南開大學
Nankai University

计算机学院

计算机系统设计期末报告

虚实交错的魔法：分时多任务

姓名：杨馨仪

学号：2011440

专业：计算机科学与技术

2024 年 6 月 4 日

目录

1 概述 **2**
1.1 实验目的 . . . 2
1.2 实验内容 . . . 2

2 阶段一 **2**
2.1 超越容量的界限 . . . 2
2.1.1 一些问题 . . . 2
2.1.2 空指针真的是“空”的吗? . . . 3
2.2 准备内核页表 . . . 3
2.2.1 在 NEMU 中实现分页机制 . . . 3
2.3 让用户程序运行在分页机制上 . . . 6
2.3.1 让用户程序运行在分页机制上 . . . 6
2.3.2 内核映射的作用 . . . 7
2.3.3 在分页机制上运行仙剑奇侠传 . . . 8

3 阶段二 **9**
3.1 上下文切换 . . . 9
3.1.1 实现内核自陷 . . . 9
3.1.2 实现上下文切换 . . . 10
3.2 分时多任务 . . . 12
3.2.1 分时运行仙剑奇侠传和 hello 程序 . . . 12
3.2.2 优先级调度 . . . 12

4 阶段三 **13**
4.1 来自外部的声音 . . . 13
4.1.1 灾难性的后果 . . . 13
4.1.2 添加时钟中断 . . . 13
4.2 必答题 . . . 15

1 概述

1.1 实验目的

1. 了解物理地址到虚拟地址的映射机制，实现分页机制
2. 了解进程调度原理，实现上下文切换、进程调度、分时多任务
3. 实现时钟中断

1.2 实验内容

1. 了解分段和分页机制，在 nemu 中实现分页机制，并让用户程序运行在分页机制之上。
2. 实现内核自陷触发进程切换，实现上下文切换进一步实现分时多任务，对进程调度做调整，实现优先级调度。
3. 实现时钟中断，使用时钟中断触发上下文切换。

2 阶段一

2.1 超越容量的界限

2.1.1 一些问题

1. **Q:** 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?

   A: 因为一个内存段的大小通常不会超过 1MB，而 1MB 用 20 位二进制数表示刚好够用。这样做有助于节省内存空间，并且可以提高内存访问效率

2. **Q:** 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?

   A: CR3 是用来提供一级页表的物理地址。用虚拟地址则寄存器存储的是能够映射到一级页表的物理地址的虚拟地址，由于 CR3 存储虚拟地址，如果 CR3 是虚拟地址 MMU 在进行翻译时会用把 CR3 中的虚拟地址当作为物理地址进行映射，会导致映射到其他内存地址，所以不能用虚拟地址。

3. **Q:** 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?

   A:

   - 占用内存空间较大:

     一级页表需要记录整个虚拟地址空间的映射关系，当虚拟地址空间较大时，页表的大小也会相应变大，占用较多的内存空间。

- 访问时间复杂度高：
  一级页表是线性表结构，在查找页表项时需要对整个页表进行遍历，访问时间复杂度为 O(n)。
- 更新页表项的代价较高：
  页表项的更新需要遍历整个页表，在更新时也会带来较大的开销。
- 不易实现共享：
  由于每个进程都有自己的一级页表，因此不易实现页表的共享，无法实现类似于写时复制等功能。

2.1.2 空指针真的是“空”的吗？

Q: 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?
A: 等于 NULL 时指针指向的地址为 0x0，其物理存储的内容没有访问权限。

2.2 准备内核页表

2.2.1 在 NEMU 中实现分页机制

分页机制的实现需要使用到用来存放页目录基地址的 CR3 寄存器及用来表示分页机制的开始与否的 CR0 寄存器的 PG 位。因此需要先在 nemu/include/cpu/reg.h 中的 CPU_state 结构中完善寄存器结构。先前通过阅读代码已知在 nemu/include/memory/mmu.h 有对 CR0、CR3 的结构定义，因此在完善寄存器结构的时候可以直接使用到已经定的 CR0、CR3 联合体。

```
CR0 cr0;
CR3 cr3;
```

添加好后需要在 nemu/src/monitor/monitor.c 的 restart() 函数中完成对 CR0 的初始化：

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

内核页表相关代码已经在框架中实现, 需要在 nanos-lite/src/main.c 中定义宏 HAS_PTE, 使得在 Nanos-lite 初始化时可以调用 init_mm 函数初始化存储管理器，启动对分页机制的存储管理。

此时在 nanos-lite 目录下先 make update 再 make run，运行 dummy 程序，发现程序中断。

出现报错是由于在执行 init_mm 函数时遇到了没有实现过的指令。通过查阅 i386 手册 P415 可知当前遇到的未实现指令是 mov_r2cr，同样会被用到的未实现指令还有 mov_cr2r。

首先在 nemu/src/cpu/exec/exec.c 的 2 byte_opcode_table 中补全指令：

```
/* 0x20 */    IDEX(mov_G2E, mov_cr2r), EMPTY, IDEX(mov_E2G, mov_r2cr), EMPTY,
```

查阅 i386 手册 P347 了解到 mov_r2cr、mov_cr2r 这两条指令的实现需要先判断涉及到的特殊寄存器是 CR0 还是 CR3(对于其他寄存器由于不涉及，认为不会出现，如果出现就使用 assert 中断)，然后再将源操作数的值赋给目标操作数：

```
make_EHelper(mov_r2cr) {
  //TODO();
  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);
  }
  print_asm("movl %%%s,%%cr%d", reg_name(id_src->reg, 4), id_dest->reg);
```

```
make_EHelper(mov_cr2r) {
  //TODO();
  switch(id_src->reg){
    case 0:{
      t0 = cpu.cr0.val;
      break;
    }
    case 3:{
      t0 = cpu.cr3.val;
      break;
    }
    default:
      assert(0);
  }
```

由于使用分页机制，内存的读写也相应发生了变化，地址读取的过程中需要进行分页地址转换，nemu/src/memory/memory.c 中的内存读写函数 vaddr_read 和 vaddr_write 函数也会相应的进行修改。

分页地址转换在 memory.c 中编写 page_translate 函数实现。根据实验手册中对于 vaddr_read 函数的示例书写可知内存读写时应先调用 page_translate 函数应当返回页地址，之后再对页地址使用 paddr_read 或 paddr_write 函数进行读写。

根据 vaddr_read 函数伪代码示例，修改 vaddr_read 和 vaddr_write 函数：

```
uint32_t vaddr_read(vaddr_t addr, int len) {
  paddr_t paddr;
  if (CrossBoundary(addr, len)) {
    /* data cross the page boundary */
    assert(0);
  }
  else {
    paddr = page_translate(addr, false);
    paddr_read(addr, len, data);
  }
}
```

CrossBoundary 函数用于判断数据跨越虚拟页边界，根据手册目前可以先使用 assert(0) 中止，等遇到这种情况再处理。CrossBoundary 函数通过传入读写的起始地址和长度获取终止地址，然后起始地址和终止地址是否在一个页中来判断是否跨越边界，其实现如下：

```
bool CrossBoundary(vaddr_t addr, int len){
  vaddr_t end_addr=addr+len-1;
  if((end_addr&(~PAGE_MASK))!=(addr&(~PAGE_MASK)))
    return true;
  return false;
}
```

进行分页地址转换首先要判断是否开启了分页机制，也就是对 CR0 的 PG 是否为 1 进行检查，如果未开启分页机制可以直接返回。如果开启了分页机制，要先通过 CR3 寄存器获取页目录基地址，然后通过内存读取获取页目录表项的值。然后再通过页目录表象获取页表项的位置，通过内存读取获取页表项的值，进而得到映射到的地址。同时要注意要给页目录表项和页表项的 access 位置位，如果到是写操作还需要对 dirty 位置位：

```
paddr_t page_translate(vaddr_t addr, bool is_write) {
  if(!cpu.cr0.paging) return addr;
  paddr_t paddr = addr;

  PDE pde, *pgdir;
  pgdir = (PDE *)(intptr_t)(cpu.cr3.page_directory_base << 12);
  pde.val = paddr_read((intptr_t)&pgdir[(addr >> 22) & 0x3ff], 4);
  assert(pde.present);
  pde.accessed = 1;

  PTE pte, *pgtab;
  pgtab = (PTE *)(intptr_t)(pde.page_frame << 12);
  pte.val = paddr_read((intptr_t)&pgtab[(addr >> 12) & 0x3ff], 4);
  assert(pte.present);
  pte.accessed = 1;

  if(is_write) pte.dirty=1;
  paddr = (pte.page_frame << 12) | (addr & PAGE_MASK);

  return paddr;
}
```

实现好后去运行程序，发现会由于跨页触发中断。因此需要解决跨页读写的问题。在跨页的情况下需要在不同的页中去读写，如果一次只读写一字节，就不会出现跨页的问题

了。跨页读操作实现如下：

```
union{
  uint8_t bytes[4];
  uint32_t dword;
} data = {0};
for(int i=0; i < len; i++){
  paddr =  page_translate(addr+i, false);
  data.bytes[i] = (uint8_t)paddr_read(paddr, 1);
}
return data.dword;
```

跨页写操作实现如下：

```
for(int i=0; i < len; i++){
  paddr =  page_translate(addr, true);
  paddr_write(paddr, 1, data);
  data >>= 8;
  addr++;
}
```

实现好后重新运行程序，运行成功，跨页读写实现正确。

```
(nemu) c
[src/mm.c,24,init_mm] free physical pages starting from 0x1d90000
[src/main.c,21,main] 'Hello World!' from Nanos-lite
[src/main.c,22,main] Build time: 03:49:17, Jun  3 2024
[src/ramdisk.c,26,init_ramdisk] ramdisk info: start = 0x102480, end = 0x1d4c3f5, size = 29663093 bytes
[src/main.c,29,main] Initializing interrupt/exception handler...
[src/loader.c,22,loader] opening file /bin/dummy
[src/loader.c,27,loader] successfully open file
nemu: HIT GOOD TRAP at eip = 0x00100032
```

就此实现了分页机制。

2.3 让用户程序运行在分页机制上

2.3.1 让用户程序运行在分页机制上

为了让用户程序运行在操作系统为其分配的虚拟地址空间之上，首先将 navy-apps/Makefile.compil 中的链接地址-Ttext 参数改为 0x8048000，以避免用户程序的虚拟地址空间与内核相互重叠。nanos-lite/src/loader.c 中的 DEFAULT_ENTRY 也需要作相应的修改。

在 nanos-lite/src/proc.c 中使用 load_prog 函数来加载用户程序。load_prog() 会调用 loader() 函数加载用户程序，此时 loader 函数需要根据用户程序大小以页为单位加载程序：

首先申请空闲页；然后将该页映射到虚拟地址空间；最后从文件读入一页的内容到该物理页：

```
Log("Paging mechanism loading file %s ",filename);
int fd = fs_open(filename, 0, 0);
size_t nbyte = fs_filesz(fd);
void *pa;
void *va;
void *end = DEFAULT_ENTRY + nbyte;
for (va = DEFAULT_ENTRY; va < end; va += PGSIZE) {
  pa = new_page();
  _map(as, va, pa);
  fs_read(fd, pa, (end - va) < PGSIZE ? (end - va) : PGSIZE);
}
return (uintptr_t)DEFAULT_ENTRY;
```

在进行映射时使用到了 nexus-am/am/arch/x86-nemu/src/pte.c 中的 _map 函数，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 获取页目录基地址，映射过程中需要申请信页表时借助回调函数 palloc_f 函数获取空闲物理页。

```
void _map(_Protect *p, void *va, void *pa) {
  PDE *pt = (PDE*)p->ptr;
  PDE *pde = &pt[PDX(va)];
  if (!(*pde & PTE_P)) *pde = PTE_P | PTE_W | PTE_U | (uint32_t)palloc_f();
  PTE *pte = &((PTE*)PTE_ADDR(*pde))[PTX(va)];
  if (!(*pte & PTE_P)) *pte = PTE_P | PTE_W | PTE_U | (uint32_t)pa;
}
```

实现完上述函数后，返回 load_prog 函数会调用 _switch 函数切换到用户程序创建的地址空间。

运行 dummy 程序

```
(nemu) c
[src/mm.c,24,init_mm] free physical pages starting from 0x1d90000
[src/main.c,21,main] 'Hello World!' from Nanos-lite
[src/main.c,22,main] Build time: 03:49:17, Jun  3 2024
[src/ramdisk.c,26,init_ramdisk] ramdisk info: start = 0x1024e0, end = 0x1d4c455, size = 29663093 bytes
[src/main.c,29,main] Initializing interrupt/exception handler...
[src/loader.c,33,loader] Paging mechanism loading file /bin/dummy
nemu: HIT GOOD TRAP at eip = 0x00100032
```

输出了使用分页机制加载的 Log，并输出 GOOD TRAP 的信息，说明用户程序成功在虚拟地址空间运行。

2.3.2 内核映射的作用

Q: 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:

```
1  for (int i = 0; i < NR_PDE; i ++) {
2    updir[i] = kpdirs[i];
3  }
```

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.

A: 在 vme_init 中内核映射是用来设置一级页表的，如果不进行拷贝，则用户进程中的页表指向的就不是操作系统分配给你的一级页表而是一张空表。

2.3.3 在分页机制上运行仙剑奇侠传

尝试运行仙剑奇侠传程序，程序加载成功，但使用到了不存在的 PTE，触发了中断：

```
(nemu) c
[src/mm.c,24,init_mm] free physical pages starting from 0x1d90000
[src/main.c,21,main] 'Hello World!' from Nanos-lite
[src/main.c,22,main] Build time: 08:40:36, Jun  3 2024
[src/ramdisk.c,26,init_ramdisk] ramdisk info: start = 0x1024e0, end = 0x1d4c455, size = 29663093 bytes
[src/main.c,29,main] Initializing interrupt/exception handler...
[src/loader.c,33,loader] Paging mechanism loading file /bin/pal
game start!
nemu: src/memory/memory.c:42: page_translate: Assertion `pte.present' failed.
Makefile:46: recipe for target 'run' failed
make[1]: *** [run] Aborted (core dumped)
make[1]: Leaving directory '/home/yxy/ics2017/nemu'
/home/yxy/ics2017/nexus-am/Makefile.app:35: recipe for target 'run' failed
make: *** [run] Error 2
```

这是因为仙剑奇侠传的运行过程中需要修改用户程序的堆区大小，要想在分页机制上成功运行仙剑奇侠传，还需要在 mm_brk 函数中把新申请的堆区映射到虚拟地址空间中：

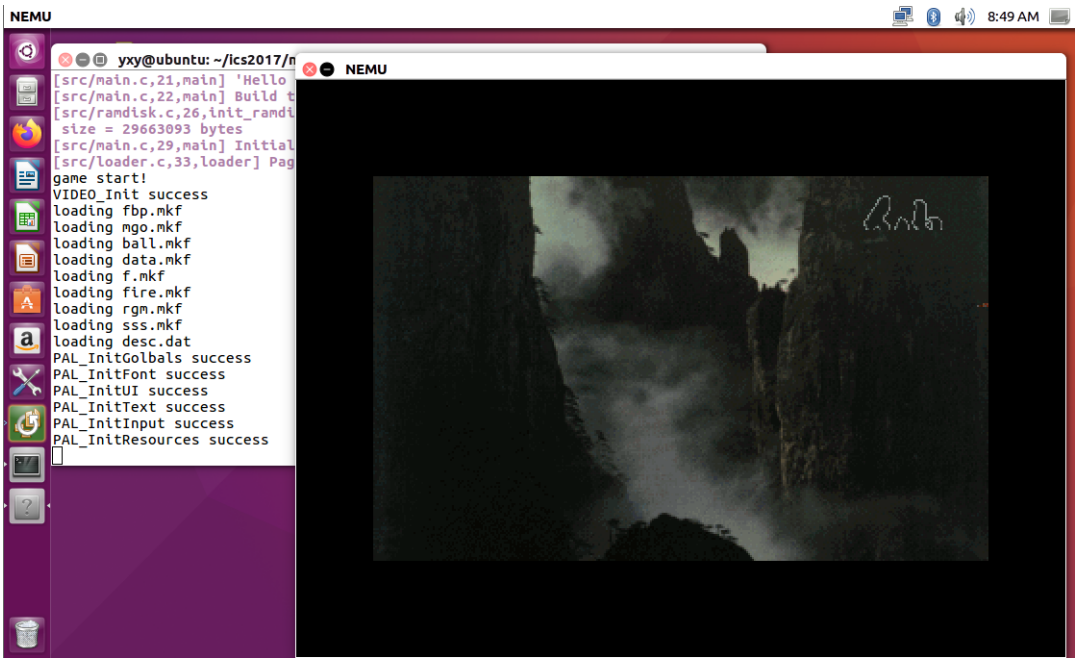
重新实现 mm_brk 函数如下：

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

修改 nanos-lite/src/syscall.c 中的 SYS_brk 系统调用，使之调用 mm_brk 函数。

重新运行仙剑奇侠传程序，成功运行：



3 阶段二

3.1 上下文切换

上下文切换其实就是不同程序之间的堆栈切换，我们只需要在程序的堆栈上人工初始化一个陷阱帧，使得将来切换的时候可以根据这个人工陷阱帧来正确地恢复现场即可。

3.1.1 实现内核自陷

为了检验后续实现的上下文切换的正确性，我们需要通过自陷的方式触发上下文切换。内核自陷的功能与 ISA 相关，是由 ASYE 的 _trap() 函数提供的，在 x86-nemu 的 AM 中内核自陷通过指令 int $0x81 触发。因此使用内联汇编完成 nexus-am/am/arch/x86-nemu/src/asye.c 中的 _trap 函数如下：

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

根据指导书说明，在 Nanos-lite 的 main.c 中调用 _trap 函数。

在 asye.c 中的 irq_handle 函数中将 0x81 异常包装成 _EVENT_TRAP 事件：

```
_RegSet* irq_handle(_RegSet *tf) {
  _RegSet *next = tf;
  if (H) {
    _Event ev;
    switch (tf->irq) {
      case 0x80: ev.event = _EVENT_SYSCALL; break;
      case 0x81: ev.event = _EVENT_TRAP; break;
      default: ev.event = _EVENT_ERROR; break;
    }
```

在 Nanos-lite 中对 _EVENT_TRAP 事件增加处理，在遇到 _EVENT_TRAP 事件后输出提示。实现在 nanos-lite/src/irq.c 的 do_event 函数中：

```
case _EVENT_TRAP:{
  printf("event trap\n");
  return NULL;
}
```

之后还需要对 _EVENT_TRAP 事件设置中断描述符。将入口函数定为 vectrap，在 nexus-am/am/arch/x86-nemu/src/asye.c 中声明，并设置好自陷系统调用：

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

然后在 nexus-am/am/arch/x86-nemu/src/trap.S 中补充具体定义：

```
.globl vectrap;  vectrap:  pushl $0;  pushl  $0x81; jmp asm_trap
```

注释掉 proc.c 中标注的三行代码，运行程序：

```
(nemu) c
[src/mm.c,39,init_mm] free physical pages starting from 0x1d90000
[src/main.c,21,main] 'Hello World!' from Nanos-lite
[src/main.c,22,main] Build time: 20:30:16, Jun  3 2024
[src/ramdisk.c,26,init_ramdisk] ramdisk info: start = 0x1025c0, end = 0x1d4c535, size = 29663093 bytes
[src/main.c,29,main] Initializing interrupt/exception handler...
[src/loader.c,33,loader] Paging mechanism loading file /bin/pal
event trap
[src/main.c,50,main] system panic: Should not reach here
nemu: HIT BAD TRAP at eip = 0x00100032
```

可以看到触发了 _EVENT_TRAP 事件，说明内核自陷实现成功。

3.1.2 实现上下文切换

要想实现真正的上下文切换需要实现 nexus-am/am/arch/x86-nemu/src/pte.c 中的 _umake 函数，在该函数中需要先入栈三个参数和指令寄存器，然后对 tf 进行初始化，为保证 differential testing 的正确运行要将陷阱帧中的 cs 设置为 8：

```
_RegSet *_umake(_Protect *p, _Area ustack, _Area kstack, void *entry, char *const argv[], char *const envp[]) {
  _RegSet **tf=ustack.start;
  uint32_t *tempStack=(uint32_t *)(ustack.end - 4);
  for(int i=0;i<3;i++){
    (*tempStack)=0;
    (*tempStack)--;
  }
  (*tf)=(void *)(tempStack-sizeof(_RegSet));
  (*tf)->eflags=0x2|(1<<9);
  (*tf)->cs=8;
  (*tf)->eip=(uintptr_t)entry;
  return *tf;
}
```

上下文切换只是 AM 的工作，而具体切换到哪个进程的上下文需要进行进程调度。进程调度由 nanos-lite/src/proc.c 中的 schedule 函数完成。current 指针用于指向当前运行进程的 PCB，目前 schedule() 只需要总是切换到第一个用户进程，即 pcb[0]。schedule 函数返回将要调度的进程的上下文：

```
_RegSet* schedule(_RegSet *prev) {
  current->tf = prev;
  current=&pcb[0];
  _switch(&current->as);
  return current->tf;
}
```

上下文是在加载程序的时候通过 _umake() 创建的，在 schedule() 中才决定要切换到它，然后在 ASYE 的 asm_trap() 中才真正地恢复这一上下文。因此还要在 nanos-lite/src/irq.c 中完善对事件的处理，收到 _EVENT_TRAP 事件后应该调用 schedule 函数返回现场：

```
    case _EVENT_TRAP:{
      printf("event trap\n");
      return schedule(r);
    }
```

除此之外还需修改 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：

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

重新运行程序：

通过自陷方式成功触发程序。

3.2 分时多任务

3.2.1 分时运行仙剑奇侠传和 hello 程序

实现好上下文切换后，要想同时运行两个程序，要在 main.c 中加载完仙剑奇侠传程序后，再通过 **load_prog("/bin/hello")** 加载第二个程序。

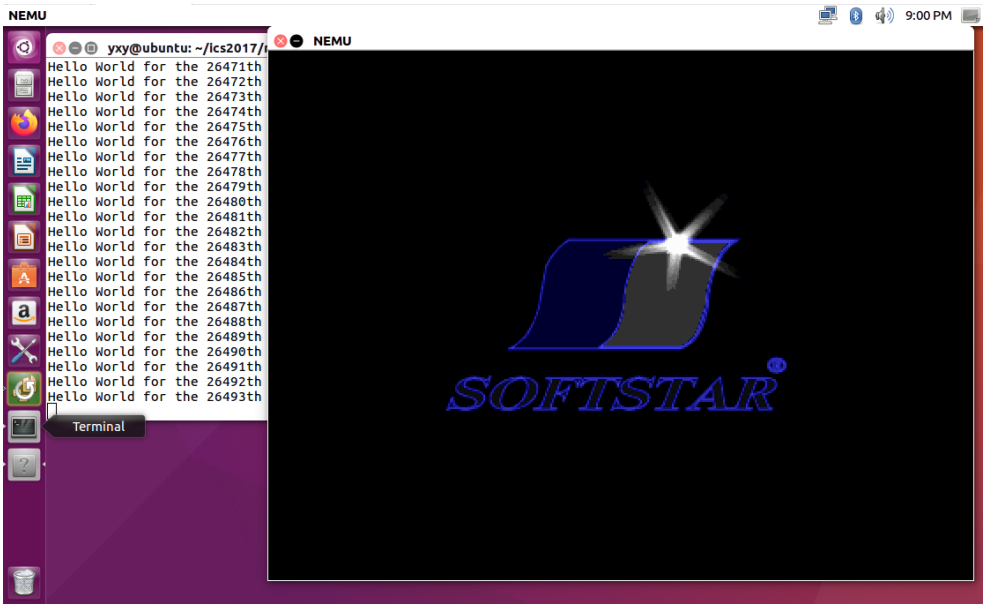
因为最多只允许一个需要更新画面的进程参与调度，还需要修改调度的代码，在 schedule 函数中设置好 current 的值：

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

进程调度需要触发，选择在系统调用之后触发进程调度，修改 do_event 函数代码，在处理完系统调用之后，调用 schedule() 函数并返回其现场：

```
case _EVENT_SYSCALL:{
  do_syscall(r);
  return schedule(r);
}
```

编译运行，实现两程序的分时运行，但明显仙剑的加载要比 hello 的输出慢的多，因此输出了很多次 Hello，仙剑程序也没加载运行到开始界面。

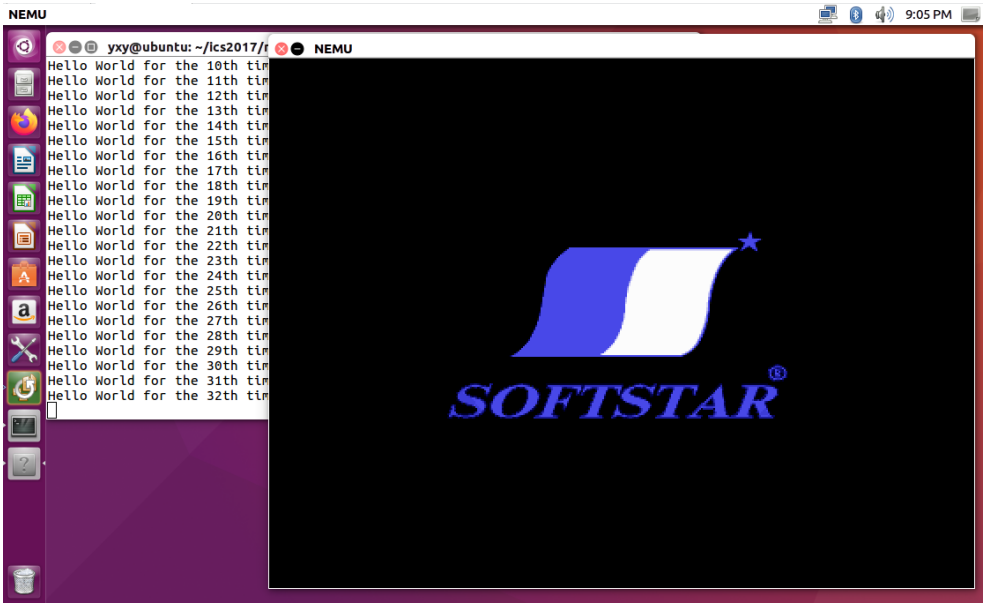


3.2.2 优先级调度

修改 schedule 代码，计算 schedule 函数的调用次数，让仙剑奇侠传调度 1000 次，才让 hello 程序调度 1 次：

```
static int count = 0;
_RegSet* schedule(_RegSet *prev) {
  current->tf = prev;
  //current = (current==&pcb[0])?&pcb[1]:&pcb[0];
  count++;
  if(count%1000 == 0) current = &pcb[1];
  else current = &pcb[0];
  _switch(&current->as);
  return current->tf;
}
```

重新运行：



仙剑的运行速率得到了很大的提升。

4 阶段三

4.1 来自外部的声音

4.1.1 灾难性的后果

Q: 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?

A: 原先保存的上下文会被破坏，当前中断执行完后会接着继续执行中断嵌套（因为新的中断的上下文保存的内容就是即将执行另一个新中断时的状态），造成死循环。

4.1.2 添加时钟中断

将时钟中断的中断引脚与 CPU 的 INTR 引脚相连，中断信号就可以穿的地道引脚中，实现 CPU 对中断的响应。在 cpu 结构体中添加一个 bool 成员 INTR;

```
bool INTR;
```

将中断引脚置为高电平，中断信号就会一直传到 CPU 的 INTR 引脚。在 nemu/src/cpu/intr.c 的 dev_raise_intr() 函数中将 INTR 引脚设置为高电平：

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

设置好引脚后，需要检查引脚来判断是否有硬件中断到来。在 nemu/src/cpu/exec/exec.c 的 exec_wrapper() 函数末尾添加轮询 INTR 引脚的代码：

```
#define TIMER_IRQ 32
  extern void raise_intr(uint8_t NO, vaddr_t ret_addr);
  if (cpu.INTR && cpu.eflags.IF) {
    cpu.INTR = false;
    raise_intr(TIMER_IRQ, cpu.eip);
    update_eip();
  }
```

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，这部分内容实现在改 nemu/src/cpu/intr.c 的 raise_intr() 函数中：

```
  cpu.eflags.IF = 0;
```

在 ASYE 中添加时钟中断的支持，把时钟中断打包成 _EVENT_IRQ_TIME 事件。在 asye.c 中的 irq_handle 函数中将 32 异常包装成 _EVENT_IRQ_TIME 事件：

```
    case 32: ev.event = _EVENT_IRQ_TIME; break;
```

并增加系统调用：

```
  idt[32] = GATE(STS_TG32, KSEL(SEG_KCODE), vectime, DPL_USER);
```

在 nanos-lite/src/irq.c 的 do_event 函数中增加对 _EVENT_IRQ_TIME 事件的处理，同时去掉系统调用之后调用的 schedule() 代码：

```
    case _EVENT_IRQ_TIME:{
      Log("_EVENT_IRQ_TIME");
      return schedule(r);
    }
```

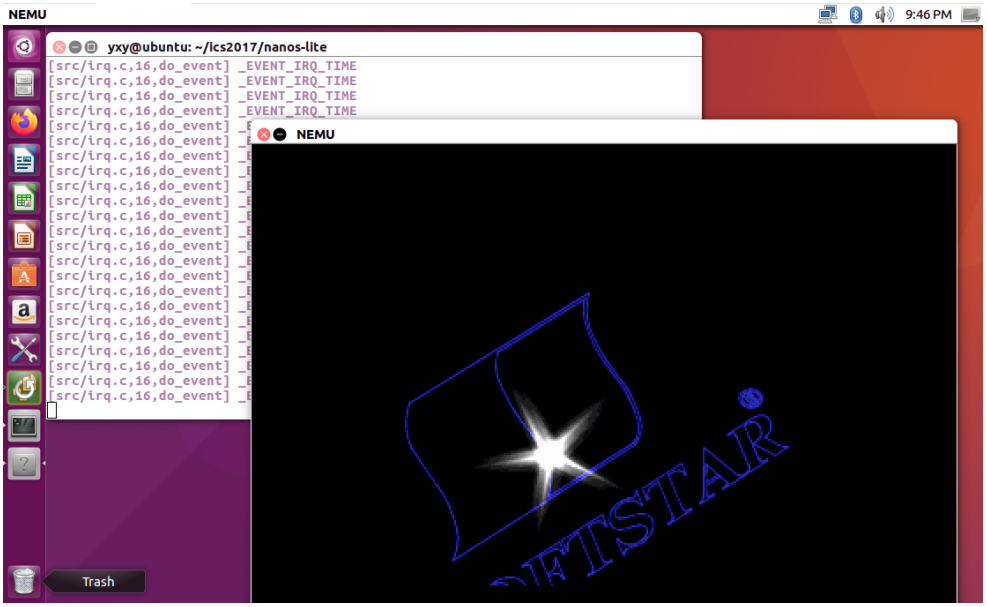
在 nexus-am/am/arch/x86-nemu/src/trap.S 中注册 vectime 函数：

```
.globl vectime;  vectime:  pushl $0;  pushl   $32; jmp asm_trap
```

为了可以让处理器在运行用户进程的时候响应时钟中断，在构造现场是需要设置好 EFLAGS 寄存器的值，在 _umake 函数中完成：

```
(*tf)->eflags=0x2|(1<<9);
```

运行 nemu：



根据输出的 Log 可知时钟中断被正常触发，任务正确分时运行。

4.2 必答题

Q: 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.

A:

分页机制可以给 hello 和仙剑奇侠传分配不同的虚拟地址是两程序运行不会发生冲突：

在 main.c 中调用 init_mm 函数初始化 MM

将堆区起始地址作为空闲物理页的首地址, 这样就不会出此案两程序运行在同一地址上导致的冲突了。

之后 main.c 调用两次 load_prog 函数分别加载不同的程序。

load_prog 函数先调用 _protect() 函数创建一个用户进程的虚拟地址空间；调用 loader 函数数以页为单位加载用户程序；调用 _umake 函数创建用户进程的现场。

这样为两个用户程序创造了用户进程，不同程序的进程存储在了不同的物理空间。

之后 main.c 调用 _trap 函数，系统陷入自陷，完成一次用户调度，跳入 asm_trap 函数中，将栈顶指针切换到回的堆栈上，然后恢复切换进程。

程序就此运行起来。

硬件中断可以使运行的地址在两程序的虚拟地址间来回切换。最终实现分时运行。

仙剑奇侠传程序在运行一定时间会触发时钟中断。时钟中断信号传递到 INTR 引脚。

每执行一条指令 CPU 会检查引脚，如果有中断产生，会触发 _EVENT_IRQ_TIME 事件，调用调度函数 schedule()。

如果达到了设定的进行调度触发次数，就会进行上下文切换，切换到 hello 程序。