目录

目录

阶段一

实现上下文切换

实现上下文切换

根据讲义的上述内容, 实现以下功能:

- CTE的 `_kcontext()` 函数
- Nanos-lite的 `schedule()` 函数
- 在Nanos-lite收到 `_EVENT_YIELD` 事件后, 调用 `schedule()` 并返回新的上下文
- 修改CTE中 `__am_asm_trap()` 的实现, 使得从 `__am_irq_handle()` 返回后, 先将栈顶指针切换到新进程的上下文结构, 然后才恢复上下文, 从而完成上下文切换的本质操作

你需要在 `init_proc()` 中单独创建一个以 `hello_fun` 为返回地址的上下文:

```
void init_proc() {
  context_kload(&pcb[0], hello_fun, NULL);
  switch_boot_pcb();
}
```

其中, `context_kload()` 在 `nanos-lite/src/loader.c` 中定义, 它会调用CTE的 `kcontext()` 来创建一个上下文, 而调用 `switch_boot_pcb()` 则是为了初始化 `current` 指针. 如果你的实现正确, 你将会看到 `hello_fun()` 中的输出信息. 需要注意的是, 虽然 `hello_fun()` 带有一个参数 `arg` , 但目前我们并不使用它, 所以 `_kontext()` 中的 `arg` 参数也可以先忽略, 我们接下来就会考虑它.

按照手册的提示，我们先完成_kcontext_，其目的是在 kstack 栈底创建一个以 entry 为返回地址的上下文结构，然后返回这一结构的指针.

```
_Context *_kcontext(_Area stack, void (*entry)(void *), void *arg) {
_Context *c=(_Context*)(stack.end-sizeof(_Context));
c->eip=(uintptr_t)entry;
c->eflags=0x2;
c->cs=8;
return c;
}
```

根据手册讲的我们目前对于进程的调度永远调度到 pcb[0]就好了，实现如下 schedule 代码进行进程调度

```
}

_Context* schedule(_Context *prev) {

    current->cp = prev;
    current=&pcb[0];
    return current->cp;
}

```

然后我们在收到 EVENT_YIELD 的时候调用 schedule 进行进程调度

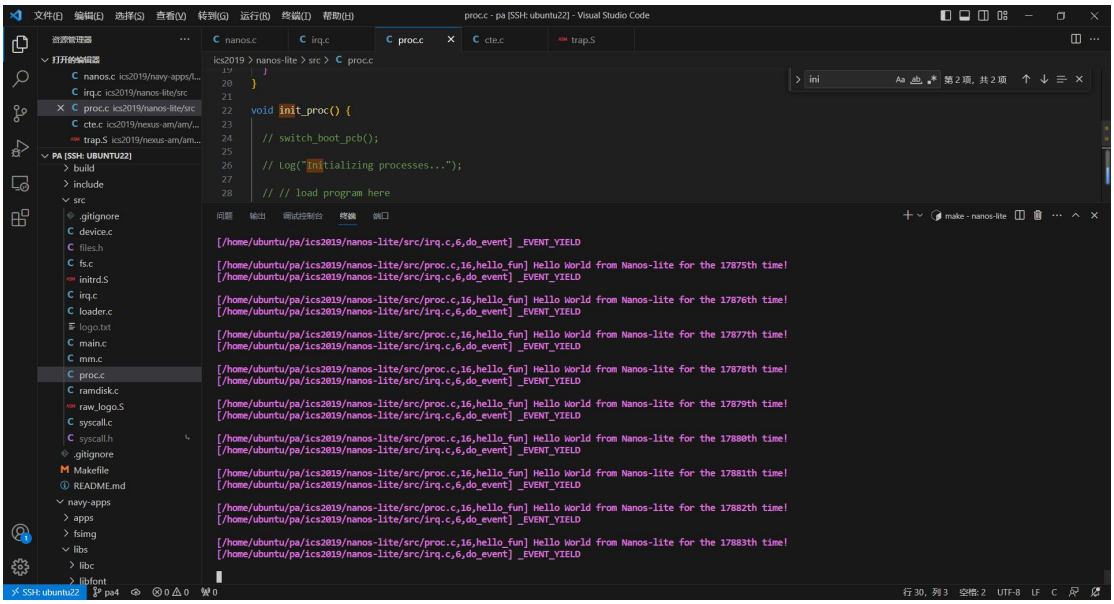
```
switch (e.event) {
  case _EVENT_YIELD:
    Log("_EVENT_YIELD\n");
    return schedule(c);
    break;
```

然后修改_am_asm_trap 使得从 __am_irq_handle() 返回后，先将栈顶指针切换到新进程的上下文结构，然后才恢复上下文，从而完成上下文切换的操作

```
cs2019 > nexus-am > am > src > x86 > nemu > ASM trap.S
.globl __am_irq0;          __am_irq0: pushl   $32;
.globl __am_vecnull;   __am_vecnull: pushl   $-1;

__am_asm_trap:
  pushal

  pushl $0

  pushl %esp
  call __am_irq_handle

  addl $4, %esp

  movl %eax,%esp

  addl $4, %esp
  popal
  addl $4, %esp

  iret
```

我们这里是添加了 movl %eax,%esp，因为调用完 __am_irq_handle 之后，返回值存储在寄存器 eax 中，只需要将 eax 的内容传递给 esp 寄存器即可实现栈顶指针的切换。

使用 hellofun 创建内核线程的效果：



实现上下文切换 2

这一部分我们希望可以为我们的内核线程加入参数，找到指导书的部分：

首先需要解决的第一个问题是, 我们要如何通过 `_kcontext()` 给 `hello_fun()` 传参? 于是我们需要继续思考, `hello_fun()` 将会如何读出它的参数? 噢, 这不就是调用约定的内容吗? 你已经非常熟悉了. 我们只需要让 `_kcontext()` 按照调用约定的内容将 `arg` 放好, 将来 `hello_fun()` 执行的时候就可以获取正确的参数了.

mips32和riscv32的调用约定

我们没有给出mips32和riscv32的调用约定, 你需要查阅相应的ABI手册. 当然, 你也可以自己动手实践来总结传参的规则.

第二个问题是如何在两个内核线程之间来回切换. `hello_fun()` 中每次输出完信息都会调用 `_yield()` , 因此我们只需要对 `schedule()` 进行简单的修改即可:

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

实现上下文切换(2)

根据讲义的上述内容, 实现以下功能:

- 修改CTE的 `_kcontext()` 函数, 使其支持参数 `arg` 的传递
- 修改 `hello_fun()` 函数, 使其输出参数. 你可以自行约定参数 `arg` 的类型, 包括整数, 字符, 字符串, 指针等皆可, 然后按照你的约定来解析 `arg` .
- 通过 `_kcontext()` 创建第二个以 `hello_fun()` 为入口的内核线程, 并传递不同的参数
- 修改Nanos-lite的 `schedule()` 函数, 使其轮流返回两个上下文

如果你的实现正确, 你将会看到 `hello_fun()` 会轮流输出不同参数的信息.

我们按照调用约定，现在除了上下文，我们还先压入的是参数和返回地址，这样的话上下文就不再紧贴着 end 了，参数和地址都是 4 字节，所以 context 的地址还要-8，然后我们把参数放进去贴着 end 的地方，kcontext 修改完成

```
_Context *_kcontext(_Area stack, void (*entry)(void *), void *arg) {

_Context *c=(_Context*)(stack.end-sizeof(_Context)-8);
 void ** arg_stack=(void**)(stack.end-4);
 *arg_stack=arg;
c->eip=(uintptr_t)entry;
c->eflags=0x2;
c->cs=8;
return c;
}
```

接下来我们在 hello_fun 里读取我们的参数

```
12
13  void hello_fun(void *arg) {
14    int j = 1;
15    while (1) {
16      Log("Hello World from Nanos-lite for the %dth time!,%s", j,(char *)arg);
17      j ++;
18      _yield();
19    }
20  }
21  char *arg[] = {"/bin/pal", "--skip"};
```

初始化线程的时候这次可以两个系统线程，我们放入不同的参数

```
2
3  void init_proc() {
4
5    // switch_boot_pcb();
6
7    // Log("Initializing processes...");
8
9    // // load program here
0    // naive_uload(0, "/bin/bmptest");
1    context_kload(&pcb[0], hello_fun, 233);
2    context_kload(&pcb[1], hello_fun, 666);
3    switch_boot_pcb();
4
5  }
```

然后我们修改调度函数，让他在两个线程间来回切换

```
35  }
36
37  _Context* schedule(_Context *prev) {
38
39      current->cp = prev;
40      if(current==&pcb[0])
41      {
42        current=&pcb[1];
43      }
44      else
45      {
46      current=&pcb[0];
47      }
48      return current->cp;
49  }
50
```

发现并没有输出出来我们的参数

```
arg is [/home/ubuntu/pa/ics2019/nanos-lite/src/proc.c,17,hello_fun] Hello World from Nanos-lite for the 2886th time!,arg is
[/home/ubuntu/pa/ics2019/nanos-lite/src/irq.c,6,do_event] _EVENT_YIELD

arg is [/home/ubuntu/pa/ics2019/nanos-lite/src/proc.c,17,hello_fun] Hello World from Nanos-lite for the 2887th time!,arg is
[/home/ubuntu/pa/ics2019/nanos-lite/src/irq.c,6,do_event] _EVENT_YIELD

arg is [/home/ubuntu/pa/ics2019/nanos-lite/src/proc.c,17,hello_fun] Hello World from Nanos-lite for the 2887th time!,arg is
[/home/ubuntu/pa/ics2019/nanos-lite/src/irq.c,6,do_event] _EVENT_YIELD

arg is [/home/ubuntu/pa/ics2019/nanos-lite/src/proc.c,17,hello_fun] Hello World from Nanos-lite for the 2888th time!,arg is
[/home/ubuntu/pa/ics2019/nanos-lite/src/irq.c,6,do_event] _EVENT_YIELD

arg is [/home/ubuntu/pa/ics2019/nanos-lite/src/proc.c,17,hello_fun] Hello World from Nanos-lite for the 2888th time!,arg is
[/home/ubuntu/pa/ics2019/nanos-lite/src/irq.c,6,do_event] _EVENT_YIELD
```

输出一下，发现根本没读进去 arg

```
[/home/ubuntu/pa/ics2019/nanos-lite/src/irq.c,19,init_irq] Initializing interrupt/exception handler...
no such arg!
no such arg!
[/home/ubuntu/pa/ics2019/nanos-lite/src/main.c,33,main] Finish initialization
^C[src/monitor/cpu-exec.c,29,monitor_statistic] total guest instructions = 45851492
```

发现一开始 context_kload 的函数居然没加 arg 这个参数= =怪不得读不进去呢(可没有这个参数为什么我三个参数调用 context_kload 居然不报错而能跑通呢？奇怪！)修改之

```
83
84  void context_kload(PCB *pcb, void *entry,void*arg) {
85    _Area stack;
86    stack.start = pcb->stack;
87    stack.end = stack.start + sizeof(pcb->stack);
88
89    pcb->cp = _kcontext(stack, entry, arg);
90  }
```

成功输出了带参数的信息

```
[/home/ubuntu/pa/ics2019/nanos-lite/src/proc.c,16,hello_fun] Hello World from Nanos-lite for the 4820th time!,kernel thread 666
[/home/ubuntu/pa/ics2019/nanos-lite/src/proc.c,16,hello_fun] Hello World from Nanos-lite for the 4821th time!,kernel thread 233
[/home/ubuntu/pa/ics2019/nanos-lite/src/proc.c,16,hello_fun] Hello World from Nanos-lite for the 4821th time!,kernel thread 666
[/home/ubuntu/pa/ics2019/nanos-lite/src/proc.c,16,hello_fun] Hello World from Nanos-lite for the 4822th time!,kernel thread 233
[/home/ubuntu/pa/ics2019/nanos-lite/src/proc.c,16,hello_fun] Hello World from Nanos-lite for the 4822th time!,kernel thread 666
[/home/ubuntu/pa/ics2019/nanos-lite/src/proc.c,16,hello_fun] Hello World from Nanos-lite for the 4823th time!,kernel thread 233
[/home/ubuntu/pa/ics2019/nanos-lite/src/proc.c,16,hello_fun] Hello World from Nanos-lite for the 4823th time!,kernel thread 666
[/home/ubuntu/pa/ics2019/nanos-lite/src/proc.c,16,hello_fun] Hello World from Nanos-lite for the 4824th time!,kernel thread 233
[/home/ubuntu/pa/ics2019/nanos-lite/src/proc.c,16,hello_fun] Hello World from Nanos-lite for the 4824th time!,kernel thread 666
[/home/ubuntu/pa/ics2019/nanos-lite/src/proc.c,16,hello_fun] Hello World from Nanos-lite for the 4825th time!,kernel thread 233
[/home/ubuntu/pa/ics2019/nanos-lite/src/proc.c,16,hello_fun] Hello World from Nanos-lite for the 4825th time!,kernel thread 666
[/home/ubuntu/pa/ics2019/nanos-lite/src/proc.c,16,hello_fun] Hello World from Nanos-lite for the 4826th time!,kernel thread 233
[/home/ubuntu/pa/ics2019/nanos-lite/src/proc.c,16,hello_fun] Hello World from Nanos-lite for the 4826th time!,kernel thread 666
[/home/ubuntu/pa/ics2019/nanos-lite/src/proc.c,16,hello_fun] Hello World from Nanos-lite for the 4827th time!,kernel thread 233
[/home/ubuntu/pa/ics2019/nanos-lite/src/proc.c,16,hello_fun] Hello World from Nanos-lite for the 4827th time!,kernel thread 666
[/home/ubuntu/pa/ics2019/nanos-lite/src/proc.c,16,hello_fun] Hello World from Nanos-lite for the 4828th time!,kernel thread 233
[/home/ubuntu/pa/ics2019/nanos-lite/src/proc.c,16,hello_fun] Hello World from Nanos-lite for the 4828th time!,kernel thread 666
[/home/ubuntu/pa/ics2019/nanos-lite/src/proc.c,16,hello_fun] Hello World from Nanos-lite for the 4829th time!,kernel thread 233
```

实现多道程序系统

我们这一部分的主要目的是实现用户线程

apps里面的 `_start` 来把栈顶位置真正设置到栈指针寄存器中.

在 `context_uload()` 中实现以上内容之后, 我们就可以加载用户进程了. 我们把其中一个 `hello_fun()` 内核线程替换成仙剑奇侠传:

```
context_uload(&pcb[1], "/bin/pal");
```

然后我们还需要在 `serial_write()` , `events_read()` 和 `fb_write()` 的开头调用 `_yield()` , 来模拟设备访问缓慢的情况. 添加之后, 访问设备时就要进行上下文切换, 从而实现多道程序系统的功能.

实现多道程序系统

根据上述内容, 实现多道程序系统. 如果你的实现正确, 你将可以一边运行仙剑奇侠传的同时, 一边输出hello信息.

思考一下, 如何验证仙剑奇侠传确实在使用用户栈而不是内核栈?

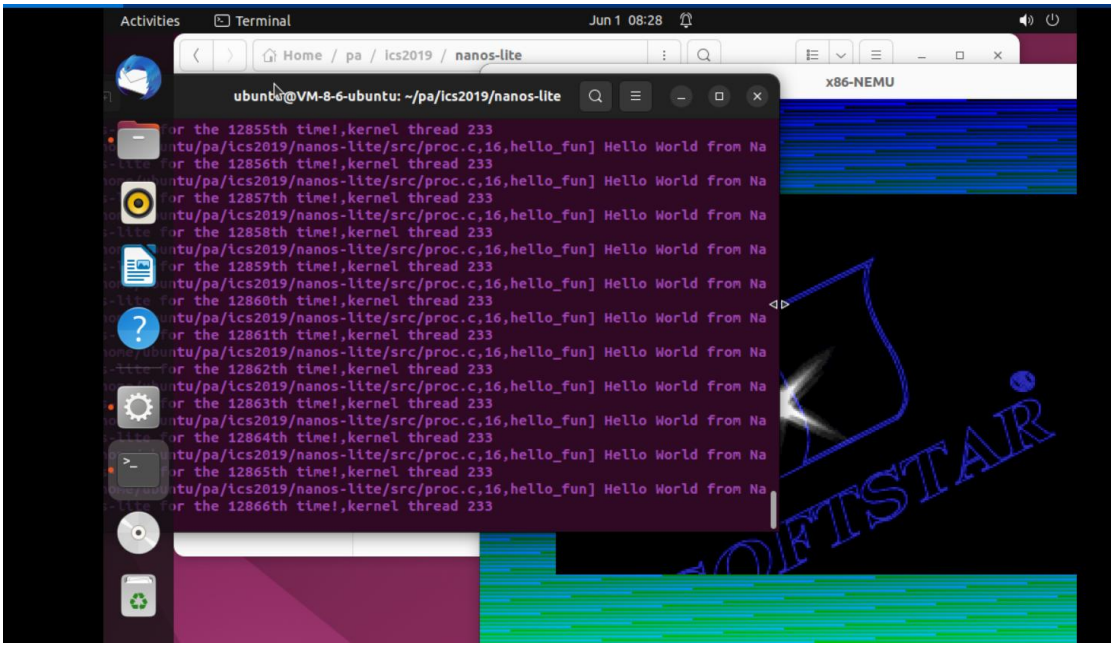
我们修改 ucontext 跟一开始一样，上下文压进去（一开始我们仍然是先不压栈参数）注意我们这里比起系统调用是 -16，因为加载用户进程的时候除了要压栈返回地址，上下文，还得压栈 argc/argv/envp，都是 4 字节（argc 是个 int，其他是指针）

```
_Context *_ucontext(_AddressSpace *as, _Area ustack, _Area kstack, void *entry, void *args) {
  _Context *c=(_Context*)(ustack.end-16-sizeof(_Context));
c->eip=(uintptr_t)entry;
c->cs=8;
c->eflags=0x202;
c->as=as;
return c;
}
```

然后我们在 serial_write,events_read,fb_write 的地方调用_yield 模拟设备访问缓慢的情况，这样一访问设备就上下文切换从而实现多道程序切换

```
size_t fb_write(const void *buf, size_t offset, size_t len) {
    _yield();
```

同时执行 hello_fun 和仙剑，一边输出 hello 一边跑仙剑成功



给用户进程传参

> 根据这一约定, 你还需要修改Navy-apps中 `_start` 的代码, 在其调用 `call_main()` 之前把它的参数设置成 `argc` 的地址. 然后修改 `navy-apps/libs/libc/src/plaform/crt0.c` 中 `call_main()` 的代码, 让它解析出真正的 `argc/argv/envp` , 并调用 `main()` . 这样以后, 用户进程就可以接收到属于它的参数了.

找了半天，ics2019 的_start 直接就在 crt0.c 里。。。然后调用 main 的时候也硬件调用 argc,argv,envp 了。。。。这地方不用改

2019 > navy-apps > libs > libc > src > platform > C crt0.c

```c
#include <stdlib.h>
#include <assert.h>

int main(int argc, char *argv[], char *envp[]);
extern char **environ;

void _start(int argc, char *argv[], char *envp[]) {
  char *env[] = {NULL};
  environ = env;
  exit(main(argc, argv, env));
  assert(0);
}

```

我们按照手册的提示修改 context_uload 的原型并且修改之，让参数能成功传进去了.

> **给用户进程传递参数**
>
> 实现上述内容. 然后修改仙剑奇侠传的少量代码, 如果它接收到一个 `--skip` 参数, 就跳过片头商标动画的播放, 否则不跳过. 商标动画的播放从代码逻辑上具体 `main()` 函数并不远, 于是来RTFSC吧.
>
> 不过为了给用户进程传递参数, 你还需要修改 `context_uload()` 的原型:
>
> ```
> void context_uload(PCB *pcb, const char *filename, char *const argv[], char
> ```
>
> 这样你就可以在 `init_proc()` 中直接给出用户进程的参数来测试了. 目前我们的测试程序中不会用到环境变量, 所以不必传递真实的环境变量字符串. 至于实参应该写什么, 这可是一个C语言的问题, 就交给你来解决吧.

```c
void context_uload(PCB *pcb, const char *filename, int argc, char *const argv[], char *const envp[])
{
  _protect(&pcb->as);
  uintptr_t entry = loader(pcb, filename);

  _Area stack;
  stack.start = pcb->stack;
  stack.end = stack.start + sizeof(pcb->stack);

  pcb->cp = _ucontext(&pcb->as, stack, stack, (void *)entry, argc, argv, envp);
  // Log("%x %x", pcb->cp, pcb->cp->as);
}
```

ucontext 我们也要改一下，把 argc,argv,envp 压栈到对应位置

```c
_Context *_ucontext(_AddressSpace *as, _Area ustack, _Area kstack, void *entry, int argc, char *const argv[], char *const envp[]) {
    memcpy(ustack.end - 12, (void *)(&argc), 4);
  memcpy(ustack.end - 8, (void *)(&argv), 4);
  memcpy(ustack.end - 4, (void *)(&envp), 4);
  _Context *c=(_Context*)(ustack.end-16-sizeof(_Context));
c->eip=(uintptr_t)entry;
c->cs=8;
c->eflags=0x202;
c->as=as;
return c;
}
```

nanos.c | irq.c | proc.c M | loader.c M | am.h | cte.c | vme.c M

ics2019 > nexus-am > am > C am.h

```c
// ================= Virtual Memory Extension (VME) ==================

int _vme_init(void *(*pgalloc)(size_t size), void (*pgfree)(void *));
int _protect(_AddressSpace *as);
void _unprotect(_AddressSpace *as);
int _map(_AddressSpace *as, void *va, void *pa, int prot);
  _Context *_ucontext(_AddressSpace *as, _Area ustack, _Area kstack,
                      void *entry, int argc, char *const argv[], char *const envp[]);

// ================= Multi-Processor Extension (MPE) =================
```

运行之后仙剑还是得等过场，但是输出一下，我们这次已经切实的把参数传进去了，读了下手册，发现

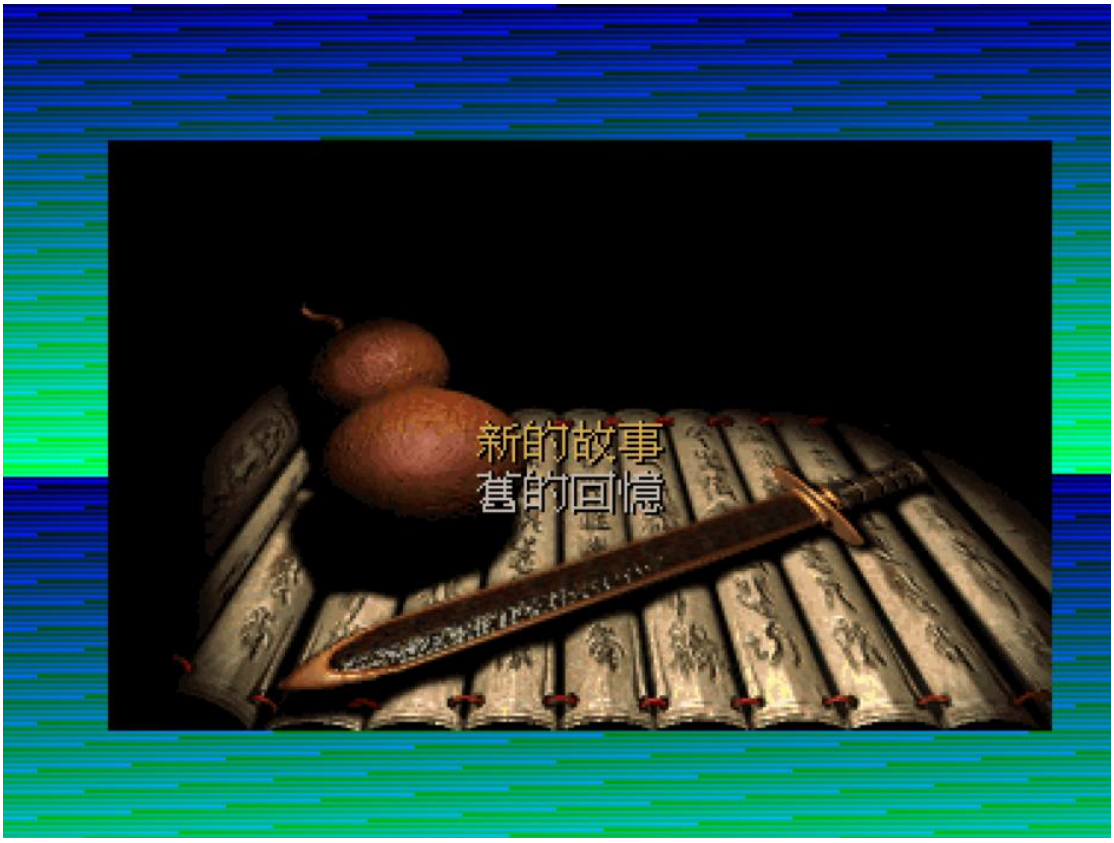
> 实现上述内容. 然后修改仙剑奇侠传的少量代码,

发现仙剑的主函数一开始压根就不接受参数，修改 main-loop 和 main 的参数以及一些判定逻辑

```c
// Show the trademark screen and splash screen
//
if(argc){
    int tf=0;
    for (int i = 0; i < argc; i++)
    {
        if(strcmp(argv[i],"--skip")==0)
        {
            tf=1;
        }
    }
if(!tf)
{
PAL_TrademarkScreen();
PAL_SplashScreen();
}

}
```

```
int
main(int argc, char *argv[]) {
    Log("game start!");
    for (int i = 0; i < argc; i++)
        printf("%s\n", argv[i]);
  hal_init();
    main_loop(argc,argv);

    return 0;
}
```

我们可以成功做到直接跳过开场动画，PA4 第一部分结束



阶段二

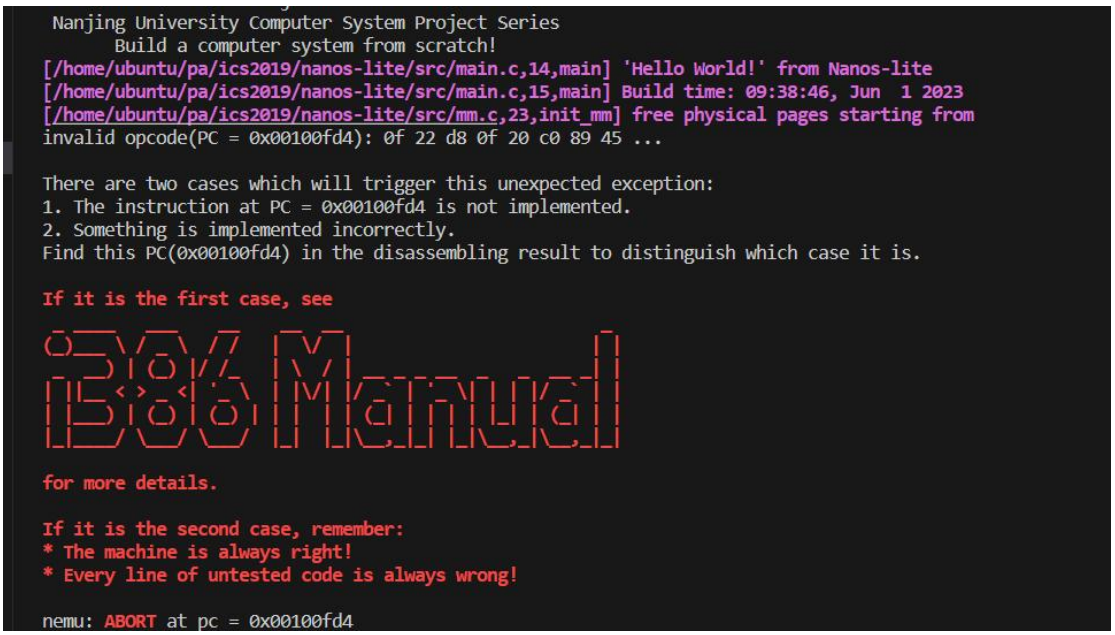
理解分页机制

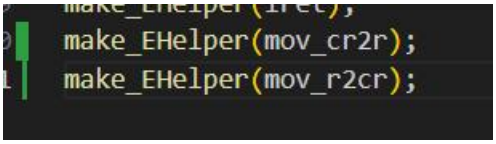
分页机制的意义

现在一些程序加上静态链接等等之后往往会十分的大，可是他一次运行的时候不会一口气用那么多代码，我们其实只需要读一小部分就可以了，这样，我们可以用一个较小的物理空间内存空间，就能跑起来一个远大于我们物理内存的程序，只要通过及时的换出不活跃的部分
事实上，我们需要一种按需分配的虚存管理机制. 之所以分段机制不好实现按需分配，就是因为段的粒度太大了，为了实现这一目标，我们需要反其道而行之：把连续的存储空间分割成小片段，以这些小片段为单位进行组织，分配和管理. 这正是分页机制的核心思想.也是他的目的所在，将进程的各个页离散地存储在内存的任一物理块中，使得从进程的角度看，认为它有一段连续的内存，进程总是从 0 号单元开始编址

在 NEMU 中实现分页机制

打开 HAS_VME，先运行一下，报错，缺指令，查手册，补，需要的是 movcr2r 和 movr2cr 这两个指令，用于对 cr0 和 cr3 进行读写操作，





```
make_EHelper(mov_cr2r);
make_EHelper(mov_r2cr);
```

```
make_EHelper(mov_r2cr) {
  if (id_dest->reg == 0)
    cpu.cr0.val = id_src->val;
  else if (id_dest->reg == 3)
  {
    cpu.cr3.val = id_src->val;
  }
  else
    panic("mov_r2cr error");
  print_asm("movl %%%s,%%cr%d", reg_name(id_src->reg, 4), id_dest->reg);
}

make_EHelper(mov_cr2r) {
  if (id_src->reg == 0)
    operand_write(id_dest, &cpu.cr0.val);
  else if (id_src->reg == 3)
    operand_write(id_dest, &cpu.cr3.val);
  else
    panic("mov_cr2r error");
  print_asm("movl %%cr%d,%%%s", id_src->reg, reg_name(id_dest->reg, 4));

  difftest_skip_ref();
}
```

```
/*2 byte_opcode_table */

  /* 0x00 */  EMPTY, IDEX(gp7_E, gp7), EMPTY, EMPTY,
  /* 0x04 */  EMPTY, EMPTY, EMPTY, EMPTY,
  /* 0x08 */  EMPTY, EMPTY, EMPTY, EMPTY,
  /* 0x0c */  EMPTY, EMPTY, EMPTY, EMPTY,
  /* 0x10 */  EMPTY, EMPTY, EMPTY, EMPTY,
  /* 0x14 */  EMPTY, EMPTY, EMPTY, EMPTY,
  /* 0x18 */  EMPTY, EMPTY, EMPTY, EMPTY,
  /* 0x1c */  EMPTY, EMPTY, EMPTY, EMPTY,
  /* 0x20 */  IDEXW(G2E, mov_cr2r, 4), EMPTY, IDEXW(E2G, mov_r2cr, 4), EMPTY,
  /* 0x24 */  EMPTY, EMPTY, EMPTY, EMPTY
```

此外，我们还需要给 cpu 的寄存器中也加上 cr0 和 cr3

```
src/isa/x86/exec/system.c: In function ‘exec_mov_r2cr’:
src/isa/x86/exec/system.c:15:8: error: ‘CPU_state’ has no member named ‘cr0’
   15 |     cpu.cr0.val = id_src->val;
      |        ^
src/isa/x86/exec/system.c:18:8: error: ‘CPU_state’ has no member named ‘cr3’
   18 |     cpu.cr3.val = id_src->val;
      |        ^
src/isa/x86/exec/system.c: In function ‘exec_mov_r2cr’:
src/isa/x86/exec/system.c:27:32: error: ‘CPU_state’ has no member named ‘cr0’
   27 |     operand_write(id_dest, &cpu.cr0.val);
      |                                ^
src/isa/x86/exec/system.c:29:32: error: ‘CPU_state’ has no member named ‘cr3’
   29 |     operand_write(id_dest, &cpu.cr3.val);
      |                                ^
```

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

isa_vaddr_read，isa_vaddr_write 和 page_translate

接下来按照指导书，修改实现 vaddr_read，vaddr_write 和 page_translate

isa_vaddr_read 与 isa_vaddr_write

两个函数用于对虚拟地址的读写，根据实验指导书他们的功能如下：

```c
uint32_t isa_vaddr_read(vaddr_t addr, int len) {
    if (data cross the page boundary) {
        /* this is a special case, you can handle it later. */
        assert(0);
    }
    else {
        paddr_t paddr = page_translate(addr);
        return paddr_read(paddr, len);
    }
}
```

以 isa_vaddr_read 为例，首先根据 CR0 判断是否开启了分页机制，如果没有开启可以直接调用 paddr_read，如果开启了的话我们先判断是否跨页，如果没跨页的话直接调用 page_translate 将我们的 vaddr 转换为 paddr（这在后面写），然后用给定的 paddr_read 读取内存即可，如果跨页了我们可以在这两个页上分别进行转换+读取的操作，然后把最终结果拼接在一起，write 思路类似，具体代码实现如下：

```c

uint32_t isa_vaddr_read(vaddr_t addr, int len)
{
  if (!cpu.cr0.paging)
    return paddr_read(addr, len);
  uint32_t s_vpage_num = VPAGE_NUM(addr);
  uint32_t e_vpage_num = VPAGE_NUM(addr + len - 1);
  if (s_vpage_num == e_vpage_num)
    return paddr_read(page_translate(addr), len);
  else
  {
    uint32_t pre_len = 4096 - OFF(addr);
    uint32_t suc_len = len - pre_len;
    uint32_t res = 0;
    res |= paddr_read(page_translate(e_vpage_num << PTXSHFT), suc_len);
    res <<= (pre_len << 3);
    res |= paddr_read(page_translate(addr), pre_len);
    return res;
  }
  return paddr_read(addr, len);
}
```

```c
void isa_vaddr_write(vaddr_t addr, uint32_t data, int len)
{
  if (!cpu.cr0.paging)
  {
    paddr_write(addr, data, len);
    return;
  }
  uint32_t s_vpage_num = VPAGE_NUM(addr);
  uint32_t e_vpage_num = VPAGE_NUM(addr + len - 1);
  if (s_vpage_num == e_vpage_num) // 没有出现跨页
    paddr_write(page_translate(addr), data, len);
  else
  {
    uint32_t pre_len = 4096 - OFF(addr);
    uint32_t suc_len = len - pre_len;
    paddr_write(page_translate(addr), data, pre_len);
    paddr_write(page_translate(e_vpage_num << PTXSHFT), (data >> (pre_len << 3)), suc_len);
  }
}
```

page_translate

用于将 vaddr 转换为 paddr
实现如下：

```c
#define PTXSHFT 12 // Offset of PTX in a linear address
#define PDXSHFT 22 // Offset of PDX in a linear address
#define PDX(va) (((uint32_t)(va) >> PDXSHFT) & 0x3ff)
#define PTX(va) (((uint32_t)(va) >> PTXSHFT) & 0x3ff)
#define OFF(va) ((uint32_t)(va)&0xfff)
#define VPAGE_NUM(va) ((uint32_t)(va) >> PTXSHFT)
#define PTE_ADDR(pte) ((uint32_t)(pte) & ~0xfff)
```

```c
paddr_t page_translate(vaddr_t addr)
{
  uint32_t dir_index = PDX(addr);
  uint32_t tab_index = PTX(addr);
  paddr_t pgdir = cpu.cr3.val;
  PDE pde;
  pde.val = paddr_read(pgdir + dir_index * 4, 4);
  if (!pde.present) {
    Log("%x %x", addr, pde.val);
    assert(0);
  }
  PTE pte;
  pte.val = paddr_read((pde.val & ~0xfff) + tab_index * 4, 4);
  if (!pte.present)
    assert(0);
  return (PTE_ADDR(pte.val) | OFF(addr));
}
```

在分页机制上运行用户进程

为此, 我们需要对创建用户进程的过程进行较多的改动. 我们首先需要在加载用户进程之前为其创建地址空间. 由于地址空间是进程相关的, 我们将 `_AddressSpace` 结构体作为PCB的一部分. 这样以后, 我们只需要在 `context_uload()` 的开头调用 `_protect()` , 就可以实现地址空间的创建. 目前这个地址空间除了内核映射之外就没有其它内容了, 具体可以参考 `nexus-am/am/src/$ISA/nemu/src/vme.c` .

不过, 此时 `loader()` 不能直接把用户进程加载到内存位置 `0x40000000` 附近了, 因为这个地址并不在内核的虚拟地址空间中, 内核不能直接访问它. `loader()` 要做的事情是, 获取程序的大小之后, 以页为单位进行加载:

- 申请一页空闲的物理页
- 通过 `_map()` 把这一物理页映射到用户进程的虚拟地址空间中
- 从文件中读入一页的内容到这一物理页上

在 uload 前调用_protect

```
void context_uload(PCB *pcb, const char *filename, int argc, char *const argv[], char *const envp[])
{
  _protect(&pcb->as);
```

完善_map 函数，用于把物理页映射到用户进程的虚拟地址空间

```
int _map(_AddressSpace *as, void *va, void *pa, int prot)
{
  PDE *pgdir = as->ptr;
  uint32_t dir_index = ((uint32_t)va >> 22);
  uint32_t tab_index = (((uint32_t)va >> 12) & 0x3ff);
  PTE *pgtab = (PTE *)(pgdir[dir_index] & ~0xfff);
  if (!(pgdir[dir_index] & 1))
  {
    pgtab = (PTE *)(pgalloc_usr(1));
    pgdir[dir_index] = (PDE)pgtab;
    pgdir[dir_index] |= 1;
  }
  pgtab[tab_index] = (((uint32_t)pa & ~0xfff) | 1);
  return 0;
}
```

完善loader：

```c
#define ROUNDUP(a, sz) ((((uintptr_t)a) + (sz)-1) & ~((sz)-1))
static uintptr_t loader(PCB *pcb, const char *filename)
{
  const int page_size = 4096;
  int fd = fs_open(filename, 0, 0);
  Elf_Ehdr elf_head;
  fs_read(fd, &elf_head, sizeof(Elf_Ehdr));
  Elf_Phdr elf_phentry;
  for (int i = 0; i < elf_head.e_phnum; i++)
  {
    fs_lseek(fd, elf_head.e_phoff + i * elf_head.e_phentsize, SEEK_SET);
    fs_read(fd, &elf_phentry, elf_head.e_phentsize);
    if (elf_phentry.p_type == PT_LOAD)
    {
      int len = 0;
      size_t file_off = elf_phentry.p_offset;
      fs_lseek(fd, file_off, SEEK_SET);
      unsigned char *v_addr = (unsigned char *)elf_phentry.p_vaddr;
      void *p_addr = 0;
      while (len < elf_phentry.p_filesz)
      {
        int mov_size = (elf_phentry.p_filesz - len > page_size ? page_size
                                                               : elf_phentry.p_filesz - len);
        int gap = page_size - ((uint32_t)v_addr & 0xfff);
        if (mov_size > gap)
          mov_size = gap;
        p_addr = new_page(1);
        p_addr = (void *)((uint32_t)p_addr | ((uint32_t)v_addr & 0xfff));
        _map(&pcb->as, v_addr, p_addr, _PROT_EXEC);
        fs_read(fd, p_addr, mov_size);
        v_addr += mov_size;
        p_addr += mov_size;
        len += mov_size;
      }
      int gap = page_size - ((uint32_t)v_addr & 0xfff);
      if (gap != page_size)
      {
        memset(p_addr, 0, gap);
        v_addr += gap;
        len += gap;
      }
      while (len < elf_phentry.p_memsz)
      {
        int mov_size = (elf_phentry.p_memsz - len > page_size ? page_size
                                                              : elf_phentry.p_memsz - len);
        p_addr = new_page(1);
        _map(&pcb->as, v_addr, p_addr, _PROT_EXEC);
        memset(p_addr, 0, mov_size);
        v_addr += mov_size;
        len += mov_size;
      }
      pcb->max_brk = pcb->max_brk > ROUNDUP(v_addr, page_size) ? pcb->max_brk : ROUNDUP(v_addr, page_size);
    }
  }
  fs_close(fd);
  return elf_head.e_entry;
}
```

继续阅读手册

nemu.h), 但你还需要

- 修改 _ucontext() 的实现, 在创建的用户进程上下文中设置地址空间相关的指针 as
- 在 __am_irq_handle() 的开头调用 __am_get_cur_as() (在 nexus-am/am/src/$ISA/nemu/vme.c 中定义), 来将当前的地址空间描述符指针保存到上下文中
- 在 __am_irq_handle() 返回前调用 __am_switch() (nexus-am/am/src/$ISA/nemu/vme.c 中定义) 来切换地址空间, 将调度目标进程的地址空间落实到MMU中

修改_ucontext:

```
_Context *_ucontext(_AddressSpace *as, _Area ustack, _Area kstack, void *entry, int argc, char *const argv[], char *const envp[])
{
  extern void memcpy(void *, const void *, int);
  memcpy(ustack.end - 12, (void *)(&argc), 4);
  memcpy(ustack.end - 8, (void *)(&argv), 4);
  memcpy(ustack.end - 4, (void *)(&envp), 4);

  _Context *new_context = ustack.end - 16 - sizeof(_Context);
  new_context->as = as;
  new_context->eip = (uintptr_t)entry;
  new_context->cs = 8;
  new_context->eflags = 0x202;
  return new_context;
}
```

修改 am_irq_handle，在开头调用 am_get_cur_as 将当前地址空间描述符指针保存到上下文中

```
_Context* __am_irq_handle(_Context *c) {
  _Context *next = c;
  __am_get_cur_as(c);
```

在返回前调用_am_switch()来切换地址空间，将调度目标进程的地址空间落实到 MMU 中

```
  }
}
__am_switch(next);
return next;
```

实现完毕，我们单独运行一手 dummy，成功

```
alization
nemu: HIT GOOD TRAP at pc = 0x00100d56
```

在分页机制上运行仙剑奇侠传

阅读手册，需要实现 mm_brk 实现堆区内存的分配

为了在分页机制上运行仙剑奇侠传, 我们还需要考虑堆区的问题. 之前我们让 `mm_brk()` 函数直接返回 `0` , 表示用户进程的堆区大小修改总是成功, 这是因为在实现分页机制之前, `0x3000000` / `0x83000000` 之上的内存都可以让用户进程自由使用. 现在用户进程运行在分页机制之上, 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, 这样才能保证运行在分页机制上的用户进程可以正确地访问新申请的堆区.

为了识别堆区中的哪些空间是新申请的, 我们还需要记录堆区的位置. 由于每个进程的堆区使用情况是独立的, 我们需要为它们分别维护堆区的位置, 因此我们在PCB中添加成员 `max_brk` , 来记录program break曾经达到的最大位置. 引入 `max_brk` 是为了简化实现: 我们可以不实现堆区的回收功能, 而是只为当前新program break超过 `max_brk` 部分的虚拟地址空间分配物理页.

在分页机制上运行仙剑奇侠传

根据上述内容, 实现 `nanos-lite/src/mm.c` 中的 `mm_brk()` 函数. 你需要注意 `_map()` 参数是否需要按页对齐的问题(这取决于你的 `_map()` 实现).

实现正确后, 仙剑奇侠传就可以正确在分页机制上运行了.

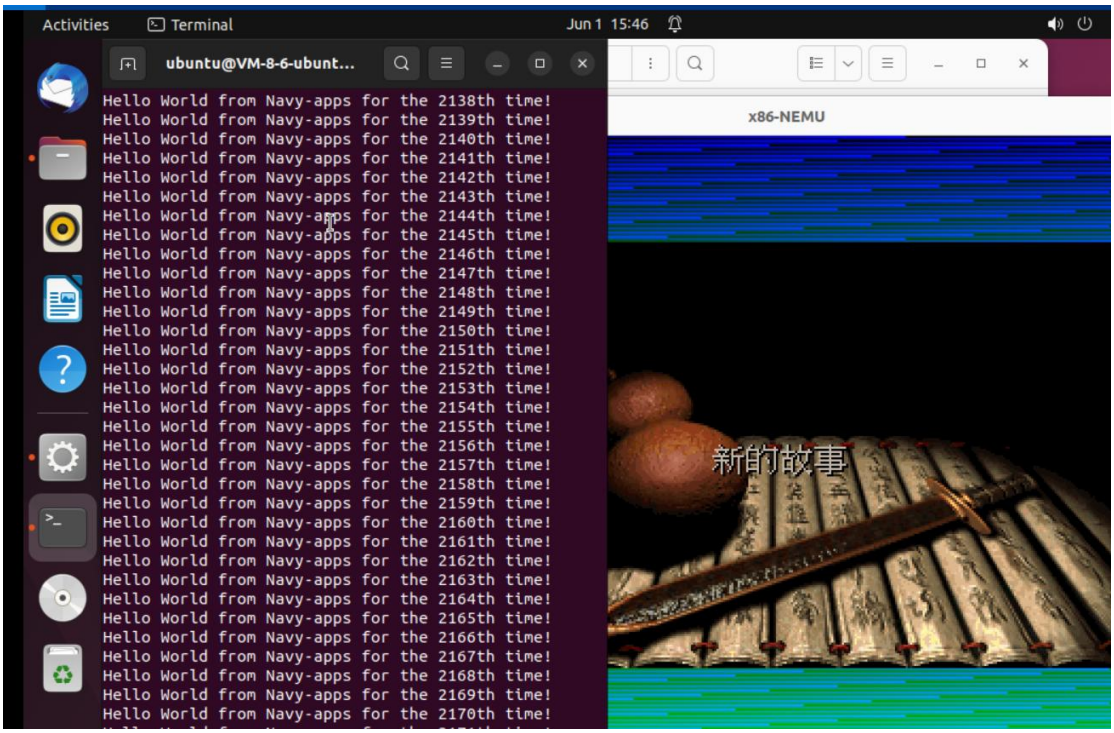
实现 mm_brk，先先判断新的 brk 是否大于当前进程的 max_brk 的值，如果大于，则为 max_brk 到 new_brk 之间的虚拟内存申请物理页并建立映射关系并更新 max_brk。

```
16  /* The brk() system call handler. */
17  int mm_brk(uintptr_t brk, intptr_t increment) {
18  if (brk+increment>current->max_brk)
19  {
20    int new_pgnum=((brk+increment-current->max_brk)+0xfff)/PGSIZE;
21    for (int i=new_pgnum-1;i>=0;--i)
22    {
23      void *pa=new_page(1);
24      _map(&(current->as),(void*)(current->max_brk),pa,1);
25      current->max_brk+=PGSIZE;
26    }
27  }
28  return 0;
29  }
30
```

```
uint32_t sys_brk(uintptr_t brk, intptr_t increment)
{    mm_brk(brk,increment);
     return 0;
}
```

支持虚存管理的多道程序

直接跑一个仙剑一个 hello_fun 来着来回切换，可以看到，成功运行



阶段三

实现抢占多任务

而对于时钟中断的中断号, 不同的ISA有不同的约定. 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/device/intr.c` 中定义). 你需要:

- 在cpu结构体中添加一个 `bool` 成员 `INTR` .
- 在 `dev_raise_intr()` 中将INTR引脚设置为高电平.
- 在 `exec_once()` 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:

```
if (isa_query_intr()) update_pc();
```

- 实现 `isa_query_intr()` 函数(在 `nemu/src/isa/$ISA/intr.c` 中定义):

按照指导书内容，给 cpu 加个 INTR 成员先

```
CR0 cr0;
CR3 cr3;
bool INTR;
```

然后设置成高电平

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

在 exec_once 末尾添加轮询 intr 引脚代码

```
vaddr_t exec_once(void) {
  decinfo.seq_pc = cpu.pc;
  isa_exec(&decinfo.seq_pc);
  update_pc();
  if (isa_query_intr())
    update_pc();

  return decinfo.seq_pc;
}
```

- 实现 `isa_query_intr()` 函数(在 `nemu/src/isa/$ISA/intr.c` 中定义):

```
#define IRQ_TIMER 32          // for x86
#define IRQ_TIMER 0           // for mips32
#define IRQ_TIMER 0x80000005  // for riscv32

bool isa_query_intr(void) {
  if ( ??? ) {
    cpu.INTR = false;
    raise_intr(IRQ_TIMER, ???);
    return true;
  }
  return false;
}
```

- 修改 `raise_intr()` 中的代码, 让处理器进入关中断状态:
  - x86 - 在保存EFLAGS寄存器后, 将其IF位置为 `0`
  - mips32 - 将status.EXL位置为 `1`
    - 你还需要修改eret指令的实现, 将status.EXL置为 `0`
  - riscv32 - 将sstatus.SIE保存到sstatus.SPIE中, 然后将sstatus.SIE位置为 `0`
    - 你还需要修改sret指令的实现, 将sstatus.SPIE还原到sstatus.SIE中, 然后将sstatus.SPIE位置为 `1`

实现 isa_query_intr, 按照手册来就好, 就根据 INTR 和 IF 的状态决定是否要进入关中断状态, 同时把 pc 作为 ret_addr 传参调用 raise_intr。

```
bool isa_query_intr(void) {
  if (cpu.INTR == true && cpu.IF == 1)
  {
    cpu.INTR = false;
    raise_intr(IRQ_TIMER, cpu.pc);
    return true;
  }
  return false;
}
```

然后接着跟着指导书走，修改 raise_intr，让处理器进入关中断

```
2
3   void raise_intr(uint32_t NO, vaddr_t ret_addr) {
4   /* TODO: Trigger an interrupt/exception with ``NO''.
5   * That is, use ``NO'' to index the IDT.
6   */
7   // step1
8   rtl_push(&cpu.eflags.val);
9   rtl_push(&cpu.cs);
10  rtl_push(&ret_addr);
11  cpu.eflags.IF = 0;
12  // step2
13  uint32_t gate_addr = cpu.idtr.base, len = cpu.idtr.limit;
14  if (len <= NO)
15  {
16  printf("the number is larger than the length of IDT!\n");
17  assert(0);
18  }
19  //step3
20  uint32_t val_l, val_h, p;
21  val_l = vaddr_read(gate_addr + NO * 8, 2);
22  val_h = vaddr_read(gate_addr + NO * 8 + 6, 2);
23  p = vaddr_read(gate_addr + NO * 8 + 5, 1) >> 7;
24  //actually no need to check p for NEMU, but we can do it.
25  if (!p)
26  {
27  printf("The gatedesc is not allowed!");
28  assert(0);
29  }
30  //step4
31  //using rtl api
32  vaddr_t goal = (val_h << 16) + val_l;
33  rtl_j(goal);
34  }
```

继续跟着手册走

在软件上, 你还需要:

- 在CTE中添加时钟中断的支持, 将时钟中断打包成 `_EVENT_IRQ_TIMER` 事件.
- Nanos-lite收到 `_EVENT_IRQ_TIMER` 事件之后, 调用 `_yield()` 来强制当前进程让出CPU, 同时也可以去掉我们之前在设备访问中插入的 `_yield()` 了.
- 为了可以让处理器在运行用户进程的时候响应时钟中断, 你还需要修改 `_ucontext()` 的代码, 在构造上下文的时候, 设置正确中断状态, 使得将来返回到用户进程后CPU处于开中断状态.

实现抢占多任务

根据讲义的上述内容, 添加相应的代码来实现抢占式的分时多任务.

为了测试时钟中断确实在工作, 你可以在Nanos-lite收到 `_EVENT_IRQ_TIMER` 事件后用 `Log()` 输出一句话.

硬件中断与基础设施

先把时钟中断打包成_EVNET__IRQ_TIMER

```
_Context* __am_irq_handle(_Context *c) {
  _Context *next = c;
  __am_get_cur_as(c);
  if (user_handler) {
    _Event ev = {0};
    switch (c->irq) {
      case 0x81:
        ev.event = _EVENT_YIELD;
      break;
      case 0x80:
        ev.event = _EVENT_SYSCALL;
        break;
      case 0x20:
        ev.event = _EVENT_IRQ_TIMER;
        break;
      default: ev.event = _EVENT_ERROR; break;
    }
    // printf("irq=%d\n", c->irq);
```

收到此事件调用_yield

```
    break;
    case _EVENT_IRQ_TIMER:
    Log("_EVENT_IRQ_TIMER");
  _yield();
  break;
```

注册 IRQ_TIMER

```
#define IRQ_TIMER 32
```

测试结果：

```
Hello World from Navy-apps for the 273th time!
[/home/ubuntu/pa/ics2019/nanos-lite/src/irq.c,13,do_event] _EVENT_IRQ_TIMER
Hello World from Navy-apps for the 274th time!
[/home/ubuntu/pa/ics2019/nanos-lite/src/irq.c,13,do_event] _EVENT_IRQ_TIMER
Hello World from Navy-apps for the 275th time!
[/home/ubuntu/pa/ics2019/nanos-lite/src/irq.c,13,do_event] _EVENT_IRQ_TIMER
Hello World from Navy-apps for the 276th time!
[/home/ubuntu/pa/ics2019/nanos-lite/src/irq.c,13,do_event] _EVENT_IRQ_TIMER
Hello World from Navy-apps for the 277th time!
[/home/ubuntu/pa/ics2019/nanos-lite/src/irq.c,13,do_event] _EVENT_IRQ_TIMER
Hello World from Navy-apps for the 278th time!
[/home/ubuntu/pa/ics2019/nanos-lite/src/irq.c,13,do_event] _EVENT_IRQ_TIMER
Hello World from Navy-apps for the 279th time!
```

展示你的计算机系统

添加前台程序及其切换功能，展示你亲手创造的计算机系统

```
  context_uload(&pcb[0], "/bin/pal", 2, arg, NULL);
  context_uload(&pcb[1], "/bin/pal", 2, arg, NULL);
  context_uload(&pcb[2], "/bin/pal", 2, arg, NULL);
  context_uload(&pcb[3], "/bin/hello", 1, arg, NULL);
  // context_kload(&pcb[1], hello_fun, "kernel thread 2");
```

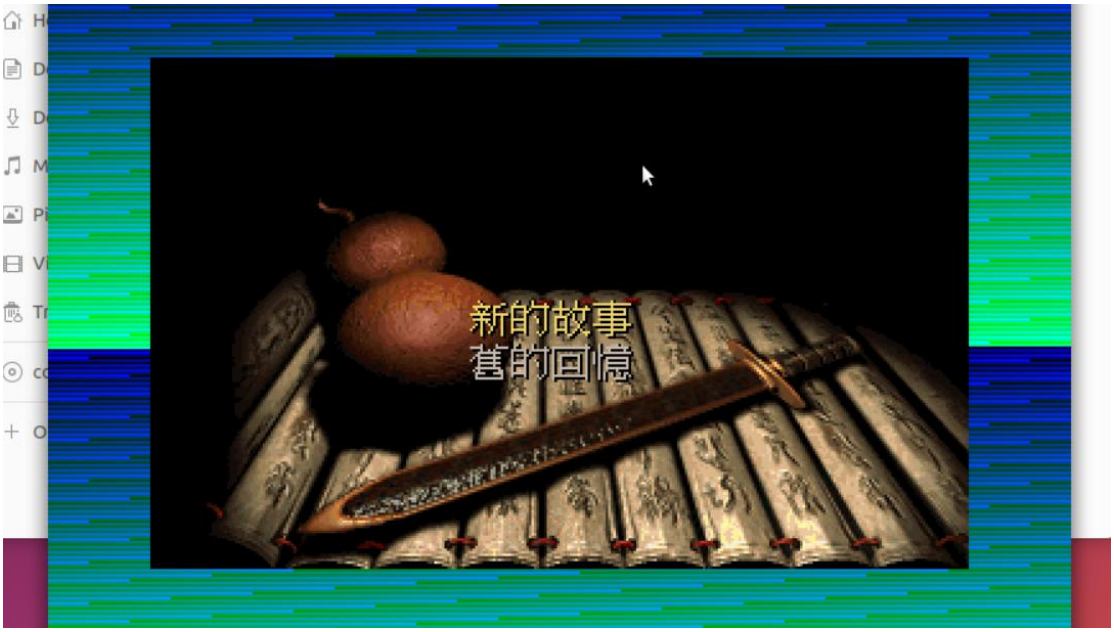
读取按键调用前台程序转换函数

```
//Log("key is%d\n",key);
  if(key != _KEY_NONE)
{     //Log("key_event\n");
      if(key & 0x8000)
    {
          key = key ^ 0x8000;
          change_front_program(key);
          sprintf(buf, "kd %s\n", keyname[key]);
          //Log("buf now is %s\n",buf);
```

如果是 F1 到 F3 就切换，我们用一个 PCB 指针 front_p 来记录我们现在的前台程序

```
66  }
67
68  void change_front_program(int key_code)
69  {
70    if (key_code >= 2 && key_code <= 4)
71    {
72      front_p = &pcb[key_code - 2];
73    }
74    return;
75  }
76
```

效果：后台跑着 hello，前台三个仙剑可以切换




```
Hello World from Navy-apps for the 675th time!
Hello World from Navy-apps for the 676th time!
Hello World from Navy-apps for the 677th time!
Hello World from Navy-apps for the 678th time!
Hello World from Navy-apps for the 679th time!
Hello World from Navy-apps for the 680th time!
Hello World from Navy-apps for the 681th time!
Hello World from Navy-apps for the 682th time!
Hello World from Navy-apps for the 683th time!
Hello World from Navy-apps for the 684th time!
Hello World from Navy-apps for the 685th time!
```

必答题

一些问题

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？

1：i386 是一个 32 位的处理器，但是在实模式下，它只能访问 1MB 的内存，因此只需要 20 位的地址来寻址。这些地址由段基地址和偏移地址组成，其中段基地址是一个 20 位的值，而偏移地址是一个 16 位的值。这些值被组合成一个 20 位的地址，用于访问内存。
至于实模式，则是为了兼容曾经的 8086 模式。
2：页表就是负责逻辑地址转到物理地址的，cpu 需要根据逻辑地址查页表得到物理地址，所以页表本身是存储在物理内存中的，其并不存放虚拟地址和物理地址的对应关系，只存放物理页面的地址，MMU 以虚拟地址为索引去查表返回物理页面地址。因此物理地址是必须的，不能使用虚拟地址，因为其本身就用于从虚拟地址到物理地址的映
射，而非用于虚拟地址到另一个虚拟地址。
3：页表必须连续存放，而且得常驻物理内存，只用一级页表得常驻占用 4M（以 32 位系统为准）连续物理内存

空指针真的是"空"的吗

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?

空指针其实是指向内存地址为 0 的地方，并不是真的空。
当程序对空指针解引用时，计算机内部会将该指针所存储的地址作为访问内存的地址，从该地址中获取数据，但是由于该指针指向的位置我们没有权限访问，这时候就会产生一个内存访问错误。

内核映射的作用

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.

注释后的效果:

```
[src/isa/x86/mmu.c,19,page_translate] 101267 0
x86-nemu: src/isa/x86/mmu.c:20: page_translate: Assertion `0' failed.
make[1]: *** [Makefile:77: run] Aborted (core dumped)
```

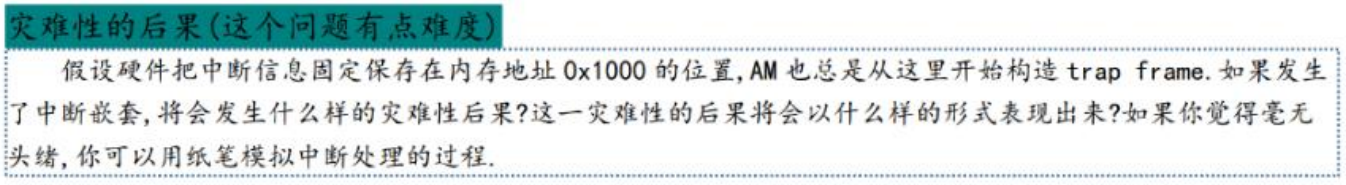
查看反汇编对应位置

```
00101250 <__am_switch>:
  101250: a1 c8 48 c0 01        mov    0x1c048c8,%eax
  101255: 85 c0                 test   %eax,%eax
  101257: 74 17                 je     101270 <__am_switch+0x20>
  101259: 55                    push   %ebp
  10125a: 89 e5                 mov    %esp,%ebp
  10125c: 8b 45 08              mov    0x8(%ebp),%eax
  10125f: 8b 00                 mov    (%eax),%eax
  101261: 8b 50 0c              mov    0xc(%eax),%edx
  101264: 0f 22 da              mov    %edx,%cr3
  101267: a3 c4 48 c0 01        mov    %eax,0x1c048c4
  10126c: 5d                    pop    %ebp
  10126d: c3                    ret
  10126e: 66 90                 xchg   %ax,%ax
  101270: c3                    ret
  101271: 66 90                 xchg   %ax,%ax
  101273: 90                    nop
```

在 __am_switch 函数中刚刚切换过页目录表基地址之后，程序执行便发生了错误。这是由于切换完页表之后，使用用户的页目录表来进行虚拟地址的转换，而上述被注释的代码是用来初始化进程的页目录表的，注释掉之后没有初

始化页目录表造成内核部分的虚拟地址到物理地址的映射缺失。

灾难性的后果

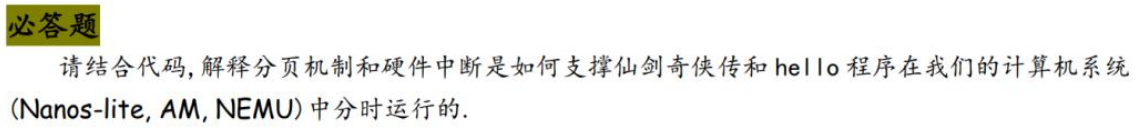


中断嵌套是指中断系统正在执行一个中断服务时，另一个优先级更高的中断发出，这会暂停当前正在执行的级别较低的中断，去处理级别更高的中断，待处理完毕，再返回到被暂停了的中断服务程序继续执行的过程。在中断发生时，其处理过程如下：

1. 寄存器入栈
2. 读取中断向量
3. 更新 SP、链接寄存器，PC

由于 trapframe 和中断信息存放地址相同，如果一旦出现中断嵌套，trap frame 的信息就会被更高优先级的中断信息覆盖，进入中断嵌套，等到结束中断嵌套时由于 trap frame 的信息会被覆盖掉所以会一直卡在嵌套中断的位置处理中断而不继续运行。

分析分时运行



(1) 分页机制保证了不同进程拥有独立的存储空间。为实现多进程分时运行，引入了虚拟内存的概念，让程序链接到固定的虚拟地址的同时，加载都不同的物理位置去执行，因此产生了分页机制。

① 分页机制由 Nanos-lite、AM 和 NEMU 配合实现。

首先，NEMU 提供 CR0 与 CR3 寄存器来辅助实现分页机制，CR0 用于开启分页，CR3 记录页表基地址。随后，MMU 进行分页地址的转换，在代码中表现为 NEMU 的 vaddr_read()与 vaddr_write()。

为启动分页机制，操作系统还需要准备内核页表，这一过程由 Nanos-lite 与 AM 协作实现：Nanos-lite 实现存储管理器的初始化：将 TRM 提供的堆区起始地址作为空闲物理页首地址，并注册物理页的分配函数 new_page()与回收函数 free_page()，调用 AM 的_pte_init()来准备内核页表。_pte_init()函数为 AM 的准备内核页表的基本框架，该函数填写内核的二级页表并设置 CR0 与 CR3 寄存器。

② 分页机制下用户程序的加载

对于仙剑奇侠传与 hello，Nanos-lite 通过 load_prog()实现用户程序的加载。load_prog 通过 AM 中提供的_protect()函数创建虚实地址映射；随后调用 loader()加载程序，加载时根据页面分配函数 new_page()分配新的物理页，利用分页机制将用户程序链接到固定虚拟地址 0x8048000，但加载到 new_page()分配的不同的物理位置去执行；最后 load_prog()通过 umake()函数创建进程的上下文，为进程切换打下基础。

(2) 硬件中断与上下文切换保证程序的分时运行

NEMU 的 exec_wrapper 每执行完一条指令，便查看是否开中断且有硬件中断到来，当触发时钟中断时，将在 AM 中将时钟中断打包成 IRQ_TIME 事件，Nanos-lite 收到该事件后调用 schedule()进行进程调度。schedule()进行进程调度时，通过 AM 的_switch()切换进程的虚拟内存空间，并将进程的上下文传递给 AM，AM 的 asm_trap()恢复这一现场。NEMU 执行下一条指令时，便开始新进程的运行。