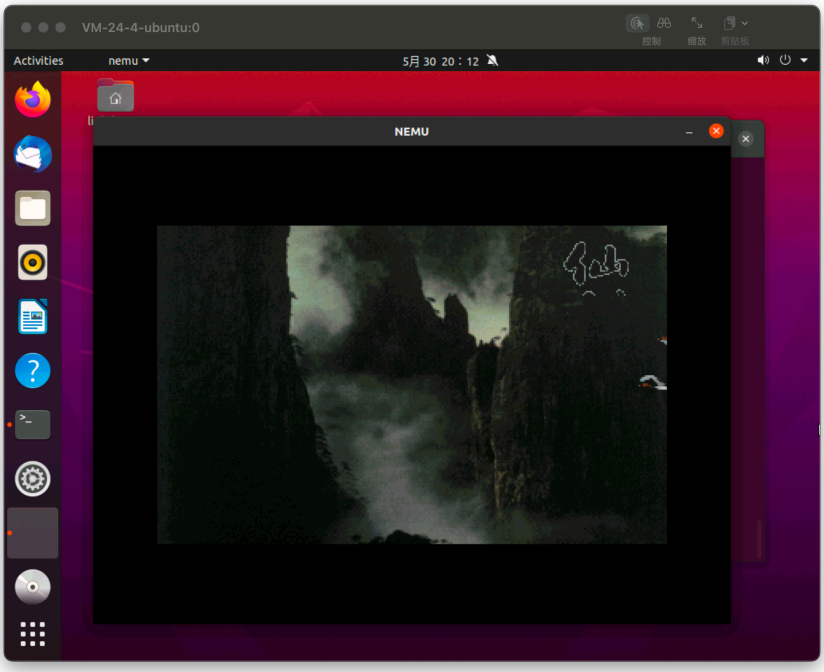
```
    uint32_t low = paddr_read(paddr1, num1);
    uint32_t high = paddr_read(paddr2, num2);

    uint32_t result = high << (num1 * 8) | low;
    return result;
  }
  else {
    paddr_t paddr = page_translate(addr, false);
    return paddr_read(paddr, len);
  }
  // return paddr_read(addr, len);
}
```

如果出现了跨页的情况，就分高低页进行读取，用两个 num 局部变量记录高低页的字节数，paddr 记录高低页对应物理地址，再进行整合读取。

跨页写入的情况和跨页读取类似。

```
void vaddr_write(vaddr_t addr, int len, uint32_t data) {
  if(PTE_ADDR(addr) != PTE_ADDR(addr+len-1)) {
    // printf("error: the data pass two pages:addr=0x%x, len=%d!\n", addr, len);
    // assert(0);
    if(PTE_ADDR(addr) != PTE_ADDR(addr + len -1)) {
      int num1 = 0x1000-OFF(addr);
      int num2 = len -num1;
      paddr_t paddr1 = page_translate(addr, true);
      paddr_t paddr2 = page_translate(addr + num1, true);

      uint32_t low = data & (~0u >> ((4 - num1) << 3));
      uint32_t high = data >> ((4 - num2) << 3);

      paddr_write(paddr1, num1, low);
      paddr_write(paddr2, num2, high);
      return;
    }
  }
  else {
    paddr_t paddr = page_translate(addr, true);
    paddr_write(paddr, len, data);
  }
  // paddr_write(addr, len, data);
}
```

完善这两个函数之后，跨页问题就得到了解决，程序可以正确运行。



（三） 让用户程序运行在分页机制上

实现了分页机制之后，我们回归到最初的目的：让用户程序能够分时运行。所以本阶段这部分内容需要将用户程序也运行在我们已经实现好的分页机制上。先按指导书修改 navy¬-apps/Makefile.compile，把链接地址改成 0x8048000：

```
ifeq ($(LINK), dynamic)
  CFLAGS += -fPIE
  CXXFLAGS += -fPIE
  LDFLAGS += -fpie -shared
else
  LDFLAGS += -Ttext 0x8048000
endif
```

同时也相对应地修改 nanos-lite/src/loader.c 当中对 DEFAULT_ENTRY 的宏定义：

```
#define DEFAULT_ENTRY ((void *)0x8048000)
```

运行 dummy 报缺页错误，这是因为我们尚未实现页的分配和映射，所以运行会有问题。

于是，在 nexus-am/am/arch/x86-nemu/src/pte.c 当中实现 _map() 函数。这个函数的具体功能是将虚拟地址空间 p 中的虚拟地址 va 映射到物理地址 pa，通过 p->ptr 可以获取页目录的基地址；如果映射过程中发现需要申请新的页，就调用 palloc_f() 函数来完成。

```
void _map(_Protect *p, void *va, void *pa) {
```

```
  if(OFF(va) || OFF(pa)) {
    // printf("page not aligned\n");
    return;
  }

  PDE *dir = (PDE*) p -> ptr;
    PTE *table = NULL;
    PDE *pde = dir + PDX(va);
    if(!(*pde & PTE_P)) {
       table = (PTE*) (palloc_f());
       *pde = (uintptr_t) table | PTE_P;
    }
    table = (PTE*) PTE_ADDR(*pde);
    PTE *pte = table + PTX(va);
    *pte = (uintptr_t) pa | PTE_P;
}
```

完成了页面的分配和映射，还需要让 loader 也按页加载。在 nanos-lite/src/loader.c 中对 loader 函数进行修改

```
uintptr_t loader(_Protect *as, const char *filename) {
  // TODO();
  // ramdisk_read(DEFAULT_ENTRY, 0, RAMDISK_SIZE);
  int fd = fs_open(filename, 0, 0);
  Log("filename=%s,fd=%d",filename,fd);
  // fs_read(fd, DEFAULT_ENTRY, fs_filesz(fd));
  int size = fs_filesz(fd);
  int ppnum = size / PGSIZE;
  if(size % PGSIZE != 0) {
    ppnum++;
  }
  void *pa = NULL;
  void *va = DEFAULT_ENTRY;
  for(int i = 0; i < ppnum; i++) {
    pa = new_page();
    _map(as, va, pa);
    fs_read(fd, pa, PGSIZE);
    va += PGSIZE;
  }

  fs_close(fd);
  return (uintptr_t)DEFAULT_ENTRY;
}
```

此时 dummy 已经可以正确运行，但仙剑奇侠传还是有一些需要进一步实现和完善的地方。为了让仙剑奇侠传在分页机制上运行，我们需要完成堆内存的映射，实现 nanos-lite/src/mm.c 当中的 mm_brk() 函数。在 PA3 的实现中，我们直接让这个函数返回 0，表示用户程序的堆区大小修改总是成功。但实现分页之后，用户程序运行在虚拟地址空间之上，于是我们在这个函数中完成了把新申请的堆区映射到虚拟地址空间中的工作

```
int mm_brk(uint32_t new_brk) {
  if(current -> cur_brk == 0) {
    current -> cur_brk = current -> max_brk = new_brk;
  }
  else {
```

```
    if(new_brk > current -> max_brk) {
      uint32_t first = PGROUNDUP(current -> max_brk);
      uint32_t end = PGROUNDDOWN(new_brk);
      if((new_brk & 0xfff) == 0) {
        end -= PGSIZE;
      }
      for(uint32_t va = first; va <= end; va += PGSIZE) {
        void *pa = new_page();
        _map(&(current -> as), (void*)va, pa);
      }
      current -> max_brk = new_brk;
    }
    current -> cur_brk = new_brk;
  }
  return 0;
}
```

修改 nanos-lite/src/syscall.c 中与之对应的 sys_brk() 函数

```
int sys_brk(int addr) {
  extern int mm_brk(uint32_t new_brk);
  return mm_brk(addr);
}
```

完成后，仙剑奇侠传已经可以正常运行，PA4 第一阶段到此结束。