

PA1 实验报告

21124001 潘昕田

2022 年 9 月 18 日

1 实验进度

1. PA1 阶段 1 完成
2. PA1 阶段 2 完成
3. PA1 阶段 3 完成

2 必答题

2.1 程序是个状态机

$1 + 2 + \dots + 100$ 的指令序列的状态机可以描述为: $(0, x, x) \rightarrow (1, 0, x) \rightarrow (2, 0, 0) \rightarrow (3, 0, 1) \rightarrow (4, 1, 1) \rightarrow (2, 1, 2) \rightarrow (3, 1, 2) \rightarrow (4, 3, 2) \rightarrow \dots \rightarrow (2, 4851, 98) \rightarrow (3, 4851, 99) \rightarrow (4, 4950, 99) \rightarrow (2, 4950, 99) \rightarrow (3, 4950, 100) \rightarrow (4, 5050, 100) \rightarrow (5, 5050, 100)$

2.2 理解基础设施

$500 * 90\% * 20 * 30 / 60 = 4500min = 75h$ 即将会在调试中花费 75h 的时间.

而如果使用简易调试器, 耗时为 $500 * 90\% * 20 * 10 / 60 = 1500min = 25h$, 将节省 $75 - 25 = 50h$ 调试的时间.

2.3 RTFM-RISCV32

我选择的是 riscv32 指令集

1. riscv32 共有 R,I,S,U,B,J 六种基本整数指令

2. LUI 指令会将 20 位立即数装载到目标寄存器的高 20 位, 并对低 12 位清零
3. 查阅手册可得 mstatus 的寄存器结构 (mstatus 为 riscv 中的 CSR 寄存器 (用于特权架构))

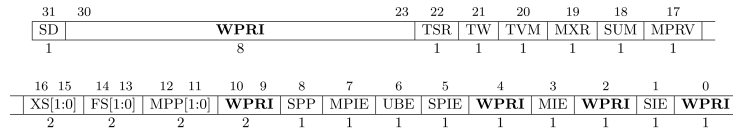


Figure 3.6: Machine-mode status register (mstatus) for RV32.

其中 WPRI 全称为 Reserved Writes Preserve Values, Reads Ignore Values, 即为保留值

xIE: Interrupt Enable in x mode 中断使能

xPIE: Previous Interrupt Enable in x mode 先前的中断使能

xPP: Previous Privilege mode up to x mode 先前的特权级别

2.4 Shell 命令

在/nemu 下使用 `find . -name "*.c" -or -name "*.h" | xargs wc -l` 可以查询到所有的代码行数 (包含空格行), 查询可得 pa1 代码总行数为 24116, 而 pa0 为 23470. 我在 pa1 中编写了 646 行代码.

而使用 `find . -name "*.c" -or -name "*.h" | xargs grep -v "$" | wc -l` 可以查询所有代码行, 但不包括空行, 查询可得 pa1 代码总行数为 20966, 而 pa0 为 20340. 我在 pa1 中编写了 626 行代码, 即有 20 行为新增加的空行数.

在 Makefile 中添加如下的 count 指令:

```

COUNT_TOTAL := $(shell find . -name "*.c" -or -name "*.h" | xargs grep -v "$" | wc -l)
COUNT_NEW := $(shell expr $(COUNT_TOTAL) - 20340)

count:
    @echo $(COUNT_TOTAL) lines in nemu
    @echo $(COUNT_NEW) lines added after pa0

```

在 shell 中执行可得结果为

```

paxintic@paxintic-ThinkPad-P1-Gen-4i:~/ics2022/nemu$ make count
20966 lines in nemu
626 lines added after pa0
paxintic@paxintic-ThinkPad-P1-Gen-4i:~/ics2022/nemu$ 

```

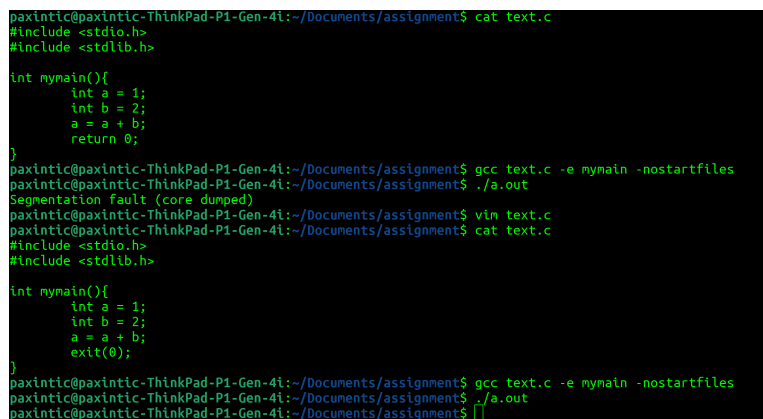
2.5 RTFM-请解释 gcc 中的-Wall 和-Werror 有什么作用？为什么要使用-Wall 和-Werror？

-Wall 选项会打开 gcc 的所有警告选项,-Werror 则会要求 gcc 将所有的警告当作错误处理. 使用这两个选项可以最大程度上将错误在编译期间处理, 而不是等到运行时出错才解决 (这样往往要花费更长的时间才能解决问题)

3 选做题

3.1 需要多费口舌？(选做题)

查阅相关资料, 实际上 C 语言并不是一定要以 main 为程序起始函数. 我们可以利用下图的 gcc 指令来重新定义入口, -e 参数即指明需要调用自己的入口函数, 而-nostartfiles 则也是必须的, 否则程序将会调用 CRT(C Runtime Library), 而其中定义了 __start 入口默认定义为 main, 同时完成对程序的初始化. 而如果通过 CRT 默认初始化程序,main 函数的 return 0 的作用实际上是调用系统函数 exit(0). 若不使用 CRT 初始化而使用自定义的入口, 则必须在函数末尾声明 exit(0). 否则报错.



```
paxintic@paxintic-ThinkPad-P1-Gen-4i:~/Documents/assignment$ cat text.c
#include <stdio.h>
#include <stdlib.h>

int mymain(){
    int a = 1;
    int b = 2;
    a = a + b;
    return 0;
}
paxintic@paxintic-ThinkPad-P1-Gen-4i:~/Documents/assignment$ gcc text.c -e mymain -nostartfiles
paxintic@paxintic-ThinkPad-P1-Gen-4i:~/Documents/assignment$ ./a.out
Segmentation fault (core dumped)
paxintic@paxintic-ThinkPad-P1-Gen-4i:~/Documents/assignment$ vim text.c
paxintic@paxintic-ThinkPad-P1-Gen-4i:~/Documents/assignment$ cat text.c
#include <stdio.h>
#include <stdlib.h>

int mymain(){
    int a = 1;
    int b = 2;
    a = a + b;
    exit(0);
}
paxintic@paxintic-ThinkPad-P1-Gen-4i:~/Documents/assignment$ gcc text.c -e mymain -nostartfiles
paxintic@paxintic-ThinkPad-P1-Gen-4i:~/Documents/assignment$ ./a.out
paxintic@paxintic-ThinkPad-P1-Gen-4i:~/Documents/assignment$
```

p.s. 这里调用 printf 依然会出现 segmentation fault, 初步猜测可能与未使用 CRT 有关

3.2 kconfig 生成的宏与条件编译？

在预编译阶段, 预处理器会将所有的宏按照定义展开

3.3 为什么全部都是函数？

模块化整个程序，能够更好地集中精力在某一功能上，而不是被暂时不需要关注的功能干扰而影响效率

3.4 参数的处理过程

参数应该来自于命令行（根据 `getopt_long()` 函数的介绍）

3.5 谁来指示程序的结束？

实际上 `main` 函数的 `return 0` 在编译时会被 CRT 初始化为 `exit(0)`，程序的结束实际上通过操作系统来实现的，程序本身无法终止自身。

3.6 如何测试字符串处理函数

可以利用 `difftest`，编写一个随机生成的字符串的生成器，将自己实现的函数与库函数处理同一字符串的结果进行比较，测试的样例越多，可靠性越高

3.7 表达式生成器如何获得 C 程序的打印结果？

首先通过 `sprintf` 将 `buf` 里的内容按照 `code_format` 的格式（`code_format` 有 `%s` 格式化符号）输入 `code_buf` 中，然后程序打开（不存在的话创立文件）`code.c` 文件并将 `code_buf` 的内容写入文件中，接着通过系统调用 `gcc` 编译该程序，若编译成功则通过 `popen` 调用二进制文件生成的进程，将进程的输出流重定位到表达式生成器进程的输入流中，再从中读出 `result`，进而生成打印结果。

3.8 除 0 的确切行为

由于加上了 `-Wall, -Werror`，会提前报错并终止执行。

3.9 温故而知新

`static` 声明在声明 `wp_pool` 等变量时的作用在于使其只可在其所在的文件内可见，外部文件的函数等都不可使用，类似于面向对象中类的 `private` 成员。

3.10 一点也不能长?

1 个字节是 x86 中最短的指令, 如果 int3 有 2 个或以上的字节, 那么在插入断点时就可能污染其它指令 (x86 插入断点是通过将指令的第一个字节替换为 0xCC 实现的). 导致整个进程被破坏.

3.11 随心所欲的断点

可能会导致程序错误地被解读为其它指令, 事实上,ROP 攻击的实现就是通过把不同指令的某一段连接起来解读为 ret 指令从而发动攻击 (参考 CMU 的 CSAPP 课程介绍)

3.12 NEMU 的前世今生

Debugger 实际上是用一个计算机程序去测试并调试另一个程序, 而为了提供强大的测试功能,Debugger 会使用 instruction set simulator (ISS)(不过因为效率的考量可能只会模拟部分的指令) 而这一点和 Emulator 具有相似性, 不过 Emulator 的主要是用于在一个计算机系统上模拟另一个计算机系统, 因此会有相较于 Debugger 更多的功能. 同时可以在更加接近底层的位置 Debug.

4 实验心得

4.1 表达式生成器程序的编写

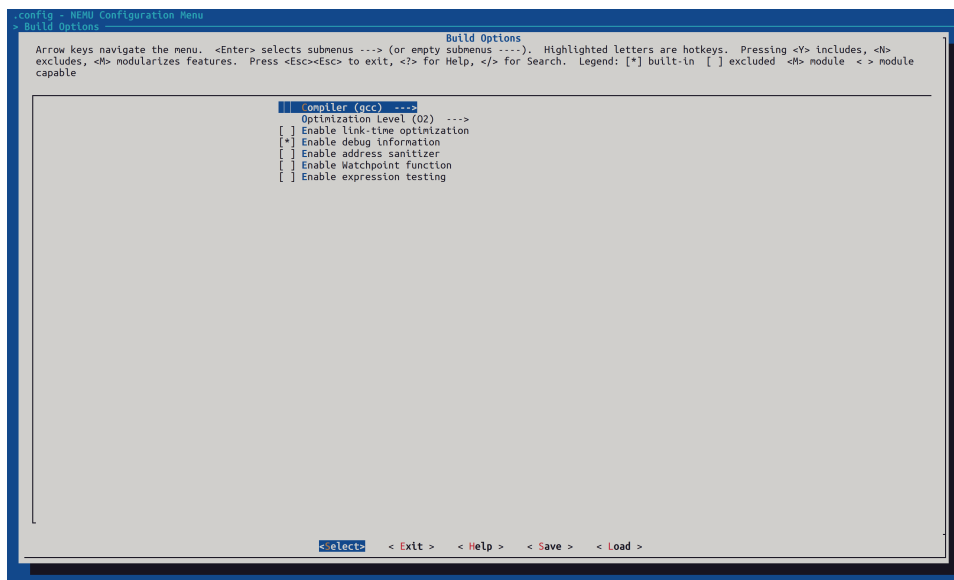
表达式生成器其实是整个 PA1 我最后一个完成的部分, 由于 C 对字符串无法使用 + 操作符 (没有 String 类的痛), 虽然伪代码比较简单, 但由于 C 的原因实现非常困难. 所以在这里花了很长时间来完成. 最后在修改 nemu-main.c 时还因为将测试放置在 init_monitor 之前导致 Segmentation Fault(找了半天没找到 Bug, 一开始因为是字符串越界, 但调用库函数并不会出错, 后来发现原先的读入无法读入空格, 修改发现仍然出错, 最后定位到 expr 才解决问题)

4.2 Menuconfig

PA1.3 要求在 menuconfig 中定义 CONFIG_WATCHPOINT 宏, 通过其开关来决定是否启用监视点. 后来我又自己增加了 CONFIG_EXPRTEST

宏用于确定是否在 nemu-main.c 测试相关程序.

具体声明位置在 menuconfig 的界面如下:



其被声明在了 Build Options(可从 Kconfig 中看出) 中. 其中若禁用监视点, 则 info w, w 等与监视点相关的指令也一并无法执行.(执行时会提示没有此命令)

4.3 总结

完成 PA1 的过程中, 一方面感受到了编程水平的提升, 另一方面也对大项目的组织有了更加清晰的了解, 同时也加深了自己对计算机系统底层架构的理解, 希望能够将 PA 坚持做下去, 收获更多的知识.