

PA4 实验报告

231880394 翟笑晨

January 23, 2025

Contents

1	实验进度描述	2
2	必答题	2
2.1	三. 实验过程	3
2.1.1	PA4.1	3
2.1.2	PA4.2	5
3	实验心得描述	8

1 实验进度描述

在 PA4 的实验中，我按照要求完成了 PA4.1 的内容：创建内核线程上下文、实现线程/进程调度、在 Nanos-lite 中实现上下文切换、创建用户进程上下文、实现带参数的 `SYS_execve`、实现分页机制、并且额外支持访问 `mscratch` 和 `satp` 这两个寄存器。同时能在分页机制上运行仙剑奇侠传，并且试着写了一些抢占多任务的代码（虽然不知道为什么 OJ 没有检测到）。

2 必答题

1. 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。

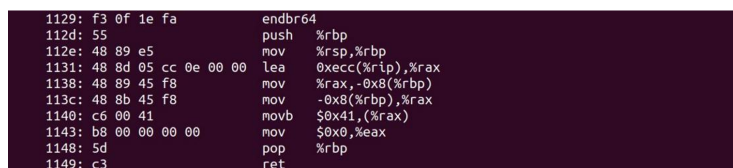
答：详见报告实验过程的 PA4.2 部分的叙述。

2. 理解计算机系统：尝试在 Linux 中编写并运行以下程序：

```
1 #include <stdio.h>
2
3 int main() {
4     char *p = "abc";
5     p[0] = 'A';
6     return 0;
7 }
```

你会看到程序因为往只读字符串进行写入而触发了段错误。请你根据学习的知识和工具，从程序，编译器，链接器，运行时环境，操作系统和硬件等视角分析“字符串的写保护机制是如何实现的”。换句话说，上述程序在执行 `p[0] = 'A'` 的时候，计算机系统究竟发生了什么而引发段错误？计算机系统又是如何保证段错误会发生？如何使用合适的工具来证明你的想法？

答：查看 `objdump` 反汇编之后的结果如下：



```
1129: f3 0f 1e fa      endbr64
112d: 55              push    %rbp
112e: 48 89 e5        mov     %rsp,%rbp
1131: 48 8d 05 cc 0e 00 00 lea     0xccc0e000(%rip),%rax
1138: 48 89 45 f8      mov     %rax,-0x8(%rbp)
113c: 48 8b 45 f8      mov     -0x8(%rbp),%rax
1140: c6 00 41        movb    $0x41,(%rax)
1143: b8 00 00 00 00  mov     $0x0,%eax
1148: 5d              pop     %rbp
1149: c3              ret
```

Figure 1:

(1) 程序视角

程序试图修改字符串字面量的内容，而字符串字面量是常量，存储在只读数据段。在 C 语言中，字符串字面量默认是不可修改的。因此，执行 `p[0] = 'A'`；会触发段错误。

(2) 编译器和链接器视角

编译器会将字符串字面量放入只读数据段（如 `.rodata` 段）。链接器则负责将这些只读段合并到可执行文件中，并分配只读权限。因此，运行时对只读数据段的修改尝试会导致段错误。

(3) 运行时环境视角

程序执行时，操作系统和运行时环境会将代码和数据加载到内存，并为其设置访问权限。字符串字面量通常被标记为只读，因此对它们的写入会触发异常，导致段错误。

(4) 操作系统和硬件视角

操作系统负责将程序加载到内存中，并通过硬件 MMU 为不同的内存区域设置权限。当程序试图写入只读数据段时，MMU 会检测到权限违规，触发异常并导致段错误。

2.1 三. 实验过程

2.1.1 PA4.1

(1) 多道程序 (multi-programming)

两个核心的问题是：

- 在内存中可以同时存在多个进程（我们需要让 loader 把不同进程加载到不同位置即可）。
- 在满足某些条件的情况下，让执行流在进程间切换，即上下文切换。

在多个进程的 CTE 中如何找到别的进程的上下文结构？我们需要一个 `cp` 指针来记录上下文结构的位置。同时，为每个进程维护一个进程控制块 PCB (process control block) 的数据结构。

我们为每个进程创建了一个 32KB 的堆栈，在上下文切换时，把 PCB 中的 `cp` 指针返回给 CTE 的 `__am_irq_handle()` 函数即可。

这里需要理解的是：所谓上下文切换（或者说进程的切换），切换的其实就是栈空间！不同的进程在本质上就是不同的栈空间。或者说，切换的其实是 `cp` 指针。

(2) 内核线程

内核线程是操作系统所属的线程，如 `yield-os`、`nanos-lite` 等。

1. 首先创建内核线程上下文，即在进程的栈上人工创建一个上下文结构。这一过程由 `cte.c` 中的 `kcontext()` 函数进行创建：

```
Context *kcontext(Area kstack, void (*entry)(void *), void *arg);
```

其中：

- `kstack` 是栈的范围。
- `entry` 是内核线程的入口。
- `arg` 是内核线程的参数。

2. 下面进行线程/进程的调度，即 `schedule()` 函数。在 `yield-os` 中，会通过一个 `current` 指针来记录当前正在运行的是哪一进程，指向当前进程的 PCB。
3. 最后是内核线程的参数传递，即通过 `kcontext()` 给 `f()` 传参。在创建内核线程时给出了参数，因此只需要让 `kcontext()` 按照调用约定将参数 `arg` 放置在正确位置，`f()` 就可以读取到正确的参数。

(3) 上下文切换

为了实现上下文切换，需要完成 CTE 中 `kcontext()` 函数的实现，并修改 `__am_asm_trap()` 的逻辑。其流程包括以下几个步骤：

- 在 `yield-os` 中初始化时，调用 `cte_init(schedule)`，将线程调度函数 `schedule()` 注册为异常处理的回调函数。
- 通过以下语句创建内核线程：

```
pcb[0].cp = kcontext((Area){pcb[0].stack, &pcb[0] + 1}, f, (void *)1L);
```

- 在运行时，调用 `yield()` 触发异常进入 `__am_asm_trap()`，处理事件后切换进程，并进入新进程的执行流。

(4) Nanos-lite 中的上下文切换

我们要在 `loader.c` 和 `proc.c` 中实现 `context_kload()` 函数和 `schedule()` 函数。直接仿照 `yield-os` 的 `schedule()` 来做即可。

`context_kload()` 中，我们只需要调用 `kcontext()` 即可，传入的区域为：

```
{&pcb->stack[0], &pcb->stack[STACK_SIZE]}
```

从 `do_event()` 识别事件为 `EVENT_YIELD` 时调用 `schedule()` 即可。

(5) 用户进程

在 PA3 的批处理系统中，`naive_uload()` 直接通过函数调用转移用户程序的代码。但如果栈溢出，可能损坏操作系统的数据。因此我们将栈分为内核栈和用户栈，并完成以下步骤：

1. 在 `abstract-machine/am/src/riscv/nemu/vme.c` 中实现 `ucontext()` 函数。其参数包括：
 - `as`：地址空间。
 - `kstack`：内核栈。
 - `entry`：用户进程的入口。
2. 在 `nanos-lite/src/loader.c` 中实现 `context_uload()` 函数：

```
loader() 得到文件 filename 对应的入口地址 entry,
然后复制 cp 指针, 对 heap.end 写 GPRx,
调用 ucontext() 即可。
```

3. 在 `navy-apps/libs/libos/src/crt0/start.S` 中设置正确的栈指针，确保地址空间与上下文一致。

(6) 用户进程的参数

`nanos-lite` 在加载用户进程时，会将 `argc/argv/envp` 以及字符串参数放在用户栈上，并进行对齐。具体步骤如下：

- 修改 `context_uoload()` 函数，使其接收 `argv` 和 `envp` 参数。
- 逆序拷贝 `argv` 和 `envp` 字符串到栈中。
- 设置 `argc` 的值，并初始化用户上下文。

(7) 带参数的 `execve`

为了支持用户指定进程参数，需修改系统调用 `sys_execve()`，并在其中调用 `context_uoload()`。其中，遇到以下两个问题：

1. 如何在进程 A 中创建用户进程 B：通过 `new_page()` 为用户栈分配内存，并调用 `context_uoload()` 创建新的 PCB。
2. 如何结束进程 A 的执行流：调用 `switch_boot_pcb()` 切换到新进程，随后调用 `yield()` 触发调度。

(8) 运行 `Busybox`

`Busybox` 是一个精简版的 shell 工具集合，通过参数传递不同功能模块进行调用。运行时，为支持 `execvp()`，需遍历 `PATH` 环境变量，确保文件路径解析正确。

2.1.2 PA4.2

(9) 程序和内存位置

在多进程环境下，操作系统需要通过内存管理记录每个程序的分配情况。根据程序代码的特性，可分为以下三种类型：

- 绝对代码：生成固定的内存地址。
- 可重定位代码：加载时调整地址适应新位置。
- 位置无关代码：采用相对寻址，可在任意位置运行。

(10) 虚实交错的魔法

程序的执行过程可分为四个阶段：编译、连接、加载和运行。在编译和连接阶段，绝对代码会生成一个固定的内存地址，确保程序正确执行。而虚拟内存的概念则将程序所看到的地址（虚拟地址）与实际使用的物理地址分开。

虚拟内存地址的转换通过内存管理单元（MMU）实现，主要有两种方法：

- **分段机制**：将虚拟地址和物理地址通过偏移量关联。
- **分页机制**：将内存空间划分为固定大小的小块（页面），通过页表进行映射和管理。

在 PA4.2 中，系统实现了分页机制。分页的核心思想是将虚拟地址划分为以下三部分：

- 高 10 位作为一级页表的虚拟页号。
- 中间 10 位作为二级页表的虚拟页号。
- 低 12 位作为页内偏移量。

(11) 理解分页机制

以下以 riscv32 的两层基数树 (Radix Tree) 为例，解释地址转换过程：

1. 从系统寄存器 `satp` 获取一级页表的基址 `base_1`，通过 `base_1 + VPN[1]` 找到一级页表项地址，并解引用以获取二级页表的基址 `base_0`。
2. 使用 `base_0 + VPN[0]` 获取二级页表项地址，并解引用以获得物理地址的高 20 位。
3. 将物理地址的高 20 位与虚拟地址的低 12 位偏移量组合形成最终的物理地址。

整个过程可以总结为：

`Physical Address = (LeafPTE.PPN × 4096 + VA[11:0])`

(12) TLB (Translation Lookaside Buffer)

为加速地址转换，MMU 通常会采用 TLB 作为缓存机制。TLB 保存最近的页级地址转换结果，当发生 TLB 命中时，直接使用缓存结果；若未命中，则触发页表遍历并更新 TLB。

在 riscv32 架构中，TLB 通常由硬件管理。当 TLB miss 发生时，硬件会自动进行页表遍历并填充结果，因此在 riscv32-nemu 上无需实现 TLB 管理功能。

(13) 将虚拟内存抽象成 VME

需实现以下核心函数：

- `void protect(AddrSpace *as);`：创建默认地址空间。
- `void unprotect(AddrSpace *as);`：销毁指定地址空间。
- `void map(AddrSpace *as, void *va, void *pa, int prot);`：核心映射函数。
- `bool vme_init(void *(*pgalloc_f)(int), void (*pgfree_f)(void *));`：初始化函数。

`map()` 用于建立虚拟地址与物理地址的映射关系。当二级页表不存在时，需调用 `pgalloc_usr()` 创建页表。

(14) 在分页机制上运行 Nanos-lite

Nanos-lite 中定义了存储器管理模块 MM，其主要功能包括：

- 初始化空闲物理页。

- 调用 `vme_init()` 初始化虚拟内存模块。
- 设置恒等映射（虚拟地址等于物理地址）以简化内核空间的地址转换。

通过恒等映射，可验证内核空间地址转换的正确性，例如在 NEMU 中检查：

```
assert(vaddr == paddr);
```

(15) 在 NEMU 中实现分页机制

在 NEMU 中实现分页机制，需要让 `map()` 填写的映射生效。以下是具体步骤：

- 在 `vme_init()` 函数中调用 `set_satp()` 开启 sv32 分页机制。
- 在 `cpu.csr` 系统寄存器中增加一个 `satp` 寄存器，用于存储页目录基址。
- 实现地址转换函数：

```
paddr_t isa_mmu_translate(vaddr_t vaddr, int len, int type);
```

该函数根据虚拟地址通过页表层层解析，最终得到物理地址，同时检查权限和有效性。

- 修改 `vaddr.c` 中的机制，使其根据分页模式决定是否需要地址转换。例如，在 `vaddr_read()` 中判断是否需要调用 `isa_mmu_translate()`：

```
if (isa_mmu_check(vaddr)) {
    vaddr = isa_mmu_translate(vaddr, len, type);
}
```

(16) 在分页机制上运行用户进程

在分页机制上运行用户进程需要解决以下问题：

1. 编译 Navy 应用程序时，需通过 `make` 命令添加 `VME=1` 参数开启虚拟内存支持。开启后，用户进程的链接地址从 `0x40000000` 开始，避免与内核地址空间重叠。
2. 修改 `loader()` 函数，以支持按页加载程序内容到虚拟地址空间。例如：
 - 按页分配物理内存。
 - 调用 `map()` 函数将物理页映射到用户虚拟地址。
 - 处理段偏移和对齐，确保加载的内容正确映射到对应地址。

3. 修改 CTE 的地址空间切换逻辑。在切换进程上下文时，将地址空间描述符指针 `as->ptr` 写入上下文，确保异常调度函数能够正确切换地址空间。

(17) 在分页机制上运行仙剑奇侠传

为了在分页机制上运行仙剑奇侠传，需要实现 `mm_brk()` 函数支持动态内存分配。具体步骤：

- 在 PCB 结构中增加 `max_brk` 成员，表示 `program break` 曾经达到的最大位置。
- 修改系统调用 `_sbrk()` 和 `sys_brk()` 的实现，使其传递两个参数：`program_break` 和 `increment`。
- 当 `program break` 超过 `max_brk` 时，为新增的虚拟地址空间分配物理页。

通过上述实现，仙剑奇侠传可以成功在分页机制的支持下运行，并实现了动态内存分配。

3 实验心得描述

通过对分页机制的学习和实现，我理解了操作系统中的内存管理。以下是实验的主要收获：

- 理解了虚拟内存的概念及其实现细节，包括分页机制、TLB 加速等。
- 学会了在 NEMU 和 Nanos-lite 中实现分页机制，并支持用户进程的加载和运行。
- 掌握了动态内存分配方法，如 `mm_brk()` 的实现。
- 通过调试和优化，使仙剑奇侠传成功运行在分页机制支持的虚拟内存环境中。

在 PA4 中，讲义不再详尽，而是强调关键代码细节，要求我们深入理解系统的工作原理，以便找到实现要求，否则将难以定位相关代码。

debug 问题 在 Nanos-lite 的上下文切换实现中，`hello_fun()` 并未正常输出不同参数，而是持续输出“yield”信息。经过梳理，我发现是在 PA3 阶段，`do_event` 识别到 `yield` 系统调用后直接运行 `halt(0)`，而不是调用 `schedule()` 函数。

致谢 最后非常感谢陈奕睿同学能够和我一起找 bug 的根源并且不断地测试纠错，以及非常感谢助教的耐心和帮助。