

PA4 实验报告

- 姓名：陈睿颖
- 学号：2013544
- 专业：计算机科学与技术

目录

1. 实验内容	2
2. 阶段一	2
2.1 实现分页机制	2
2.2 让用户程序运行在分页机制上	5
在 PTE 中实现_map()	6
在分页机制上运行仙剑奇侠传	8
3. 阶段二	11
3.1 实现内核自陷	11
3.2 实现上下文切换	12
3.3 分时运行仙剑奇侠传和 hello 程序	15
4. 阶段三	16
4.1 添加时钟中断	16
5. 编写不朽的传奇	18
5.1 展示计算机系统	18
6. 思考题	20
一些问题	20
空指针真的是"空"的吗?	21
内核映射的作用	22
灾难性的后果(这个问题有点难度)	23
必答题	23
万变之宗 - 重新审视计算机	24
7. Bug Log	24
缺少 klibxxx.a 文件	25
0x0010135f 处缺少指令	25
实现完成按 F12 报错 BAD TRAP	27

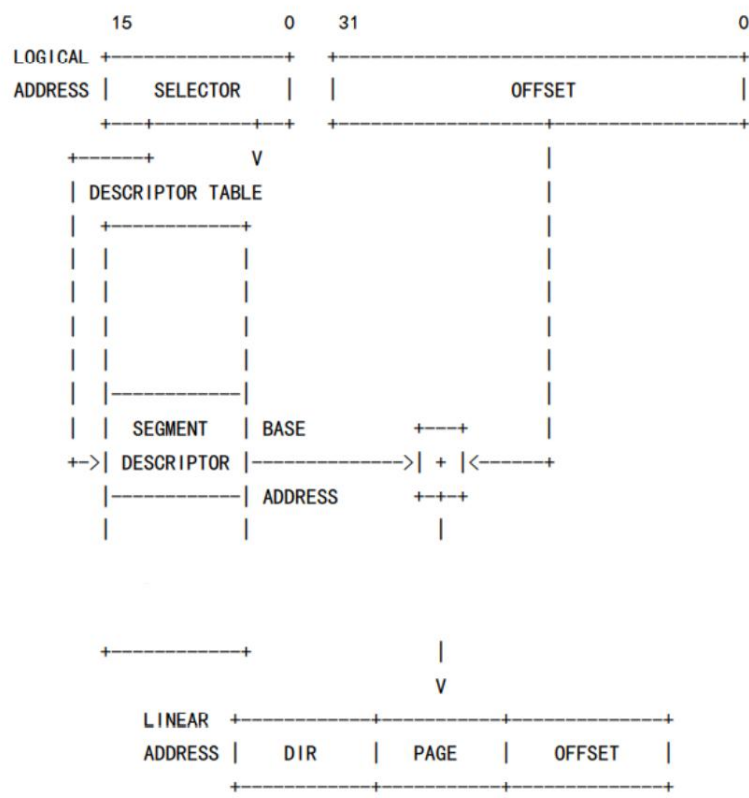
1. 实验内容

- 阶段 1: 实现分页机制
- 阶段 2: 实现上下文切换
- 最后阶段: 实现真正的分时多任务

2. 阶段一

2.1 实现分页机制

i386 手册中的分段机制如下：



i386 架构中的分页机制使用多级页表来管理内存：

1. 物理内存被划分为固定大小的页面（**Page**），通常为 4KB。
2. 虚拟地址空间也被划分为相同大小的页面。
3. 分页机制使用页表来建立虚拟地址和物理地址之间的映射关系。
4. 页面目录（**Page Directory**）是一个特殊的数据结构，用于存储指向页表的指针。它由多个页面目录项（**Page Directory Entry**）组成。
5. 页表（**Page Table**）是另一个数据结构，用于存储虚拟地址到物理地址的映射。

每个页表包含多个页面表项 (Page Table Entry)。

6. 页面目录和页表位于内存中的固定位置，可以通过控制寄存器 (CR3 寄存器) 来获得其基地址。
7. 当程序引用虚拟地址时，处理器会将虚拟地址拆分为目录索引、表索引和页内偏移。
8. 处理器使用目录索引查找页面目录项，然后使用表索引查找对应的页表项。
9. 页表项存储了虚拟地址与物理地址的映射关系。通过将页表项中的物理地址与页内偏移相结合，可以获得最终的物理地址。
10. 如果虚拟地址没有有效的映射，或者页表项中标记为无效 (例如，被标记为不存在或禁止访问)，则会触发页面错误 (Page Fault)。

在 `nanos-lite/src/main.c` 中定义宏 `HAS_PTE`:

```
C++
#define HAS_PTE
```

编写 `page_translate()` 函数:

```
C++
paddr_t page_translate(vaddr_t addr, int rw) {
    if (!cpu.cr0.protect_enable || !cpu.cr0.paging)
        return addr;

    PDE* pd = (PDE*)(cpu.cr3.val & 0xfffff000);
    int pd_index = (addr >> 22) & 0x3ff;
    PTE* pt;

    PDE pde;
    pde.val = paddr_read((paddr_t)&pd[pd_index], 4);
    if (!pde.present)
        Assert(0, "pde present bit is 0!");

    pt = (PTE*)(pde.val & ~0xfff);
    pde.accessed = 1;
    paddr_write((paddr_t)&pd[pd_index], 4, pde.val);

    int pt_index = (addr >> 12) & 0x3ff;
    PTE pte;
    pte.val = paddr_read((paddr_t)&pt[pt_index], 4);
```

```

if (!pte.present)
    Assert(0, "pte present bit is 0!");

addr = (pte.val & ~0xffff) | (addr & 0xffff);
pte.accessed = 1;
if (rw)
    pte.dirty = 1;
paddr_write((paddr_t)&pt[pt_index], 4, pte.val);

return (paddr_t)addr;
}

```

1. 首先，检查是否进入保护模式且开启了分页机制。如果不满足条件，直接返回虚拟地址，不进行转换。
2. 从 CR3 寄存器获取页目录表的地址，并根据虚拟地址的高 10 位计算目录表的索引。
3. 读取目录表项内容，并检查目录项的 **present** 位是否为 1，表示该目录表项存在。
4. 获取二级目录指针，并将目录项的 **accessed** 位设置为 1，表示已访问。然后将目录项写回目录表。
5. 根据虚拟地址的中间 10 位计算二级页表的索引。
6. 读取二级页表项内容，并检查页表项的 **present** 位是否为 1，表示该页表项存在。
7. 将页表项的高 20 位和虚拟地址的低 12 位合并，得到物理地址。
8. 设置页表项的 **accessed** 位为 1，表示已访问。如果是写操作，则设置页表项的 **dirty** 位为 1，表示已修改。
9. 将页表项写回页表。
10. 返回转换后的物理地址。

这样，通过对目录表和页表的查找和更新，`page_translate()` 函数能够将虚拟地址转换为物理地址，并在转换过程中处理访问和修改位，实现了分页机制下的地址转换功能。

在 `nemu/src/memory/memory.c` 中修改 `vaddr_read()` 和 `vaddr_write()` 函数：

```

C++
uint32_t vaddr_read(vaddr_t addr, int len) {

```

```

    if (((addr) + (len) - 1) & ~PAGE_MASK) != ((addr) &
~PAGE_MASK)) {
        // 数据跨页边界
        int i;
        uint32_t data = 0;
        for (i = 0; i < len; i++) {
            paddr_t paddr = page_translate(addr + i, 0);
            data += paddr_read(paddr, 1) << (i * 8);
        }
        return data;
    } else {
        paddr_t paddr = page_translate(addr, 0);
        return paddr_read(paddr, len);
    }
}

void vaddr_write(vaddr_t addr, int len, uint32_t data) {
    if (((addr) + (len) - 1) & ~PAGE_MASK) != ((addr) &
~PAGE_MASK)) {
        // 数据跨页边界
        int i;
        for (i = 0; i < len; i++) {
            paddr_t paddr = page_translate(addr + i, 1);
            paddr_write(paddr, 1, data >> (i * 8));
        }
    } else {
        paddr_t paddr = page_translate(addr, 1);
        paddr_write(paddr, len, data);
    }
}
}

```

2.2 让用户程序运行在分页机制上

先需要将 `navy-apps/Makefile.compile` 中的链接地址 `-Ttext` 参数改为 `0x8048000`:

```

Makefile
#....
ifeq ($(LINK), dynamic)
    CFLAGS    += -fPIE

```

```

CXXFLAGS += -fPIE
LDFLAGS += -fpie -shared
else
    LDFLAGS += -Ttext 0x8048000
endif
#...

```

这是为了避免用户程序的虚拟地址空间与内核相互重叠, 从而产生非预期的错误..同样的, `nanos-lite/src/loader.c` 中的 `DEFAULT_ENTRY` 也需要作相应的修改。

```

C++
#define DEFAULT_ENTRY ((void *)0x8048000)

```

这时, "虚拟地址作为物理地址的抽象"这一好处已经体现出来了: 原则上用户程序可以运行在任意的虚拟地址, 不受物理内存容量的限制. 我们让用户程序的代码从 `0x8048000` 附近开始, 这个地址已经超过了物理地址的最大值(NEMU 提供的物理内存是 128MB), 但分页机制保证了程序能够正确运行. 这样, 链接器和程序都不需要关心程序运行时刻具体使用哪一段物理地址, 它们只要使用虚拟地址就可以了, 而虚拟地址和物理地址之间的映射则全部交给操作系统的 MM 来管理.

然后, 我们让 Nanos-lite 通过 `load_prog()` 函数(在 `nanos-lite/src/proc.c` 中定义)来进行用户程序的加载:

```

Plain Text
--- nanos-lite/src/main.c
+++ nanos-lite/src/main.c
@@ -33,2 +33,1 @@
-  uintptr_t entry = loader(NULL, "/bin/pal");
-  ((void (*)(void))entry)();
+  load_prog("/bin/dummy");

```

在 PTE 中实现 `_map()`

`_map()` 函数(在 `nexus-am/am/arch/x86-nemu/src/pte.c` 中定义). 它的函数原型如下

```

C++
void _map(_Protect *p, void *va, void *pa);

```

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. 若在映射过程中发现需要申请新的页表, 可以通过回调函数 `palloc_f()` 向 Nanos-lite 获取一页空闲的物理页.

从 `loader()` 返回后, `load_prog()` 会调用 `_switch()` 函数(在 `nexus-am/am/arch/x86-nemu/src/pte.c` 中定义), 切换到刚才为用户程序创建的地址空间. 最后跳转到用户程序的入口, 此时用户程序已经完全运行在分页机制上了。

需要更改 `loader()` 函数: 获取用户程序的大小之后, 以页为单位进行加载:

- 申请一页空闲的物理页
- 把这一物理页映射到用户程序的虚拟地址空间中
- 从文件中读入一页的内容到这一物理页上

```
C++
uintptr_t loader(_Protect *as, const char *filename) {
    int fd = fs_open(filename, 0, 0);
    int bytes = fs_filesz(fd);
    int n = bytes / PGSIZE;
    int m = bytes % PGSIZE;
    int i;
    void *pa;

    for (i = 0; i < n; i++) {
        pa = new_page();
        _map(as, DEFAULT_ENTRY + i * PGSIZE, pa);
        fs_read(fd, pa, PGSIZE);
    }

    pa = new_page();
    _map(as, DEFAULT_ENTRY + i * PGSIZE, pa);
    fs_read(fd, pa, m);

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

编写 `_map()` 函数:

```
C++
void _map(_Protect *p, void *va, void *pa) {
    PDE *pdir = p->ptr;
    uint32_t pd_index = ((uint32_t)va) >> 22 & 0x3ff;
```

```

PTE *pt = NULL;
uint32_t pt_index = ((uint32_t)va) >> 12 & 0x3ff;

if (pdir[pd_index] & PTE_P) {
    pt = (PTE *)(pdir[pd_index] & ~0xfff);
} else {
    pt = (PTE *)palloc_f();
    pdir[pd_index] = ((uint32_t)pt & ~0xfff) | PTE_P;
}

pt[pt_index] = ((uint32_t)pa & ~0xfff) | PTE_P;
}

```

首先，获取当前保护区 `_Protect` 的页目录表指针 `pdir`。然后根据虚拟地址 `va`，计算出页目录表的索引 `pd_index` 和二级页表的索引 `pt_index`。

接下来，检查页目录表的相应项是否有效。如果有效（即存在），则获取对应的二级页表指针 `pt`。如果无效，说明需要建立页目录项和二级页表的映射关系。这时，通过 `palloc_f()` 分配一个新的页面，并将其地址保存在页目录表的相应项中。

无论之前的情况如何，我们都获得了二级页表的地址 `pt`。

最后，将物理地址 `pa` 和页表标志 `PTE_P` 合并，并存储在二级页表的相应项中，建立虚拟地址 `va` 到物理地址 `pa` 的映射关系。

运行如下：

```

[src/monitor/monitor.c,65,load_img] The image is /home/user/ics2017/nanos-lite/build/nanos-lite-x86-nemu.bin
Welcome to NEMU!
[src/monitor/monitor.c,30,welcome] Build time: 19:22:54, May 22 2023
For help, type "help"
(nemu) c
[src/mm.c,24,init_mm] free physical pages starting from 0x2cdf000
[src/main.c,20,main] 'Hello World!' from Nanos-lite
[src/main.c,21,main] Build time: 19:30:46, May 22 2023
[src/ramdisk.c,26,init_ramdisk] ramdisk info: start = 0x1029e0, end = 0x2c99f38, size = 45708632 bytes
[src/main.c,28,main] Initializing interrupt/exception handler...
[src/fs.c,43,fs_open] Pathname: /bin/dummy
nemu: HIT GOOD TRAP at eip = 0x00100032

```

成功显示 GOOD TRAP 的信息。

在分页机制上运行仙剑奇侠传

现在用户程序运行在虚拟地址空间之上，我们还需要在 `mm_brk()` 中把新申请的堆区映射到虚拟地址空间中。

```

C++
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) {
            // TODO: map memory region [current->max_brk, new_brk)
            // into address space current->as
            uint32_t brk = K4(current->max_brk);
            while(brk<new_brk)
            {
                _map(&current->as, (void*)brk, new_page()); //new_page 获得的
物理页地址一定是按 4K 对齐的,
                                                                    //按照我的_map 函数
的实现方式, brk 也一定要按 4K 对齐!
                brk += PGSIZE;
            }

            current->max_brk = new_brk;
        }

        current->cur_brk = new_brk;
    }

    return 0;
}

```

在 `nanos-lite/src/mm.c` 中完成 `mm_brk()` 的 TODO 编写:

```

C++
#define K4(va) (((uint32_t)(va) + 0xfff) & ~0xfff) // 使得虚拟地址
以 4K 对齐, 确保按照物理页的对齐方式进行映射

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 brk = K4(current->max_brk);
            while (brk < new_brk) {
                _map(&current->as, (void*)brk, new_page());
                brk += PGSIZE;
            }
            current->max_brk = new_brk;
        }
    }
}

```

```

    current->cur_brk = new_brk;
}
return 0;
}

```

修改 sys_brk:

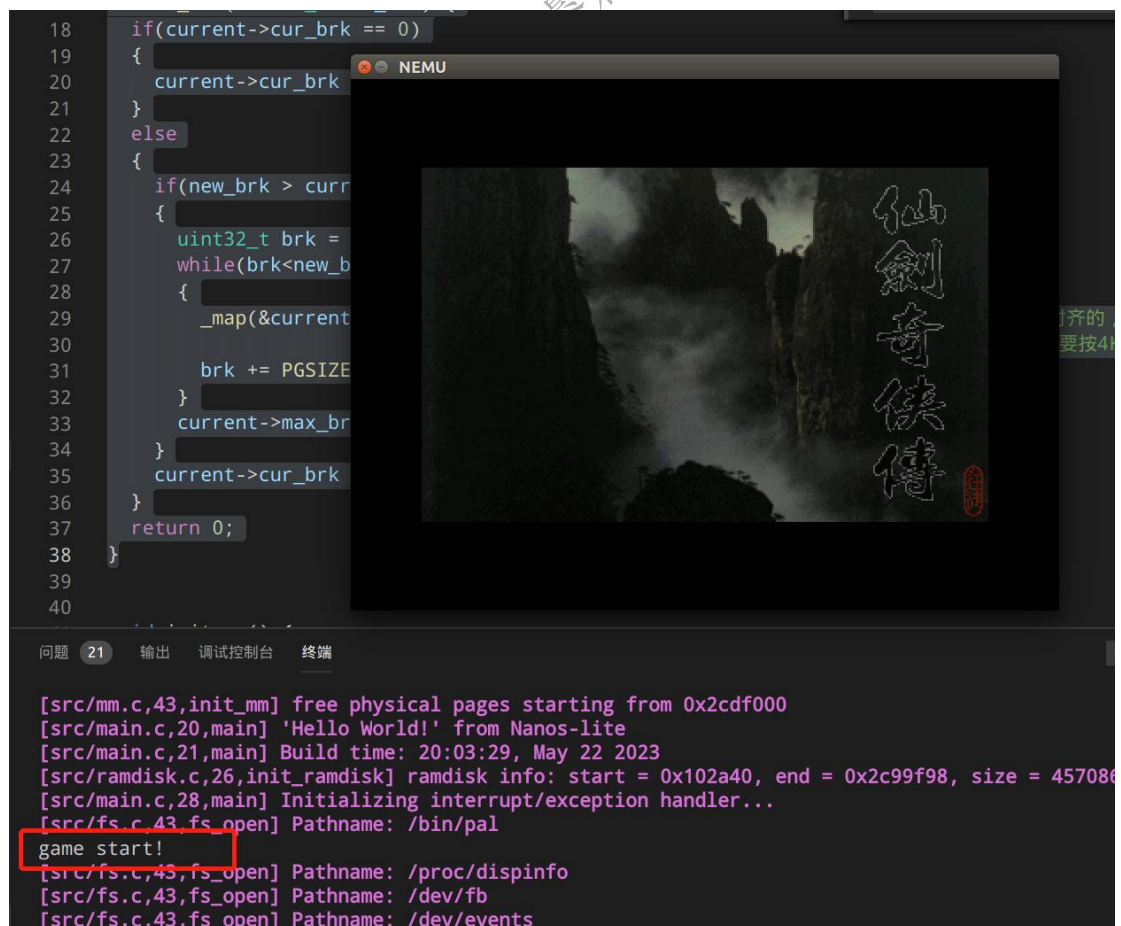
```

C++
extern int mm_brk(uint32_t new_brk);
static inline _RegSet* sys_brk(_RegSet *r){
//  SYSCALL_ARG1(r) = 0;//总是返回 0
//  //r->eax=0;
//  return NULL;

    SYSCALL_ARG1(r) = mm_brk(SYSCALL_ARG2(r));
    return NULL;
}

```

make update 后运行:



```

18  if(current->cur_brk == 0)
19  {
20      current->cur_brk
21  }
22  else
23  {
24      if(new_brk > curr
25      {
26          uint32_t brk =
27          while(brk<new_b
28          {
29              _map(&current
30
31              brk += PGSIZE
32          }
33          current->max_br
34      }
35      current->cur_brk
36  }
37      return 0;
38  }
39
40

```

```

[src/mm.c,43,init_mm] free physical pages starting from 0x2cdf000
[src/main.c,20,main] 'Hello World!' from Nanos-lite
[src/main.c,21,main] Build time: 20:03:29, May 22 2023
[src/ramdisk.c,26,init_ramdisk] ramdisk info: start = 0x102a40, end = 0x2c99f98, size = 457086
[src/main.c,28,main] Initializing interrupt/exception handler...
[src/fs.c,43,fs_open] Pathname: /bin/pal
game start!
[src/fs.c,43,fs_open] Pathname: /proc/dispinfo
[src/fs.c,43,fs_open] Pathname: /dev/fb
[src/fs.c,43,fs_open] Pathname: /dev/events

```

成功运行仙剑奇侠传!

3. 阶段二

3.1 实现内核自陷

修改 Nanos-lite 的如下代码:

```
C++
/*main.c*/
load_prog("/bin/pal");

_trap();

panic("Should not reach here");
/*proc.c*/

// TODO: remove the following three lines after you have
implemented _umake()
// _switch(&pcb[i].as);
// current = &pcb[i];
// ((void (*)(void))entry)();
```

并在 ASYE 添加相应的代码, 使得 `irq_handle()` 可以识别内核自陷并包装成 `_EVENT_TRAP` 事件, Nanos-lite 接收到 `_EVENT_TRAP` 之后可以输出一句话, 然后直接返回即可, 因为真正的上下文切换还需要正确实现 `_umake()` 之后才能实现:

nexus-am/am/arch/x86-nemu/src/asye.c:

```
C++
void _trap() {
    asm volatile("int $0x81");
}

case 0x81: ev.event = _EVENT_TRAP; break;

idt[0x81] = GATE(STS_TG32, KSEL(SEG_KCODE), vecself, DPL_USER);
```

Nanos-lite 触发了 `main()` 函数中最后的 panic:

```
[src/main.c,20,main] 'Hello World!' from Nanos-lite
[src/main.c,21,main] Build time: 16:32:38, May 24 2023
[src/ramdisk.c,26,init_ramdisk] ramdisk info: start = 0x102a60, end = 0x2c99fb8, size = 45708632 bytes
[src/main.c,28,main] Initializing interrupt/exception handler...
[src/fs.c,43,fs_open] Pathname: /bin/pal
[src/irq.c,8,do_event] system panic: Unhandled event ID = 7
nemu: HIT BAD TRAP at eip = 0x00100032
(nemu) □
```

3.2 实现上下文切换

- PTE 的 `_umake()` 函数

构造一个初始的用户进程陷阱帧，以便从内核态切换到用户态时，能够正确传递参数、设置寄存器和标志位，使用户进程能够正确运行。

```
C++
_RegSet *_umake(_Protect *p, _Area ustack, _Area kstack, void
*entry, char *const argv[], char *const envp[]) {
    extern void *memcpy(void *,const void*,int);
    //设置_start()的栈帧 int arg1=0;
    char *arg2=NULL;
    memcpy((void*)ustack.end-4,(void*)arg2,4);
    memcpy((void*)ustack.end-8,(void*)arg2,4);
    memcpy((void*)ustack.end-12,(void*)arg1,4);
    memcpy((void*)ustack.end-16,(void*)arg1,4);
    //trapframe
    _RegSet tf;
    tf.eflags=0x02;
    tf.cs=8;
    tf.eip=(uintptr_t)entry;//返回地址为 entryvoid
    *ptf=(void*)(ustack.end-16-typeof(_RegSet)); //tf 的基址
    memcpy(ptf,(void*)&tf,sizeof(_RegSet)); //把 tf 压栈 return
    (_RegSet*)ptf;
}
```

1. 在用户栈中创建陷阱帧，以备用户进程从内核态切换到用户态时使用。
2. 使用指定的参数填充陷阱帧的各个字段，包括寄存器值、标志位等。
3. 返回陷阱帧指针（`_RegSet*` 类型），供进程切换时使用

- Nanos-lite 的 `schedule()` 函数

```
C++
```

```

_RegSet* schedule(_RegSet *prev) {
    if(current!=NULL){//current:当前进程的 PCB 指针
        current->tf=prev;//保存 tf
    }
    current=&pcb[0];//切换到第一个用户进程
    _switch(&current->as);//切换虚拟地址空间
    return current->tf;
}

```

1. 接收一个陷阱帧指针 `prev`，表示当前运行的进程的陷阱帧。
2. 如果当前进程 `current` 不为空（即之前已经有进程运行过），则将当前进程的陷阱帧指针更新为 `prev`，即保存当前进程的上下文。
3. 将 `current` 指针指向第一个用户进程的 PCB（Process Control Block），即 `pcb[0]`。
4. 调用 `_switch` 函数，切换到选定进程的地址空间。
5. 返回选定进程的陷阱帧指针，以便从用户态切换到内核态时使用。

实现了一个非常简单的进程调度逻辑，每次调用 `schedule` 函数时切换到第一个用户进程，并保存当前进程的上下文。通过这种方式，可以实现基本的进程切换和调度功能。

- Nanos-lite 收到 `_EVENT_TRAP` 事件后，调用 `schedule()` 并返回其现场

```

C++
static _RegSet* do_event(_Event e, _RegSet* r) {
    switch (e.event) {
        case _EVENT_SYSCALL:
            do_syscall(r);
            break;
        case(_EVENT_TRAP):
            printf("self-trapped event!\n");
            return schedule(r);//返回新的进程的 tf
        default: panic("Unhandled event ID = %d", e.event);
    }

    return NULL;
}

```

- 修改 ASYE 中 `asm_trap()` 的实现，使得从 `irq_handle()` 返回后，先将栈顶指针切换到新进程的陷阱帧，然后才根据陷阱帧的内容恢复现场，从而完成上下文切换的本质

操作

```
C++
asm_trap:
    pushal

    pushl %esp
    call irq_handle

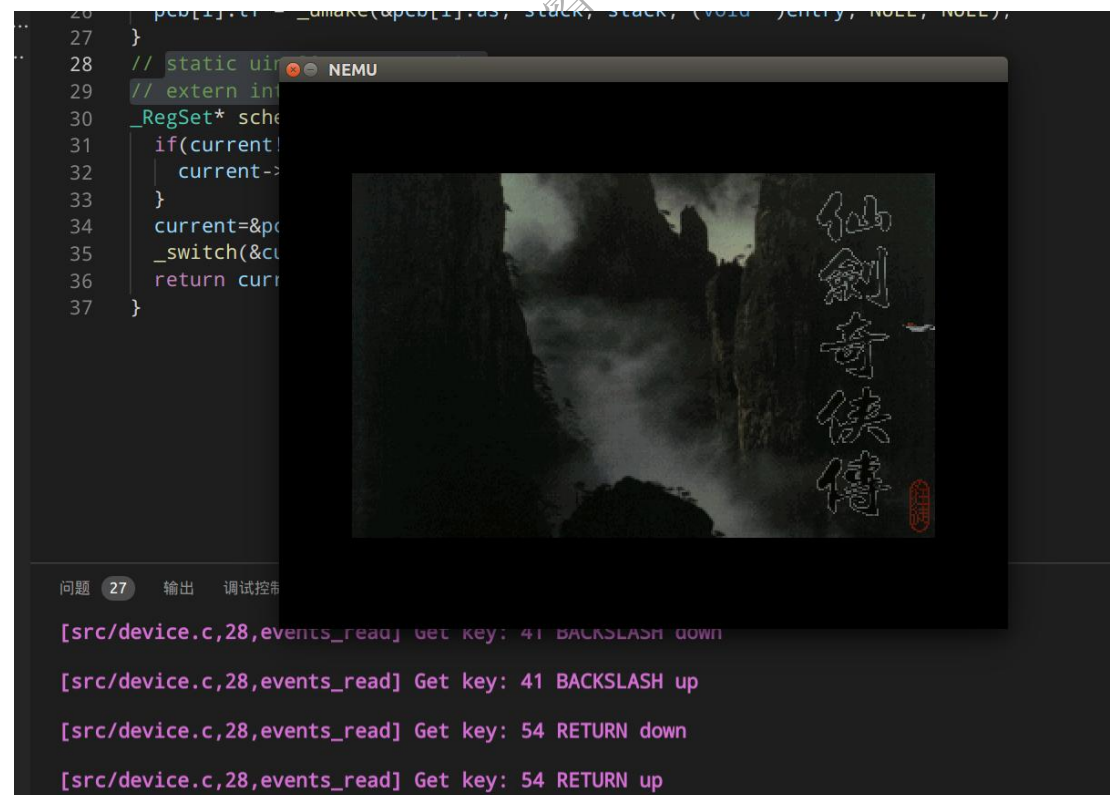
    #addl $4, %esp
    movl %eax,%esp

    popal
    addl $8, %esp

    iret
```

asm_trap 中，中断处理函数 irq_handle 将 tf 的位置作为返回值存放在 eax 中，我们现在将它赋给 esp，这样就将栈顶切换到新进程的 tf 了。

实现成功后，Nanos-lite 就可以通过内核自陷触发上下文切换的方式运行仙剑奇侠传了：



3.3 分时运行仙剑奇侠传和 hello 程序

修改调度的代码, 让 `schedule()` 轮流返回仙剑奇侠传和 hello 的现场:

```
Plaintext
current = (current == &pcb[0] ? &pcb[1] : &pcb[0]);
```

我们可以修改 `schedule()` 的代码, 来调整仙剑奇侠传和 hello 程序调度的频率比例, 使得仙剑奇侠传调度若干次, 才让 hello 程序调度 1 次. 这是因为 hello 程序做的事情只是不断地输出字符串, 我们只需要让 hello 程序偶尔进行输出, 以确认它还在运行就可以了. 具体方法是当仙剑 (进程 1) 被调度 1000 次后, 切换成 hello (进程 2) 执行一次, 通过一个静态的计数器 `num` 来记录次数。

```
C++
//进程调度
_RegSet* schedule(_RegSet *prev) {
    // return NULL;
    if(current!=NULL){//current:当前进程的 PCB 指针
        current->tf=prev;//保存 tf
    }
    // current=(current==&pcb[0]?&pcb[1]:&pcb[0]);
    // Log("ptr=0x%x\n", (uintptr_t)current->as.ptr);
    else{
        current=&pcb[0];//初始进程为 0 号进程
    }
    static int num=0;
    static const int freq=1000;
    if(current==&pcb[0]){
        num++;
    }
    else{
        current=&pcb[0];
    }
    if(num==freq){//如果到达 1000 次, 则切换成进程 1
        current=&pcb[1];
        num=0;
    }
    _switch(&current->as);//切换虚拟地址空间
    return current->tf;
}
```

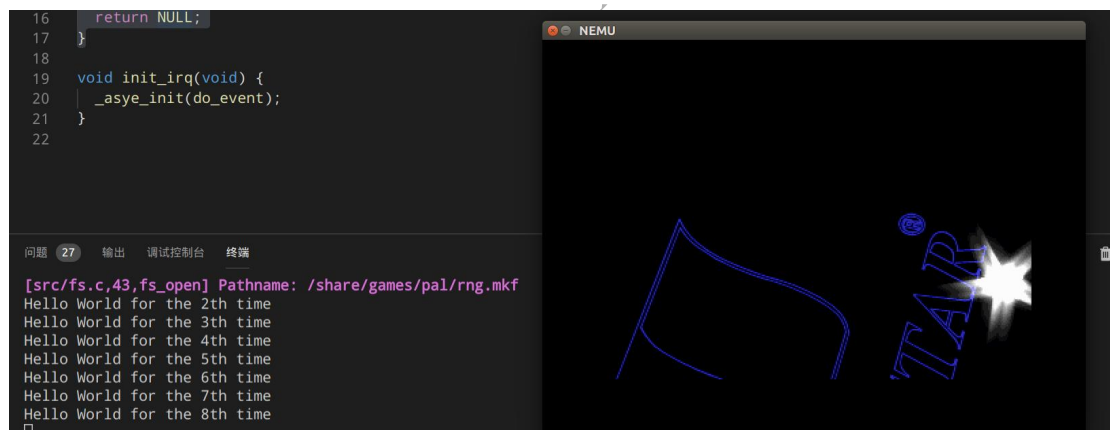
最后, 我们还需要选择一个时机来触发进程调度. 目前比较合适的时机就是处理系统调

用之后: 修改 `do_event()` 的代码, 在处理完系统调用之后, 调用 `schedule()` 函数并返回其现场:

```
C++
static _RegSet* do_event(_Event e, _RegSet* r) {
    switch (e.event) {
        case _EVENT_SYSCALL:
            do_syscall(r);
            return schedule(r); // 返回新的进程的 tf
            break;
        case(_EVENT_TRAP):
            printf("self-trapped event!\n");
            return schedule(r); // 返回新的进程的 tf
        default: panic("Unhandled event ID = %d", e.event);
    }

    return NULL;
}
```

运行结果:



可以看到仙剑奇侠传一边运行的同时, hello 程序也会一边输出

4. 阶段三

4.1 添加时钟中断

在 NEMU 中, 我们只需要添加时钟中断这一种中断就可以了. 由于只有一种中断, 我们也不需要通过中断控制器进行中断的管理, 直接让时钟中断连接到 CPU 的 INTR 引脚即可, 我们也约定时钟中断的中断号是 32. 时钟中断通过 `nemu/src/device/timer.c` 中的 `timer_intr()` 触发, 每 10ms 触发一次. 触发后, 会调用 `dev_raise_intr()` 函数 (在 `nemu/src/cpu/intr.c` 中定义):

- 在 `cpu` 结构体中添加一个 `bool` 成员 `INTR`:

```
C++
bool INTR;
```

- 在 `dev_raise_intr()` 中将 `INTR` 引脚设置为高电平:

```
C++
void dev_raise_intr() {
    cpu.INTR = true;
}
```

- 在 `exec_wrapper()` 的末尾添加轮询 `INTR` 引脚的代码, 每次执行完一条指令就查看是否有硬件中断到来:

```
C++
#define TIMER_IRQ 32
if (cpu.INTR & cpu.eflags.IF) {
    cpu.INTR = false;
    raise_intr(TIMER_IRQ, cpu.eip);
    update_eip();
}
```

- 修改 `raise_intr()` 中的代码, 在保存 `EFLAGS` 寄存器后, 将其 `IF` 位置为 `0`, 让处理器进入关中断状态:

```
C++
cpu.eflags.IF = 0; // 关闭 IF 位
```

在软件上:

- 在 `ASYS` 中添加时钟中断的支持, 将时钟中断打包成 `_EVENT_IRQ_TIME` 事件.

```
C++
case 0x32: ev.event = _EVENT_IRQ_TIME; break;
```

- `Nanos-lite` 收到 `_EVENT_IRQ_TIME` 事件之后, 直接调用 `schedule()` 进行进程调度, 同时也可以去掉系统调用之后调用的 `schedule()` 代码了.

```
C++
case _EVENT_SYSCALL:
    do_syscall(r);
    // return schedule(r); // 返回新的进程的 tf
```

```

        break;
    case(_EVENT_TRAP):
        printf("self-trapped event!\n");
        return schedule(r); //返回新的进程的 tf
    case(_EVENT_IRQ_TIME):
        printf("IRQ TIME event!\n");
        return schedule(r); //返回新的进程的 tf

```

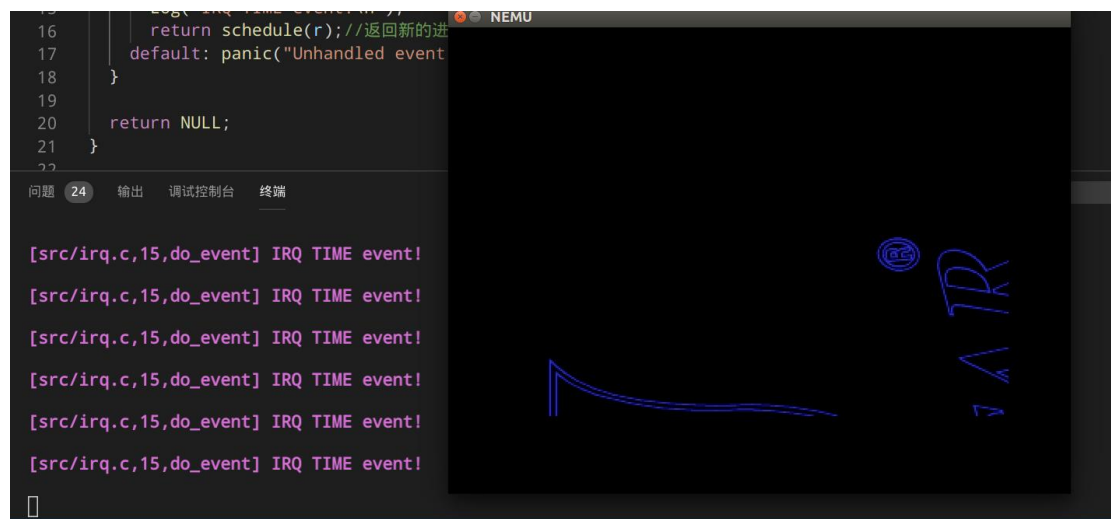
- 为了可以让处理器在运行用户进程的时候响应时钟中断, 还需要修改 `_umake()` 的代码, 在构造现场的时候, 设置正确的 EFLAGS.

```

C++
    tf.eflags=0x02|FL_IF; // turn on irq time

```

运行结果如下:



Nanos-lite 收到了 `_EVENT_IRQ_TIME` 事件。

5. 编写不朽的传奇

5.1 展示计算机系统

注意先将系统调用之后调用的 `schedule()` 代码恢复。

让 Nanos-lite 加载第 3 个用户程序 `bin/videotest`, 并在 Nanos-lite 的 `events_read()` 函数中添加以下功能: 当发现按下 F12 的时候, 让游戏在仙剑奇侠传和

videotest 之间切换.

C++

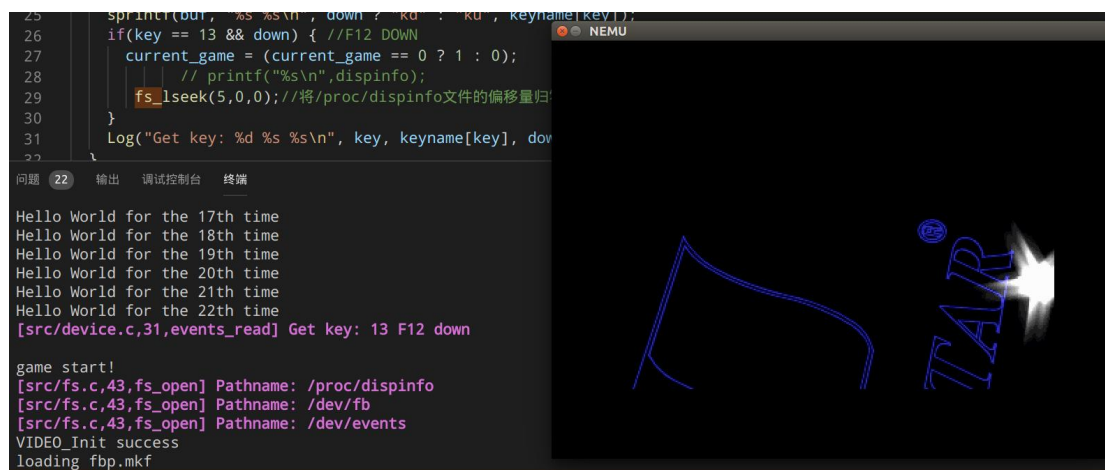
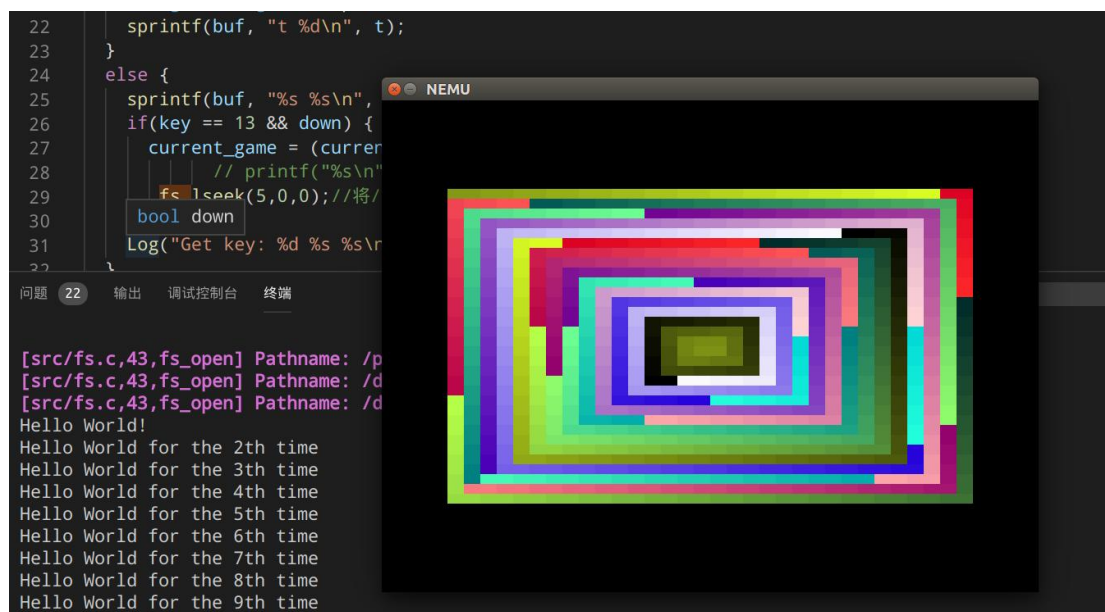
```
    if(key == 13 && down) { //F12 DOWN
        current_game = (current_game == 0 ? 1 : 0);
                        // printf("%s\n",dispinfo);
        fs_lseek(5,0,0);//将/proc/dispinfo 文件的偏移量归零, 否则
    }
    则会报错
```

为了实现这一功能, 还需要修改 `schedule()` 的代码: 通过一个变量 `current_game` 来维护当前的游戏, 在 `current_game` 和 `hello` 程序之间进行调度. 例如, 一开始是 `videotest` 和 `hello` 程序分时运行, 按下 F12 之后, 就变成仙剑奇侠传和 `hello` 程序分时运行:

C++

```
    if(num==freq){//如果到达 1000 次, 则切换成进程 1
        current=&pcb[1];
        num=0;
    }
    else{
        if(current_game)
        {
            current = &pcb[0];
            num++;
        }
        else
        {
            current = &pcb[2];
            num++;
        }
    }
}
```

运行结果如下:



可以看到，按下 F12 后成功切换为仙剑奇侠传。

6. 思考题

一些问题

- i386 不是一个 32 位的处理器吗，为什么表项中的基地址信息只有 20 位，而不是 32 位？

- 手册上提到表项(包括 CR3)中的基地址都是物理地址, 物理地址是必须的吗? 能否使用虚拟地址?
- 为什么不采用一级页表? 或者说采用一级页表会有什么缺点?

i386 是一个 32 位的处理器架构, 但在表项中的基地址信息只有 20 位的原因是因为历史原因和设计考虑。

在 i386 架构中, 采用的是分页机制来管理内存。每个页表项 (Page Table Entry, PTE) 用来描述一个页的属性和物理地址。由于 32 位处理器的寻址空间为 32 位, 可寻址的最大物理地址空间为 4GB。为了对这 4GB 的物理地址空间进行分页, 使用了 4KB 大小的页 (Page), 而每个页表项需要对应一个页的物理地址。

20 位的基地址字段能够表示 2^{20} 个不同的页, 每个页大小为 4KB, 因此基地址字段可以表示 $2^{20} \times 4KB = 4GB$ 的物理地址空间。因此, 20 位基地址可以覆盖整个 32 位处理器的物理地址空间。

在 x86 架构中, 页表的基地址 (CR3 寄存器的值) 是物理地址, 而不是虚拟地址。这是因为分页机制是在虚拟地址和物理地址之间建立映射关系, 虚拟地址通过页表转换为对应的物理地址。因此, 页表的基地址需要是物理地址, 以确保在转换过程中能够正确找到页表。

关于为什么不采用一级页表的问题, 一级页表可以在理论上覆盖整个物理地址空间, 但它存在以下一些缺点:

1. 内存开销: 一级页表需要占用连续的物理内存空间, 而对于大容量的物理地址空间来说, 一级页表的大小会非常大, 导致内存开销较大。
2. 转换效率: 一级页表需要进行线性搜索以找到对应的页表项, 这会导致页表转换的效率降低。而采用多级页表, 如二级页表、三级页表等, 可以将整个物理地址空间划分为多个较小的区域, 从而减少搜索的时间和开销。

通过使用多级页表, 可以更有效地管理大容量的物理地址空间, 并且在转换时减少搜索时间和内存开销。因此, 多级页表是在实际系统中更常见和实用的页表管理方式。

空指针真的是"空"的吗?

程序设计课上老师告诉你, 当一个指针变量的值等于 NULL 时, 代表空, 不指向任何东西. 仔细想想, 真的是这样吗? 当程序对空指针解引用的时候, 计算机内部具体都做了些什么? 你对空指针的本质有什么新的认识?

空指针确实表示指针不指向任何有效的内存位置。当一个指针变量的值等于 NULL 时, 它并不指向任何具体的内存地址。

然而，对空指针进行解引用（即访问空指针所指向的内存）是一种未定义行为。这意味着程序员不应该对空指针进行解引用，因为它不会有定义明确的行为。当程序对空指针解引用时，其结果是不可预测的，可能会导致崩溃或其他意想不到的行为。

在计算机内部，当尝试解引用空指针时，处理器通常会发现指针为空，这可能会导致异常或错误。具体行为可能因编程语言、编译器和操作系统而异。

空指针的本质是一个特殊的指针值，用于表示指针变量不指向有效的内存地址。它提供了一种机制来表示缺少有效指向的情况。然而，对空指针进行解引用是一种编程错误，应该避免。

在编程中，通常需要在`使用指针之前检查其是否为空`，以确保安全性。这样可以避免潜在的崩溃或不确定行为。此外，编程语言和标准库通常提供了一些机制来处理空指针，例如通过条件语句或异常处理来捕获和处理空指针异常。

内核映射的作用

在`_protect()`函数中创建虚拟地址空间的时候，有一处代码用于拷贝内核映射：

```
for (int i = 0; i < NR_PDE; i++) {  
    updir[i] = kpdirs[i];  
}
```

尝试注释这处代码，重新编译并运行，你会看到发生了错误。请解释为什么会发生这个错误。

错误如下：

```
[src/monitor/monitor.c,65,load_img] The image is /home/user/ics2017/nanos-lite/build/nanos-lite-x86-nemu.bin  
Welcome to NEMU!  
[src/monitor/monitor.c,30,welcome] Build time: 19:22:54, May 22 2023  
For help, type "help"  
(nemu) c  
[src/mm.c,24,init_mm] free physical pages starting from 0x2cdf000  
[src/main.c,20,main] 'Hello World!' from Nanos-lite  
[src/main.c,21,main] Build time: 19:30:46, May 22 2023  
[src/ramdisk.c,26,init_ramdisk] ramdisk info: start = 0x1029c0, end = 0x2c99f18, size = 45708632 bytes  
[src/main.c,28,main] Initializing interrupt/exception handler...  
[src/fs.c,43,fs_open] Pathname: /bin/dummy  
pde present bit is 0!  
nemu: src/memory/memory.c:47: page_translate: Assertion `0' failed.  
make[1]: *** [run] 已放弃 (core dumped)  
make[1]: 正在离开目录 `/home/user/ics2017/nemu'  
make: *** [run] 错误 2
```

原因是在默认情况下，操作系统需要将内核空间映射到每个进程的地址空间中，以便进程可以访问内核提供的功能和资源。通常通过拷贝内核页目录项的方式实现这个映射。页目录项的 `present` 位应该设置为 1，以表示该页目录项有效。然而，将该行代码注释后，没有对 `updir` 中的页目录项进行正确的初始化，导致它们的 `present` 位为 0。

灾难性的后果(这个问题有点难度)

假设硬件把中断信息固定保存在内存地址 `0x1000` 的位置, AM 也总是从这里开始构造 `trap frame`. 如果发生了中断嵌套, 将会发生什么样的灾难性后果? 这一灾难性的后果将会以什么样的形式表现出来? 如果你觉得毫无头绪, 你可以用纸笔模拟中断处理的过程.

发生中断嵌套可能导致严重的后果, 其中最常见的问题是栈溢出 (`stack overflow`) 和数据损坏. 让我们分析一下可能的情况和后果:

1. 栈溢出: 当发生中断嵌套时, 每个中断会使用一部分栈空间来保存当前的上下文信息 (如中断处理程序的返回地址、寄存器状态等). 如果中断嵌套层级过多, 栈空间可能会被耗尽, 导致栈溢出. 栈溢出可能导致程序异常终止或系统崩溃.
2. 数据损坏: 中断处理程序通常会使用栈来保存临时数据和局部变量. 如果中断嵌套层级过多, 不同中断处理程序之间使用的栈空间可能会重叠, 导致数据损坏. 这可能导致错误的数据处理、不一致的状态或未定义的行为.
3. 上下文混乱: 中断处理程序通常需要保存和恢复处理前的上下文信息, 如寄存器状态、程序计数器等. 如果中断嵌套层级过多, 保存和恢复上下文的过程可能会混乱或出错, 导致错误的上下文恢复, 进而导致程序错误或系统不稳定.

这些灾难性后果可能表现为系统崩溃、数据错误、程序异常终止、无法预测的行为等. 严重情况下, 中断嵌套可能导致整个系统无法正常运行, 需要进行系统重启或修复. 因此, 对于中断处理程序的编写和中断嵌套的管理需要特别小心, 以确保系统的稳定性和可靠性.

必答题

分时多任务的具体过程 请结合代码, 解释分页机制和硬件中断是如何支撑仙剑奇侠传和 `hello` 程序在我们的计算机系统(Nanos-lite, AM, NEMU)中分时运行的.

1. 分页机制

首先, `nemu` 这一平台提供了 `CR0` 和 `CR3` 寄存器, `CR0` 寄存器负责开启分页机制, `CR3` 寄存器存储页目录基址. MMU 进行虚拟地址到物理地址的转换, 通过 `vaddr_read`, `vaddr_write` 进行对虚拟地址的访问.

我们准备了内核页表用于访问内核虚拟空间. AM 的 `init_mm` 调用 `_pte_init` 函数填写了内核页表.

为了在用户空间上加载用户程序, `nanos-lite` 使用 `load_prog` 函数来加载程序, 调用 `_protect` 函数将内核页表的内容拷贝到用户进程的页表, 然后 `loader` 加载程序. 当

不断分配从 `0x8048000` 开始的虚拟地址后，用 `new_page` 申请物理页，然后用将 `va->pa` 的映射加入页表。

2. 硬件中断与进程调度

为了实现进程调度，需要用 `umake` 来创建进程的上下文，保存在 `trap frame` 中。当需要进程切换时，就通过硬件产生一个外部中断（这里是时钟中断 `IRQ_TIME`），封装成事件并进行分发。中断处理时，将旧进程的 `tf` 压栈，使用 `schedule` 函数得到新进程的 `tf` 地址，`asm_trap` 让栈顶指针指向它，中断返回时就完成了进程切换。

万变之宗 - 重新审视计算机

什么是计算机？为什么看似平淡无奇的机械，竟然能够搭建出如此缤纷多彩的计算机世界？那些酷炫的游戏画面，究竟和冷冰冰的电路有什么关系？看着仙剑奇侠传运行的画面，不妨思考一下，**NEMU** 和 **AM** 分别如何支撑仙剑奇侠传的运行？

计算机是一种能够进行数据处理和执行指令的设备。它由硬件和软件组成，其中硬件包括各种电子元件和电路，而软件则是运行在计算机上的程序和数据。

计算机之所以能够创造出如此缤纷多彩的计算机世界，是因为计算机具备了处理和存储信息的能力，并且可以根据程序的指令进行自动化的操作。通过编写不同的软件程序，我们可以实现各种功能，如游戏、图形处理、音频播放等。这些软件程序利用计算机的硬件资源，通过算法和逻辑来实现各种复杂的计算和操作，从而展现出令人惊叹的画面、音效和交互体验。

NEMU 和 **AM** 是计算机模拟器和操作系统的组成部分，它们共同支撑着仙剑奇侠传的运行。

- **NEMU** 能够运行仙剑奇侠传的机器码指令，从而实现游戏的运行。**NEMU** 负责将指令翻译成对应的硬件操作，并处理硬件中断、内存管理等功能，以确保游戏的正确执行。
- 在仙剑奇侠传运行时，**AM** 负责管理和调度进程、分配内存、处理文件系统和设备驱动等。它为游戏提供了运行环境和必要的资源管理，使得游戏可以在计算机中正确运行，并与用户进行交互。

总而言之，**NEMU** 模拟计算机的硬件行为，而 **AM** 提供了操作系统的功能，它们共同协作，为仙剑奇侠传等游戏的运行提供了必要的支持和环境。通过计算机的工作原理和软硬件的配合，才能让冷冰冰的电路变得生动起来，创造出令人沉浸的游戏世界。

7. Bug Log

缺少 klibxxx.a 文件

```
Makefile:4: /Makefile.app: 没有那个文件或目录
make: *** 没有规则可以创建目标“/Makefile.app”。 停止。
user@user-VirtualBox:~/ics2017/nanos-lite$ ^C
user@user-VirtualBox:~/ics2017/nanos-lite$ echo $AM_HOME
/home/user/ics2017/nexus-am
```

```
ld: cannot find /home/user/ics2017/nexus-am/libs/klib/build/klib-x86-nemu.a: 没有那个文件或目录
ld: cannot find /home/user/ics2017/nexus-am/libs/klib/build/klib-x86-nemu.a: 没有那个文件或目录
objdump: '/home/user/ics2017/nanos-lite/build/nanos-lite-x86-nemu': 没有这个文件
objcopy: '/home/user/ics2017/nanos-lite/build/nanos-lite-x86-nemu': 没有这个文件
```

Git merge 出了问题，重新回到 pa3 把分支 merge 好即可。

0x0010135f 处缺少指令

实现完 `_map()` 后报错：

```
[src/mm.c,24,init_mm] free physical pages starting from 0x2cdf000
invalid opcode(eip = 0x0010135f): 0f 22 d8 0f 20 c0 89 45 ...

There are two cases which will trigger this unexpected exception:
1. The instruction at eip = 0x0010135f is not implemented.
2. Something is implemented incorrectly.
Find this eip(0x0010135f) in the disassembling result to distinguish which case it is.
```

If it is the first case, see



经查询，该指令是 `mov cr0` 的指令。

为了在 NEMU 中实现分页机制，需要添加 CR3 寄存器和 CR0 寄存器，以及相应的操作它们的指令。对于 CR0 寄存器，我们只需要实现 PG 位即可。如果发现 CR0 的 PG 位为 1，则开启分页机制，从此所有虚拟地址的访问(包括 `vaddr_read()`, `vaddr_write()`)都需要经过分页地址转换。为了让 differential testing 机制正确工作，在 `restart()` 函数中我们需要对 CR0 寄存器初始化为 `0x60000011`：

```
C++
cpu.cr0.val = 0x60000011;
```

填写 opcode table：

```
C++
/* 0x20 */      IDEX(G2E, mov_cr2r), EMPTY, IDEX(E2G, mov_r2cr),
EMPTY,
```

在 `system.c` 中实现：

```

C++
make_EHelper(mov_r2cr) {
    switch (id_dest->reg) {
        case 0:
            cpu.cr0.val = id_src->val;
            break;
        case 3:
            cpu.cr3.val = id_src->val;
            break;
        default:
            Assert(0, "Shoule reach here for NO cr%d",
id_dest->reg);
            break;
    }

    print_asm("movl %%s,%%cr%d", reg_name(id_src->reg, 4), id_dest-
>reg);
}

make_EHelper(mov_cr2r) {
    switch (id_src->reg) {
        case 0:
            operand_write(id_dest, &cpu.cr0.val);
            break;
        case 3:
            operand_write(id_dest, &cpu.cr3.val);
            break;
        default:
            Assert(0, "Shoule reach here for NO cr%d",
id_dest->reg);
            break;
    }

    print_asm("movl %%cr%d,%%s", id_src->reg, reg_name(id_dest-
>reg, 4));

#ifdef DIFF_TEST
    diff_test_skip_qemu();
#endif
}

```

即可解决。

实现完成按 F12 报错 BAD TRAP

```
Hello World for the 4th time
Hello World for the 5th time
Hello World for the 6th time
Hello World for the 7th time
[src/device.c,30,events_read] Get key: 13 F12 down

game start!
[src/fs.c,43,fs_open] Pathname: /proc/dispinfo
Assertion failed: screen_w > 0 && screen_h > 0, file src/ndl.c, line 141
nemu: HIT BAD TRAP at eip = 0x00100032
```

解决办法：

需要在 `events_read` 函数中，识别到 F12 键按下时将 `/proc/dispinfo` 文件的偏移量归零：

```
C++
fs_lseek(5,0,0);
```

即可解决。

陈睿颖