

实验一（PA1） 简易调试器

实验简介(请认真阅读以下内容，若有违反，后果自负)

预计平均耗时/代码量: 30 小时/约 400 行

实验内容: 本次实验包含如下三个阶段:

- 阶段 1: 实现“单步、打印寄存器状态、扫描内存”三个调试功能
- 阶段 2: 实现调试功能的表达式求值
- 阶段 3: 实现监视点

提交说明: 见[这里](#)


评分依据: 代码实现占 70%，实验报告占 30%。其中，对于代码实现中的必做任务完成阶段 1 占 50%，阶段 2 占 30%，阶段 3 占 10%，剩下 10%为选做任务。


进行本实验前，请在工程目录下执行以下命令进行分支整理，否则将影响成绩：


```
git commit --allow-empty -am "before starting PA1"
```


```
git checkout -b PA1
```


阅读说明：

 普通阅读，无需提交。

 必做任务，需要提交

 选做任务，可以不提交

 思考题，需要提交

 实验简介，请仔细阅读。

NEMU 概述

本实践课程所要设计的“NEMU”一款经过简化的 x86 全系统模拟器。那什么是模拟器呢？

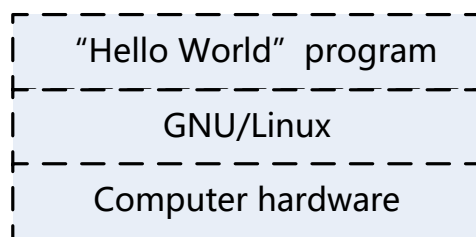
你小时候应该玩过红白机，超级玛丽，坦克大战，魂斗罗... 它们的画面是否让你记忆犹新？（希望我们之间没有代沟...）随着时代的发展，你已经很难在市场上看到红白机的身影了。当你正在为此感到苦恼的时候，模拟器的横空出世唤醒了你心中尘封已久的童年回忆。红白机模拟器可以为你模拟出红白机的所有功能，有了它，你就好像有了一个真正的红白机，可以玩你最喜欢的红白机游戏（这里是一个小型红白机模拟器 [LiteNES](#)）。你可以在如今这个红白机难以寻觅的时代，再次回味你儿时的快乐时光，这实在是太神奇了！

你被计算机强大的能力征服了，你不禁思考，这到底是怎么做到的？你已经学习了诸如程序设计等很多计算机专业课，但仍然找不到你想要的答案。但你可以肯定的是，红白机模拟器只是一个普通的程序，因为你还是需要像运行 Hello World 程序那样运行它。但同时你又觉得，红白机模拟器又不像一个普通的程序，它究竟是怎么模拟出一个红白机的世界，让红白机游戏在这个世界中运行的呢？

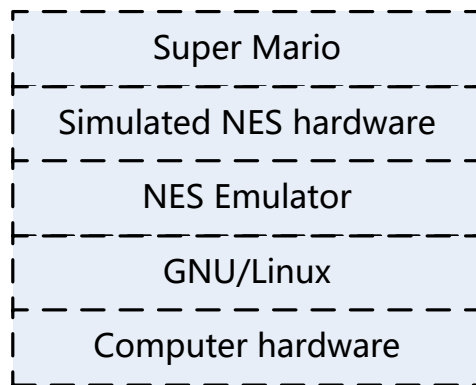
事实上，NEMU 就是在做类似的事情！它模拟了一个 x86-32 (IA32) 的世界（准确地说，是 x86-32 的一个子集），你可以在这个 x86-32 世界中执行程序。换句话说，你将要编写一个用来**执行其它程序的程序**！为了更好地理解 NEMU 的功能，下面将

- 在 GNU/Linux 中运行 Hello World 程序
- 在 GNU/Linux 中通过红白机模拟器玩超级玛丽
- 在 GNU/Linux 中通过 NEMU 运行 Hello World 程序

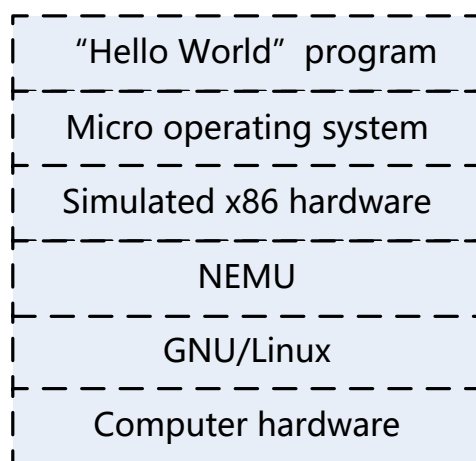
这三种情况进行比较。



上图展示了“在 GNU/Linux 中运行 Hello World 程序”的情况。GNU/Linux 操作系统直接运行在计算机硬件上，对计算机底层硬件进行了抽象，同时向上层的用户程序提供接口和服务。HelloWorld 程序输出信息的时候，需要用到操作系统提供的接口，因此 Hello World 程序并不是直接运行在计算机硬件上，而是运行在操作系统（在这里是 GNU/Linux）上。



上图展示了"在 GNU/Linux 中通过红白机模拟器玩超级玛丽"的情况。在 GNU/Linux 看来, 运行在其上的红白机模拟器 NES Emulator 和上面提到的 Hello World 程序一样, 都只不过是一个用户程序而已。神奇的是, 红白机模拟器的功能是负责模拟出一套完整的红白机硬件, 让超级玛丽可以在其上运行。事实上, 对于超级玛丽来说, 它并不能区分自己是运行在真实的红白机硬件之上, 还是运行在模拟出来的红白机硬件之上, 这正是"虚拟化"的魔术。



上图展示了"在 GNU/Linux 中通过 NEMU 执行 Hello World 程序"的情况。在 GNU/Linux 看来, 运行在其上的 NEMU 和上面提到的 Hello World 程序一样, 都只不过是一个用户程序而已。但 NEMU 的功能是负责模拟出一套 x86 硬件。让程序可以在其上运行。不过, 我们还需要先在模拟出的 x86 硬件之上运行一个微型操作系统。之后才让 Hello World 程序在这个微型操作系统上面运行。为了方便叙述, 我们将在 NEMU 中运行的程序称为"用户程序"。

要虚拟出一个计算机系统并没有你想象中的那么困难。我们可以把计算机看成由若干个硬件部件组成, 这些部件之间相互协助, 完成"运行程序"这件事情。在 NEMU 中, 每一个硬件部件都由一个 C 语言的数据对象来模拟, 例如变量, 数组, 结构体等; 而对这些部件的操作则通过对相应数据对象的操作来模拟。例如 NEMU 中使用结构体来模拟通用寄存器, 那么对这个结构体进行读写则相当于

对通用寄存器进行读写。这些数据对象之间相互协助的威力会让你感到吃惊! NEMU 不仅仅能运行 Hello World 这样的小程序, 还可以在 NEMU 中运行仙剑奇侠传 (很酷!). 完成 NEMU 之后, 你对程序的认识会被彻底颠覆, 你会觉得红白机模拟器不再是一件神奇的玩意儿, 甚至你会发现编写一个属于自己的红白机模拟器不再是遥不可及!

让我们来开始这段激动人心的旅程吧! 但请不要忘记以下原则:

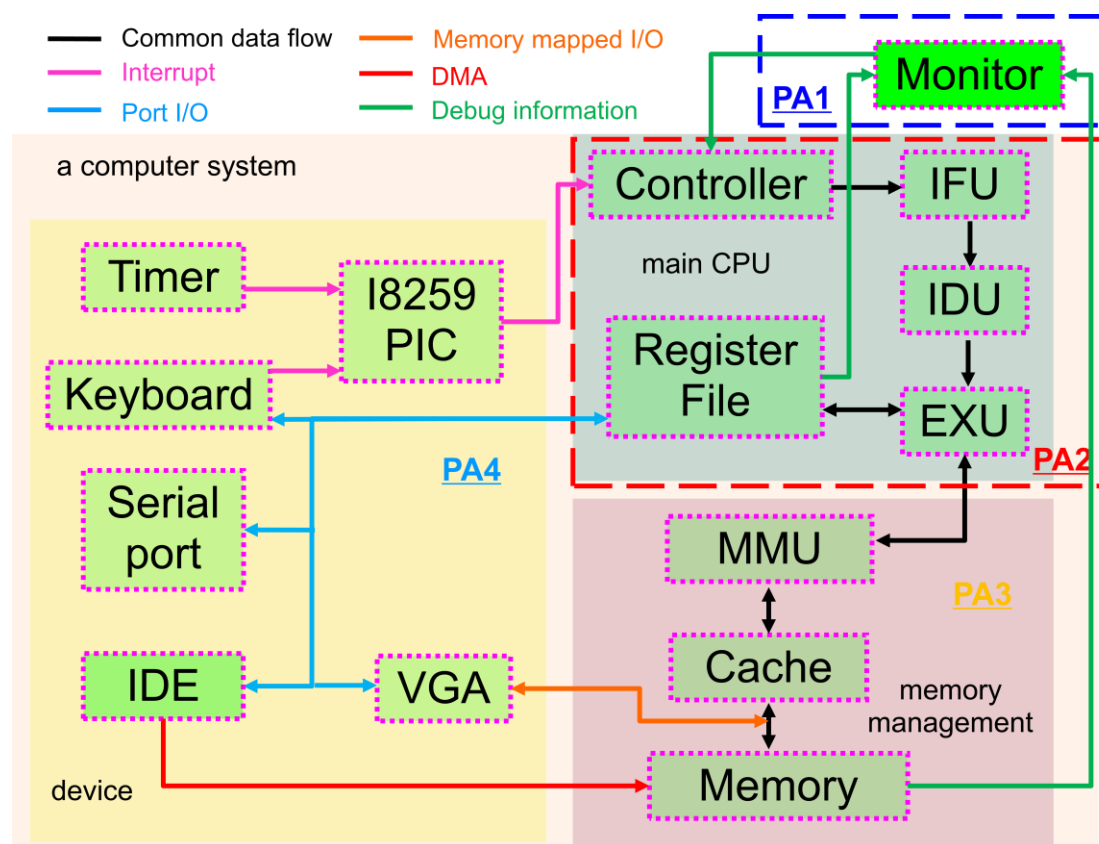
- 机器永远是对的
- 未测试代码永远是错的
- 反复阅读源代码和技术手册 (man、intel 手册等)

阅读源代码

拿到 NEMU 框架代码, 第一件事就是阅读源代码。不过框架代码内容众多, 其中包含了很多在后续阶段中才使用的代码, 随着实验进度的推进, 我们会逐渐解释所有的代码。因此在阅读的时候, 你只需要关心和当前进度相关的模块就可以了, 不要纠缠于和当前进度无关代码, 否则会给你的心灵带来不必要的恐惧。



目前我们只需要关心 NEMU 的内容, 其它内容会在将来进行介绍。下图给出了 NEMU 的结构。



NEMU 主要由 4 个模块构成: monitor、CPU、存储管理和设备, 它们依次

作为 4 个 NEMU 关注的主题。其中，CPU、存储管理和设备这 3 个模块共同组成一个虚拟的计算机系统，程序可以在其上运行；monitor 位于这个虚拟计算机系统之外，主要用于监视这个虚拟计算机系统是否正确运行。monitor 虽然不属于虚拟计算机系统，但对 NEMU 来说，它是必要的。它除了负责与 GNU/Linux 进行交互(例如读写文件)之外，还带有调试器的功能，为 NEMU 的调试提供了方便的途径。缺少 monitor 模块，对 NEMU 的调试将会变得十分困难。

代码中 **nemu** 目录下的源文件组织如下(部分目录下的文件并未列出)：

```
nemu
├── include                                # 存放全局使用的头文件
│   ├── common.h                        # 公用的头文件
│   ├── cpu
│   │   ├── decode                    # 译码相关
│   │   ├── exec                      # 执行相关
│   │   └── reg.h                    # 寄存器结构体的定义
│   ├── debug.h                      # 一些方便调试用的宏
│   ├── device                      # 设备相关
│   ├── macro.h                      # 一些方便的宏定义
│   ├── memory
│   │   └── memory.h                  # 访问内存相关
│   ├── misc.h                      # 杂项
│   ├── monitor
│   │   ├── monitor.h
│   │   └── watchpoint.h            # 监视点相关
│   └── nemu.h
├── Makefile.part                      # 指示NEMU的编译和链接
├── src                                # 源文件
│   ├── cpu
│   │   ├── decode                    # 译码相关
│   │   ├── exec                      # 执行相关
│   │   └── reg.c                    # 寄存器相关
│   ├── device                      # 设备相关
│   ├── lib
│   │   └── logo.c                    # "i386"的logo
│   ├── main.c                      # 你知道的...
│   ├── memory
│   │   ├── burst.h
│   │   ├── dram.c                  # DRAM工作方式的模拟
│   │   └── memory.c                # 访问内存的接口函数
│   └── monitor
│       ├── cpu-exec.c                # 指令执行的主循环
│       ├── debug                    # 简易调试器相关
│       │   ├── elf.c                # ELF文件格式的解析
│       │   ├── expr.c              # 表达式求值的实现
│       │   ├── ui.c                 # 用户界面相关
│       │   └── watchpoint.c         # 监视点的实现
│       └── monitor.c
```

为了给出一份可以运行的框架代码，代码中完整实现了 **mov** 指令的功能(部分特殊的 **mov** 指令并未实现，例如 **mov %eax, %cr3**)，并附带一个 **mov** 指令的

用户程序(testcase/src/mov.S)。另外，部分代码中会涉及一些硬件细节(例如 nemu/src/cpu/decode/modrm.c) 和 文件格式 (例如 nemu/src/monitor/debug/elf.c)。在你第一次阅读代码的时候，你需要尽快掌握 NEMU 的框架，而不要纠缠于这些细节。随着实验任务的进行，你会反复回过头来探究这些细节。大致了解上述的目录树之后，你就可以开始阅读代码了，至于从哪里开始，就不用多费口舌了吧。

需要多费口舌吗？

嗯... 如果你觉得提示还不够，那就来一个劲爆的：回忆程序设计课的内容，一个程序从哪里开始执行呢？

如果你不屑于回答这个问题，不妨先冷静下来。其实这是一个值得探究的问题，你会在将来重新审视它。

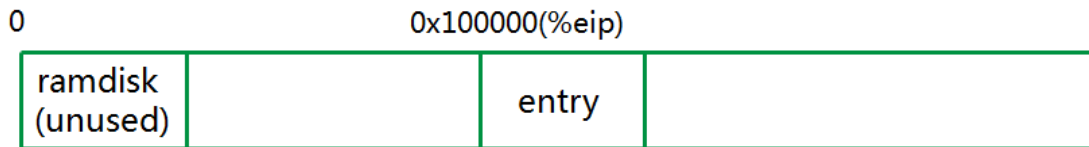
NEMU 开始执行的时候，会进行一些和 monitor 相关的初始化工作，包括打开日志文件，读入 ELF 文件的符号表和字符串表，编译正则表达式，初始化监视点结构池。这些初始化工作你几乎一个也看不懂，但不要紧，因为你现在根本不必关心它们的细节，因此可以继续阅读代码。之后代码会对寄存器结构的实现进行测试，测试通过后会调用 `restart()` 函数(在 `nemu/src/monitor/monitor.c` 中定义)，它模拟了"计算机启动"的功能，主要是进行一些和"计算机启动"相关的初始化工作，包括

- 初始化 ramdisk
- 读入入口代码 entry
- 设置 `%eip` 的初值
- 初始化 DRAM 的模拟(目前不必关心)

在一个完整的计算机中，程序的可执行文件应该存放在磁盘里，但目前我们并没有实现磁盘的模拟，因此 NEMU 先把内存开始的位置附近的一段区间作为磁盘来使用，这样的磁盘有一个专门的名称，叫 **ramdisk**。目前的 ramdisk 只用于存放将要在 NEMU 中运行的程序的可执行文件，这个文件是运行 NEMU 的一个参数，在运行 NEMU 的命令中指定，`init_ramdisk()` 函数把这个文件从真实磁盘读入到 ramdisk。

入口代码 entry 的引入其实是一种简化。我们知道内存是一种 RAM，是一种易失性的存储介质，这意味着计算机刚启动的时候，内存中的数据都是无意义的；而 BIOS 是固化在 ROM 中的，它是一种非易失性的存储介质，BIOS 中的内容不会因为断电而丢失。因此在真实的计算机系统中，计算机启动后首先会把控制权交给 BIOS，BIOS 经过一系列初始化工作之后，再从磁盘中将有意义的

程序读入内存中执行。对这个过程的模拟需要了解很多超出本课程范围的细节, 我们在这里做了简化, 让 monitor 直接把一个有意义的程序 entry 读入到一个固定的内存位置 0x100000, 并把这个内存位置作为 %eip 的初值。这时内存的布局如下:



从 0 开始的一段物理内存被当作 ramdisk 来使用, 但 PA1 中在 NEMU 中运行的程序并不需要使用 ramdisk, 因此这段区间目前暂时不使用。从 0x100000 开始的物理内存用于存放 entry, 现在 entry 的内容就是将要在 NEMU 中运行的程序, NEMU 的模拟执行将从这里开始。在 PA2 中, 我们将会把一个操作系统微内核 (kernel) 作为 entry, kernel 负责从 ramdisk 中读出将要运行的程序, 并把它加载到正确的内存位置。

restart() 函数执行完毕后, NEMU 会进入用户界面主循环 ui_mainloop() (在 nemu/src/monitor/debug/ui.c 中定义), 代码已经实现了几个简单的命令, 它们的功能和 GDB 是很类似的。键入 c 之后, NEMU 开始进入指令执行的主循环 cpu_exec() (在 nemu/src/monitor/cpu-exec.c 中定义)。

cpu_exec() 模拟了 CPU 的工作方式: 不断执行指令。exec() 函数 (在 nemu/src/cpu/exec/exec.c 中定义) 的功能是让 CPU 执行一条指令。已经执行的指令会输出到日志文件 log.txt 中, 你可以打开 log.txt 来查看它们。

执行指令的相关代码在 nemu/src/cpu/exec 目录下, 其中一个重要的部分是定义在 nemu/src/cpu/exec/exec.c 文件中的 opcode_table 数组, 在这个数组中, 你可以看到框架代码中都已经实现了哪些指令, 其中 inv 的含义是 invalid, 代表对应的指令还没有实现 (也可能是 x86-32 中不存在该指令)。在以后的 PA 中, 随着你实现越来越多的指令, 这个数组会逐渐被它们代替。关于指令执行的详细解释和 exec() 相关的内容需要涉及很多细节, 目前你不必关心, 我们将会在 PA2 中进行解释。

思考题 1: 温故而知新

opcode_table 到底是一个什么类型的数组?

如果你感到困惑, 那么说明你需要马上复习程序设计的知识了。[这里](#)有一份十分优秀的 C 语言入门教程, 如果你觉得你的程序设计知识比较生疏, 请你务必阅读它。

NEMU 不断执行指令,直到遇到以下情况之一,才会退出指令执行的循环:

- 达到要求的循环次数。
- 用户程序执行了 `nemu_trap` 指令。这是一条特殊的指令,机器码为 `0xd6`。x86-32 中并没有这条指令,它是为了指示程序的结束而加入的。在后续的实验中,我们还会使用这条指令实现一些无法通过程序本身完成的,需要 NEMU 帮助的功能。

退出 `cpu_exec()` 之后, NEMU 将返回到 `ui_mainloop()`, 等待用户输入命令。但为了再次运行程序,你需要退出 NEMU, 然后重新运行。

思考题 2: 究竟要执行多久?

在 `cmd_c()` 函数中, 调用 `cpu_exec()` 的时候传入了参数 `-1`, 你知道为什么吗?

谁来指示程序的结束?

在程序设计课上老师告诉你, 当程序执行到 `main()` 函数返回处的时候, 程序就退出了, 你对此深信不疑。但你是否怀疑过, 凭什么程序执行到 `main()` 函数的返回处就结束了? 如果有人告诉你, 程序设计课上老师的说法是错的, 你有办法来证明/反驳吗? 如果你对此感兴趣, 请在互联网上搜索相关内容。

最后我们聊聊代码中一些值得注意的地方。

- 三个对调试有用的宏(在 `nemu/include/debug.h` 中定义)
 - ◆ `Log()` 是 `printf()` 的升级版, 专门用来输出调试信息, 同时还会输出使用 `Log()` 所在的源文件, 行号和函数, 当输出的调试信息过多的时候, 可以很方便地定位到代码中的相关位置。
 - ◆ `Assert()` 是 `assert()` 的升级版, 当测试条件为假时, 在 **assertion fail** 之前可以输出一些信息。
 - ◆ `panic()` 用于输出信息并结束程序, 相当于无条件的 **assertion fail**。

代码中已经给出了这三个宏的例子, 如果你不知道如何使用它们, 阅读源代码。

- 访问模拟的内存
 - ◆ 在程序运行的过程中, 总是使用 `swaddr_read()` 和 `swaddr_write()` 访问

模拟的内存。 **swaddr, lnaddr, hwaddr** 分别代表虚拟地址, 线性地址, 物理地址, 这些概念将在 PA3 中用到, 但从现在开始保持接口的一致性可以在将来避免一些不必要的麻烦。

大致弄清楚 NEMU 工作方式之后, 你就可以开始做 PA1 了。需要注意的是, 上面描述的只是一个十分大概的过程, 如果你对这个过程有疑问, **阅读源代码**。

理解框架代码

你需要结合上述文字理解 NEMU 的框架代码。需要注意的是, 阅读代码也是有技巧的, 如果你分开阅读框架代码和上述文字, 你可能会觉得阅读之后没有任何效果, 因此, 你需要一边阅读上述文字, 一边阅读相应的框架代码。

如果你不知道"怎么才算是看懂了框架代码", 你可以先尝试进行后面的任务, 如果发现不知道如何下手, 再回来仔细阅读这一页面。**理解框架代码是一个螺旋上升的过程**, 不同的阶段有不同的重点, 你不必因为看不懂某些细节而感到沮丧, **更不要试图一次把所有代码全部看明白**。

第三视点: 简易调试器

简易调试器是 monitor 的一项重要功能。我们知道 NEMU 是一个用来执行其它用户程序的程序, 这意味着, NEMU 可以随时了解用户程序执行的所有信息。然而这些信息对外面的调试器(例如 GDB)来说, 是不容易获取的。例如在通过 GDB 调式 NEMU 的时候, 你将很难在 NEMU 中运行的用户程序中设置断点, 但对于 NEMU 来说, 这是一件不太困难的事情。

我们需要在 monitor 中实现一个具有如下功能的简易调试器(相关部分的代码在 `nemu/src/monitor/debug` 目录下), 如果你不清楚命令的格式和功能, 请参考如下表格:

命令	格式	使用举例	说明
帮助(1)	help	help	打印命令的帮助信息
继续运行(1)	c	c	继续运行暂停的程序
退出(1)	q	q	退出 NEMU
单步执行	si [N]	si 10	程序单步执行 N 条指令后暂停, 当 N 没有给出时, 缺省为 1。
打印程序状态	info SUBCMD	info r info w	打印寄存器状态 打印监视点信息
表达式求值	p EXPR	p \$eax + 1	求出表达式 EXPR 的值, EXPR 支持的运算请见 调试中的表达式求值 小节。
扫描内存(2)	x N EXPR	x 10 \$esp	求出表达式 EXPR 值, 将结果作为起始内存地址, 以 16 进制形式输出连续的 N 个 4 字节。
设置监视点	w EXPR	w *0x2000	当表达式 EXPR 的值发生变化时, 暂停程序执行。
删除监视点	d N	d 2	删除序号为 N 的监视点
打印栈帧链(3)	bt	bt	打印栈帧链

备注:

- (1) 命令已实现
- (2) 与 GDB 相比, 我们在这里做了简化, 更改了命令的格式
- (3) 在 PA2 中实现

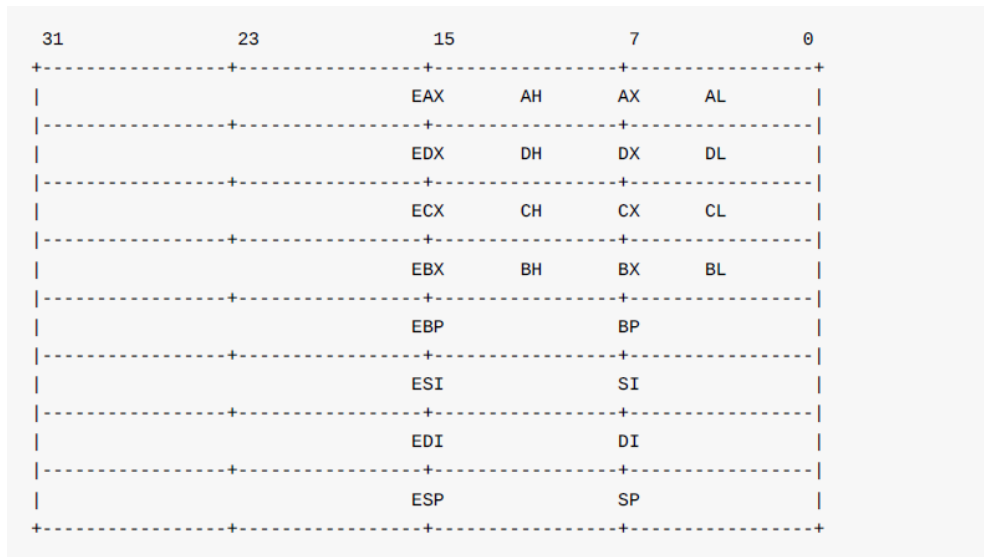
总有一天会找上门来的 bug

你需要在将来的 PA 中使用这些功能来帮助你进行 NEMU 的调试, 如果你的实现是有问题的, 将来你有可能面临以下悲惨的结局: 你实现了某个新功能之后, 打算对它进行测试, 通过扫描内存的功能来查看一段内存, 发现输出并非预期结果。为是刚才实现的新功能有问题, 于是对它进行调试。过了几天几夜的调试之后, 你泪流满面地发现, 原来是扫描内存的功能有 bug!

如果你想避免类似的悲惨结局, **你需要在实现一个功能之后对它进行充分的测试**。随着时间的推移, 发现同一个 bug 所需要的代价会越来越大。

寄存器结构体

寄存器是 CPU 中一个重要的组成部分,在 CPU 中进行运算所用到的数据和结果都会存放在寄存器中。i386 手册的第 2.3 节对 i386 中所用寄存器进行了简单的介绍。在现阶段的 NEMU 中,我们只会用到其中的两类寄存器:首先是通用寄存器。通用寄存器的结构如下图所示:



其中

- EAX, EDX, ECX, EBX, EBP, ESI, EDI, ESP 是 32 位寄存器;
- AX, DX, CX, BX, BP, SI, DI, SP 是 16 位寄存器;
- AL, DL, CL, BL, AH, DH, CH, BH 是 8 位寄存器。

但它们在物理上并不是相互独立的,例如 EAX 的低 16 位是 AX,而 AX 又分成 AH 和 AL。这样的结构有时候在处理数据时能提供一些便利。至于如何实现这样的结构,当然是难不倒聪明的你啦!

第二类在 NEMU 中用到的寄存器就是 EIP,也就是大名鼎鼎的程序计数器(Program Counter)。程序执行就是执行一行一行的 C 代码;在计算机硬件的世界里,程序执行也有类似的表现,就是执行一条一条的指令。但计算机怎么知道程序已经执行到哪里呢?肩负着这一重要使命的就是程序计数器了,i386 给它起了一个名字叫 EIP。可别小看了这个 32 位的家伙,你会在 PA2 中频繁地跟它打交道。随着实验的推进,更多的寄存器会加入到 NEMU 中。

必做任务 1：实现正确的寄存器结构体

我们在 PA0 中提到，运行 NEMU 会出现 assertion fail 的错误信息，这是因为框架代码并没有正确地实现用于模拟寄存器的结构体 `CPU_state`，现在你需要实现它了(结构体的定义在 `nemu/include/cpu/reg.h` 中)。关于 i386 寄存器的更多细节，请查阅 i386 手册。Hint：使用匿名 `struct` 和 `union`。

在 `nemu/src/cpu/reg.c` 中有一个 `reg_test()` 函数，它会生成一些随机的数据，来测试你的实现是否正确，若不正确，将会触发 assertion fail。实现正确之后，NEMU 将不会在 `reg_test()` 中触发 assertion fail，同时会输出 NEMU 的命令提示符：

```
(nemu)
```

输入 `c` 之后，NEMU 将会运行一个由 `mov` 指令组成的用户程序，最后输出如下信息：

```
nemu: HIT GOOD TRAP at eip = 0x001002b1
```

这说明程序成功地结束运行。键入 `q` 退出 NEMU。此时可以打开 `log.txt` 文件查看刚才程序执行的每一条指令。

解析命令

NEMU 通过 `readline` 库与用户交互，使用 `readline()` 函数从键盘上读入命令。与 `gets()` 相比，`readline()` 提供了“行编辑”的功能，最常用的功能就是通过上、下方向键翻阅历史记录。事实上，shell 程序就是通过 `readline()` 读入命令的。关于 `readline()` 的功能和返回值等信息，请查阅

```
man readline
```

从键盘上读入命令后，NEMU 需要解析该命令，然后执行相关的操作。解析命令的目的是识别命令中的参数，例如在 `si 10` 的命令中识别出 `si` 和 `10`，从而得知这是一条单步执行 `10` 条指令的命令。解析命令的工作是通过一系列的字符串处理函数来完成的，例如框架代码中的 `strtok()`。`strtok()` 是 C 语言中的标准库函数，如果你从来没有使用过 `strtok()`，并且打算继续使用框架代码中的 `strtok()` 来进行命令的解析，请务必查阅

```
man strtok
```

另外, `cmd_help()` 函数中也给出了使用 `strtok()` 的例子. 事实上, 字符串处理函数有很多, 键入以下内容:

```
man 3 str<TAB><TAB>
```

其中 `<TAB>` 代表键盘上的 TAB 键. 你会看到很多以 `str` 开头的函数, 其中有你应该很熟悉的 `strlen()`, `strcpy()` 等函数. 你最好都先看看这些字符串处理函数的 manual page, 了解一下它们的功能, 因为你很可能会用到其中的某些函数来帮助你解析命令. 当然你也可以编写你自己的字符串处理函数来解析命令.

另外一个值得推荐的字符串处理函数是 `sscanf()`, 它的功能和 `scanf()` 很类似, 不同的是 `sscanf()` 可以从字符串中读入格式化的内容, 使用它有时候可以很方便地实现字符串的解析. 如果你从来没有使用过它们, 请阅读源代码, 或者到互联网上查阅相关资料.

单步执行

单步执行的功能十分简单, 而且框架代码中已经给出了模拟 CPU 执行方式的函数, 你只要使用相应的参数去调用它就可以了. 如果你仍然不知道要怎么做, 请阅读源代码.

打印寄存器

打印寄存器就更简单了, 执行 `info r` 之后, 直接用 `printf()` 输出所有寄存器的值即可. 如果你不知道要输出什么, 你可以参考 GDB 中的输出.

扫描内存

扫描内存的实现也不难, 对命令进行解析之后, 先求出表达式的值. 但你还没有实现表达式求值的功能, 现在可以先实现一个简单的版本: 规定表达式 `EXPR` 中只能是一个十六进制数, 例如

```
x 10 0x100000
```

这样的简化可以让你暂时不必纠缠于表达式求值的细节. 解析出待扫描内存的起始地址之后, 你就使用循环将指定长度的内存数据通过十六进制打印出来. 如果你不知道要怎么输出, 同样的, 你可以参考 GDB 中的输出.

实现了扫描内存的功能之后, 你可以打印 `0x100000` 附近的内存, 你应该

会看到程序的代码，和用户程序的 objdump 结果进行对比(此时用户程序是 `mov`，其 dump 结果在 `obj/testcase/mov.txt` 中)，看看你的实现是否正确。

必做任务 2：实现单步执行、打印寄存器、扫描内存

熟悉了 NEMU 的框架之后，这些功能实现起来都很简单，同时我们对输出的格式不作硬性规定，就当做是熟悉 GNU/Linux 编程的一次练习吧。

不知道如何下手？嗯，看来你需要再阅读一遍“阅读源代码”小节的内容了。不敢下手？别怕，放手去写！编译运行就知道写得对不对。代码改挂了，就改回来呗。

温馨提示

PA1 阶段 1 到此结束.

数学表达式求值

给你一个表达式的字符串

```
"5 + 4 * 3 / 2 - 1"
```

你如何求出它的值？表达式求值是一个很经典的问题，以至于有很多方法来解决它。我们在所需知识和难度两方面做了权衡，在这里使用如下方法来解决表达式求值的问题：

- 首先识别出表达式中的单元
- 根据表达式的归纳定义进行递归求值

词法分析

"词法分析"这个词看上去很高端，说白了就是做上面的第 1 件事情，"识别出表达式中的单元"。这里的"单元"是指有独立含义的子串，它们正式的称呼叫 **token**。具体地说，我们需要在上述表达式中识别出 **5, +, 4, *, 3, /, 2, -, 1** 这些 token。你可能会觉得这是一件很简单的事情，但考虑以下的表达式：

```
"0xc0100000+ ($eax +5)*4 - *( $ebp + 8) + number"
```

它包含更多的功能，例如十六进制整数(0xc0100000)，小括号，访问寄存器(\$eax)，指针解引用(第二个 *)，访问变量(number)。事实上，这种复杂的表达式在调试过程中经常用到，而且你需要在空格数目不固定(0 个或多个)的情况下仍然能正确识别出其中的 token。当然你仍然可以手动进行处理(如果你喜欢挑战性的工作的话)，一种更方便快捷的做法是使用**正则表达式**。正则表达式可以很方便地匹配出一些复杂的 pattern，是程序员必须掌握的内容，如果你从来没有接触过正则表达式，请查阅[相关资料](#)。在实验中，你只需要了解正则表达式的一些基本知识就可以了(例如**元字符**)。

学会使用简单的正则表达式之后，你就可以开始考虑如何利用正则表达式来识别出 token 了。我们先来处理一种简单的情况 -- 算术表达式，即待求值表达式中只允许出现以下的 token 类型：

- 十进制整数
- **+, -, *, /**
- **(,)**
- 空格串(一个或多个空格)

首先我们需要使用正则表达式分别编写用于识别这些 token 类型的规则。在框架代码中，一条规则是由正则表达式和 token 类型组成的二元组。框架代码中已经给出了+和空格串的规则，其中空格串的 token 类型是 NOTYPE，因为空格串并不参加求值过程，识别出来之后就可以将它们丢弃了；+的 token 类型是 '+'，事实上 token 类型只是一个整数，只要保证不同的类型的 token 被编码成不同的整数就可以了；框架代码中还有一条用于识别双等号的规则，不过我们现在可以暂时忽略它。

这些规则会在 NEMU 初始化的时候被编译成一些用于进行 pattern 匹配的内部信息，这些内部信息是被库函数使用的，而且它们会被反复使用，但你不必关心它们如何组织。但如果正则表达式的编译不通过，NEMU 将会触发 assertion fail，此时你需要检查编写的规则是否符合正则表达式的语法。

给出一个待求值表达式，我们首先要识别出其中的 token，进行这项工作的是 make_token() 函数（nemu/src/monitor/debug/expr.c）。make_token() 函数的工作方式十分直接，它用 position 变量来指示当前处理到的位置，并且按顺序尝试用不同的规则来匹配当前位置的字符串。当一条规则匹配成功，并且匹配出的子串正好是 position 所在位置的时候，我们就成功地识别出一个 token，Log() 宏会输出识别成功的信息。你需要做的是将识别出的 token 信息记录下来（一个例外是空格串），我们使用 Token 结构体来记录 token 的信息：

```
typedef struct token {
    int type;
    char str[32];
} Token;
```

其中 type 成员用于记录 token 的类型。大部分 token 只要记录类型就可以了，例如 +, -, *, /，但这对于有些 token 类型是不够的：如果我们只记录了一个十进制整数 token 的类型，在进行求值的时候我们还是不知道这个十进制整数是多少，这时我们应该将 token 相应的子串也记录下来，str 成员就是用来做这件事情的。需要注意的是，str 成员的长度是有限的，当你发现缓冲区将要溢出的时候，要进行相应的处理（思考一下，你会如何处理？），否则将会造成难以理解的 bug。tokens 数组用于按顺序存放已经被识别出的 token 信息，nr_token 指示已经被识别出的 token 数目。

如果尝试了所有的规则都无法在当前位置识别出 token，识别将会失败。make_token() 函数将返回 false，表示词法分析失败。

系统设计的黄金法则 -- KISS 法则

这里的 KISS 是 Keep It Simple, Stupid 的缩写，它的中文翻译是：不要在一开始追求绝对的完美。

你已经学习过很多计算机专业课，这意味着你已经学会写程序了，但这并不意味着你可以顺利地完成 PA，因为在现实世界中，我们需要的是可以运行的 system，而不是求阶乘的小程序。NEMU 作为一个麻雀虽小，五脏俱全的小型系统，其代码量达到 6000 多行(不包括空行)。随着 PA 的进行，代码量会越来越多，各个模块之间的交互也越来越复杂，工程的维护变得越来越困难，一个很弱智的 bug 可能需要调好几天。在这种情况下，系统能跑起来才是王道，跑不起来什么都是浮云，追求面面俱到只会增加代码维护的难度。

唯一可以把你从 bug 的混沌中拯救出来的就是 KISS 法则，它的宗旨是从易到难，逐步推进，一次只做一件事，少做无关的事。如果你不知道这是什么意思，我们上文提到的 `str` 成员缓冲区溢出问题来作为例子。KISS 法则告诉你，你应该使用 `assert(0)`，这是因为表达式求值的核心功能和处理上述问题是不耦合的，说得通俗点，就算不"得体"地处理上述问题，仍然不会影响表达式求值的核心功能的正确性。如果你还记得调试公理，你会发现两者之间是有联系的：调试公理第二点告诉你，未测试代码永远是错的，与其一下子写那么多"错误"的代码，倒不如使用 `assert(0)` 来有效帮助你减少这些"错误"。

如果把 KISS 法则放在软件工程领域来解释，它强调的就是多做[单元测试](#)：写一个函数，对它进行测试，正确之后再写下一个函数，再对它进行测试... 一种好的测试方式是使用 `assertion` 进行验证，`reg_test()` 就是这样的例子。学会使用 `assertion`，对程序的测试和调试都百利而无一害。

KISS 法则不但广泛用在计算机领域，就连其它很多领域也视其为黄金法则，[这里](#)有一篇文章举出了很多的例子，我们强烈建议你阅读它，体会 KISS 法则的重要性。

必做任务 3：实现算术表达式的词法分析

你需要完成以下内容：

- 为算术表达式中的各种 token 类型添加规则，你需要注意 C 语言字符串中转义字符的存在和正则表达式中元字符的功能。
- 在成功识别出 token 后，将 token 的信息依次记录到 `tokens` 数组中。

递归求值

把待求值表达式中的 token 都成功识别出来之后，接下来我们就可以进行求值了。需要注意的是，我们现在是在对 tokens 数组进行处理，为了方便叙述，我们称它为"token 表达式"。例如待求值表达式

```
"4 +3*(2- 1)"
```

的 token 表达式为

```
+-----+-----+-----+-----+-----+-----+-----+-----+-----+
| NUM | '+' | NUM | '*' | '(' | NUM | '-' | NUM | ')' |
| "4" |    | "3" |    |    | "2" |    | "1" |    |
+-----+-----+-----+-----+-----+-----+-----+-----+-----+
```

根据表达式的归纳定义特性，我们可以很方便地使用递归来进行求值。首先我们给出算术表达式的归纳定义：

```
<expr> ::= <number>          # 一个数是表达式
        | "(" <expr> ")"      # 在表达式两边加个括号也是表达式
        | <expr> "+" <expr>    # 两个表达式相加也是表达式
        | <expr> "-" <expr>    # 接下来你全懂了
        | <expr> "*" <expr>
        | <expr> "/" <expr>
```

上面这种表示方法就是大名鼎鼎的[巴克斯范式 \(BNF\)](#)。根据上述 BNF 定义，一种解决方案已经逐渐成型了：既然长表达式是由短表达式构成的，我们就先对短表达式求值，然后再对长表达式求值。这种十分自然的解决方案就是[分治法](#)的应用，就算你没听过这个高大上的名词，也不难理解这种思路。而要实现这种解决方案，递归是你的不二选择。

为了在 token 表达式中指示一个子表达式，我们可以使用两个整数 **p** 和 **q** 来指示这个子表达式的开始位置和结束位置。这样我们就可以很容易把求值函数的框架写出来了：

```

eval(p, q) {
  if(p > q) {
    /* Bad expression */
  }
  else if(p == q) {
    /* Single token.
     * For now this token should be a number.
     * Return the value of the number.
     */
  }
  else if(check_parentheses(p, q) == true) {
    /* The expression is surrounded by a matched pair of parentheses.
     * If that is the case, just throw away the parentheses.
     */
    return eval(p + 1, q - 1);
  }
  else {
    /* We should do more things here. */
  }
}

```

其中 `check_parentheses()` 函数用于判断表达式是否被一对匹配的括号包围着，同时检查表达式的左右括号是否匹配，如果不匹配，这个表达式肯定是不符合语法的，也就不需要继续进行求值了。我们举一些例子来说明 `check_parentheses()` 函数的功能：

```

"(2 - 1)" // true
"(4 + 3 * (2 - 1))" // true
"4 + 3 * (2 - 1)" // false, the whole expression is not surrounded by a matched pair of parentheses
"(4 + 3)) * ((2 - 1)" // false, bad expression
"(4 + 3) * (2 - 1)" // false, the leftmost '(' and the rightmost ')' are not matched

```

至于怎么检查左右括号是否匹配，就留给聪明的你来思考吧！

上面的框架已经考虑了 BNF 中算术表达式的开头两种定义，接下来我们来考虑剩下的情况(即上述伪代码中最后一个 `else` 中的内容)。一个问题是，给出一个最左边和最右边不同时是括号的长表达式，我们要怎么正确地将它分裂成两个子表达式？我们定义 **dominant operator** 为表达式人工求值时，最后一步进行运行的运算符，它指示了表达式的类型(例如当最后一步是减法运算时，表达式本质上是一个减法表达式)。要正确地对一个长表达式进行分裂，就是要找到它的 **dominant operator**。我们继续使用上面的例子来探讨这个问题：

```

"4 + 3 * ( 2 - 1 )"
/*****/
case 1:
    "+"
    /  \
"4"    "3 * ( 2 - 1 )"

case 2:
    "*"
    /  \
"4 + 3" "( 2 - 1 )"

case 3:
    "-"
    /  \
"4 + 3 * ( 2" "1 )"

```

上面列出了 3 种可能的分裂, 注意到我们不可能在非运算符的 token 处进行分裂, 否则分裂得到的结果均不是合法的表达式。根据 dominant operator 的定义, 我们很容易发现, 只有第一种分裂才是正确的, 这其实也符合我们人工求值的过程: 先算 4 和 $3 * (2 - 1)$, 最后把它们的结果相加。第二种分裂违反了算术运算的优先级, 它会导致加法比乘法更早进行。第三种分裂破坏了括号的平衡, 分裂得到的结果均不是合法的表达式。

通过上面这个简单的例子, 我们就可以总结出如何在一个 token 表达式中寻找 dominant operator 了:

- 非运算符的 token 不是 dominant operator。
- 出现在一对括号中的 token 不是 dominant operator。注意到这里不会出现有括号包围整个表达式的情况, 因为这种情况已经在 `check_parentheses()` 相应的 `if` 块中被处理了。
- dominant operator 的优先级在表达式中是最低的。这是因为 dominant operator 是最后一步才进行的运算符。
- 当有多个运算符的优先级都是最低时, 根据结合性, 最后被结合的运算符才是 dominant operator。一个例子是 $1 + 2 + 3$, 它的 dominant operator 应该是右边的 `+`。

要找出 dominant operator, 只需要将 token 表达式全部扫描一遍, 就可以按照上述方法唯一确定 dominant operator。


```

eval(p, q) {
    if(p > q) {
        /* Bad expression */
    }
    else if(p == q) {
        /* Single token.
         * For now this token should be a number.
         * Return the value of the number.
         */
    }
    else if(check_parentheses(p, q) == true) {
        /* The expression is surrounded by a matched pair of parentheses.
         * If that is the case, just throw away the parentheses.
         */
        return eval(p + 1, q - 1);
    }
    else {
        op = the position of dominant operator in the token expression;
        val1 = eval(p, op - 1);
        val2 = eval(op + 1, q);

        switch(op_type) {
            case '+': return val1 + val2;
            case '-': /* ... */
            case '*': /* ... */
            case '/': /* ... */
            default: assert(0);
        }
    }
}
}

```

找到了正确的 dominant operator 之后，事情就变得很简单了，先对分裂出来的两个子表达式进行递归求值，然后再根据 dominant operator 的类型对两个子表达式的值进行运算即可。于是完整的求值函数如上所示。

必做 4：实现算术表达式的递归求值

我们已经把递归求值的思路和框架都列出来了，你需要做的是理解这一思路，然后在框架中填充相应内容。实现表达式求值的功能之后，`p` 命令也就不难实现了。

需要注意的是，上述框架中并没有进行错误处理，在求值过程中发现表达式不合法的时候，应该给上层函数返回一个表示出错的标识，告诉上层函数“求值的结果是无效的”。例如在 `check_parentheses()` 函数中，`(4 + 3) * ((2 - 1)` 和 `(4 + 3) * (2 - 1)` 这两个表达式虽然都返回 `false`，因为前一种情况是表达式不合法，是没有办法成功进行求值的；而后一种情况是一个合法的表达式，是可以成功求值的，只不过它的形式不属于 BNF 中的 `"(<expr>)"`，需要使用 dominant operator 的方式进行处理，因此你还需要想办法把它们区别开来。

当然，你也可以在发现非法表达式的时候使用 `assert(0)` 终止程序，不过这样的话，你在使用表达式求值功能的时候就要十分谨慎了。

选做任务 1：实现带有负数的算术表达式的求值

在上述实现中，我们并没有考虑负数的问题，例如“1 + -1”。

此时会被判定为不合法的表达式。为了实现负数的功能，需要考虑两个问题：

- 负号和减号都是 **-**，如何区分它们？
- 负号是个单目运算符，分裂的时候需要注意什么？

你可以选择不实现负数的功能，但你很快就要面临类似的问题了。

调试中的表达式求值

实现了算术表达式的求值之后，你可以很容易把功能扩展到复杂的表达式。我们用 BNF 来说明需要扩展哪些功能：

```
<expr> ::= <decimal-number>
| <hexadecimal-number>      # 以"0x"开头
| <reg_name>                  # 以"$"开头
| "(" <expr> ")"
| <expr> "+" <expr>
| <expr> "-" <expr>
| <expr> "*" <expr>
| <expr> "/" <expr>
| <expr> "==" <expr>
| <expr> "!=" <expr>
| <expr> "&&" <expr>
| <expr> "||" <expr>
| "!" <expr>
| "*" <expr>                  # 指针解引用
```

它们的功能和 C 语言中运算符的功能是一致的，包括优先级和结合性，如有疑问，请查阅相关资料。需要注意的是指针解引用(dereference)的识别，在进行词法分析的时候，我们其实没有办法把乘法和指针解引用区别开来，因为它们都是 *****。在进行递归求值之前，我们需要将它们区别开来，否则如果将指针解引用当成乘法来处理的话，求值过程将会认为表达式不合法。其实要区别它们也不难，给你一个表达式，你也能将它们区别开来。实际上，我们只要看 ***** 前一个 token 的类型，我们就可以决定这个 ***** 是乘法还是指针解引用了，不信你试试？我们在这里给出 `expr()` 函数的框架：

```

if(!make_token(e)) {
    *success = false;
    return 0;
}

/* TODO: Implement code to evaluate the expression. */

for(i = 0; i < nr_token; i++) {
    if(tokens[i].type == '*' && (i == 0 || tokens[i - 1].type == certain type) ) {
        tokens[i].type = Deref;
    }
}

return eval(?, ?);

```

其中的 **certain type** 就由你自己来思考啦！其实上述框架也可以处理负数问题，如果你之前实现了负数，***** 的识别对你来说应该没什么困难了。

另外和 GDB 中的表达式相比，我们做了简化，简易调试器中的表达式没有类型之分，因此我们需要额外说明两点：

- 为了方便统一，我们认为所有结果都是 **uint32_t** 类型。
- 指针也没有类型，进行指针解引用的时候，我们总是从内存中取出一个 **uint32_t** 类型的整数，同时记得使用 **swaddr_read()** 来读取内存。

必做任务 5：实现更复杂的表达式求值

你需要实现上文 BNF 中列出的，除 **指针解引用** 之外的功能。一个要注意的地方是词法分析中编写规则的顺序，不正确的顺序会导致一个运算符被识别成两部分，例如 **!=** 被识别成 **!** 和 **=**。关于变量的功能，它需要涉及符号表和字符串表的查找，因此你会在 PA2 中实现它。

选做任务 2：实现指针解引用

按上文提示实现对指针解引用运算符 ***** 的识别

温馨提示

PA1 阶段 2 到此结束。

监视点

监视点的功能是监视一个表达式的值何时发生变化。如果你从来没有使用过监视点，请在 GDB 中体验一下它的作用。

简易调试器允许用户同时设置多个监视点，删除监视点，因此我们最好使用链表将监视点的信息组织起来。框架代码中已经定义好了监视点的结构体(在 `nemu/include/monitor/watchpoint.h` 中)：

```
typedef struct watchpoint {
    int NO;
    struct watchpoint *next;

    /* TODO: Add more members if necessary */

} WP;
```

但结构体中只定义了两个成员：`NO` 表示监视点序号，`next` 就不用多说了吧。为了实现监视点功能，你需要根据对监视点工作原理的理解在结构体中增加必要的成员。同时我们使用"池"的数据结构来管理监视点结构体，框架代码中已经给出了一部分相关的代码(在 `nemu/src/monitor/debug/watchpoint.c` 中)：

```
static WP wp_pool[NR_WP];
static WP *head, *free_;
```

代码中定义了监视点结构的池 `wp_pool`，还有两个链表 `head` 和 `free_`，其中 `head` 用于组织使用中的监视点结构，`free_` 用于组织空闲的监视点结构，`init_wp_pool()` 函数会对两个链表进行了初始化。

必做任务 6：实现监视点池的管理

为了使用监视点池，你需要编写以下两个函数(你可以根据你的需要修改函数的参数和返回值)：

```
WP* new_wp();
```

```
void free_wp(WP *wp);
```

其中 `new_wp()` 从 `free_` 链表中返回一个空闲的监视点结构，`free_wp()` 将 `wp` 归还到 `free_` 链表中，这两个函数会作为监视点池的接口被其它函数调用。

需要注意的是，调用 `new_wp()` 时可能会出现没有空闲监视点结构的情况，为了简单起见，此时可以通过 `assert(0)` 马上终止程序。框架代码中定义了 32 个监视点结构，一般情况下应该足够使用，如果你需要更多的监视点结构，你可以修改 `NR_WP` 宏的值。

这两个函数里面都需要执行一些链表插入，删除的操作，对链表操作不熟悉的同学来说，这可以作为一次链表的练习。

思考题 2：温故而知新（2）

框架代码中定义 `wp_pool` 等变量的时候使用了关键字 `static`，`static` 在此处的含义是什么？为什么要在此处使用它？

实现了监视点池的管理之后，我们就可以考虑如何实现监视点的相关功能了。具体的，你需要实现以下功能：

- 当用户给出一个待监视表达式时，你需要通过 `new_wp()` 申请一个空闲的监视点结构，并将表达式记录下来。每当 `cpu_exec()` 执行完一条指令，就对所有待监视的表达式进行求值(你之前已经实现了表达式求值的功能了)，比较它们的值有没有发生变化，若发生了变化，程序就因触发了监视点而暂停下来，你需要将 `nemu_state` 变量设置为 `STOP` 来达到暂停的效果。最后输出一句话提示用户触发了监视点，并返回到 `ui_mainloop()` 等待用户的命令。
- 触发监视点后输出的提示语可以自行定义，**但必须包含**如下语句 “Hint watchpoint 0 at address 0x00100014”，其中 0 和 0x00100014 会根据监视点的不同而不同，其中“0”表示所触发的监视点的编号，“0x00100014”表示触发监视点指令的 `%eip` 的取值 (**必须使用 16 进制表示**)。
- 使用 `info w` 命令来打印使用中的监视点信息，至于要打印什么，你可以参考 GDB 中 `info watchpoints` 的运行结果。
- 使用 `d` 命令来删除监视点，你只需要释放相应的监视点结构即可。

必做任务 7：实现监视点

你需要实现上文描述的监视点相关功能，实现了表达式求值之后，监视点实现的重点就落在了链表操作上。如果你仍然因为链表的实现而感到调试困难，请尝试学会使用 `assertion`。

断点

断点的功能是让程序暂停下来, 从而方便查看程序某一时刻的状态。事实上, 我们可以很容易地用监视点来模拟断点的功能:

```
w $eip == ADDR
```

其中 **ADDR** 为设置断点的地址。这样程序执行到 **ADDR** 的位置时就会暂停下来。

调试器设置断点的工作方式和上述通过监视点来模拟断点的方法大相径庭。事实上, 断点的工作原理, 竟然是三十六计之中的"偷龙转凤"! 如果你想揭开这一神秘的面纱, 你可以阅读[这篇文章](#)。

从表达式求值窥探编译器？

你在程序设计课上已经知道，编译是一个将高级语言转换成机器语言的过程。但你是否曾经想过，机器是怎么读懂你的代码的？回想你实现表达式求值的过程，你是否有什么新的体会？

事实上，词法分析也是编译器编译源代码的第一个步骤，编译器也需要从你的源代码中识别出 `token`，这个功能也可以通过正则表达式来完成，只不过 `token` 的类型更多，更复杂而已。这也解释了你为什么可以在源代码中插入任意数量的空白字符(包括空格，`tab`，换行)，而不会影响程序的语义；你也可以将所有源代码写到一行里面，编译仍然能够通过。

一个和词法分析相关的有趣的应用是语法高亮。在程序设计课上，你可能完全没有想过可以自己写一个语法高亮的程序，事实是，这些看似这么神奇的东西，其实也没那么复杂，你现在确实有能力来实现它：把源代码看作一个字符串输入到语法高亮程序中，在循环中识别出一个 `token` 之后，根据 `token` 类型用不同的颜色将它的内容重新输出一遍就可以了。如果你打算将高亮的代码输出到终端里，你可以使用 [ANSI 转义码的颜色功能](#)。

在表达式求值的递归求值过程中，逻辑上其实做了两件事情：第一件事是根据 `token` 来分析表达式的结构(属于 BNF 中的哪一种情况)，第二件事才是求值。它们在编译器中也有对应的过程：语法分析就好比分析表达式的结构，只不过编译器分析的是程序的结构，例如哪些是函数，哪些是语句等等。当然程序的结构要比表达式的结构更复杂，因此编译器一般会使用一种标准的框架来分析程序的结构，理解这种框架需要更多的知识，这里就不展开叙述了。另外如果你有兴趣，可以看看 C 语言语法的 BNF。

和表达式最后的求值相对的，在编译器中就是代码生成。大家回想一下，“计算机系统基础”课程有专门的章节来讲解 C 代码和汇编指令的关系，即使你不了解代码具体是怎么生成的，你仍然可以理解它们之间的关系，这是因为 C 代码天生就和汇编代码有密切的联系，高水平程序员的思维甚至可以在 C 代码和汇编代码之间相互转换。如果要深究代码生成的过程，你也不难猜到是用递归实现的：例如要生成一个函数的代码，就先生成其中每一条语句的代码，然后通过某种方式将它们连接起来。

我们通过表达式求值的实现来窥探编译器的组成，是为了落实一个道理：学习汽车制造专业不仅仅是为了学习开汽车，是要学习发动机怎么设计。我们也强烈推荐你在将来修读“编译原理”课程，深入学习“如何设计编译器”。

实验秘笈：i386 手册

在以后的 PA 中，你需要反复阅读 i386 手册。鉴于有同学片面地认为“看手册”就是“把手册全看一遍”，因而觉得“不可能在短时间内看完”，我们在 PA1 的最后来聊聊如何科学地看手册。

学会使用目录

了解一本书都有哪些内容的最快方法就是查看目录，尤其是当你第一次看一本新书的时候。查看目录之后并不代表你知道它们具体在说什么，但你会对这些内容有一个初步的印象，提到某一个概念的时候，你可以大概知道这个概念会在手册中的哪些章节出现。这对查阅手册来说是极其重要的，因为我们每次查阅手册的时候总是关注某一个问题，如果每次都需要把手册从头到尾都看一遍才能确定关注的问题在哪里，效率是十分低下的。事实上也没有人会这么做，阅读目录的重要性可见一斑。纸上得来终觉浅，还是来动手体会一下吧！

假设你现在需要了解一个叫 **selector** 的概念，请通过 i386 手册的目录确定你需要阅读手册中的哪些地方。

怎么样，是不是很简单？虽然你还是不明白 **selector** 是什么，但你已经知道你需要阅读哪些地方了，要弄明白 **selector**，那也是指日可待的事情了。

逐步细化搜索范围

有时候你关注的问题不一定直接能在目录里面找到，例如“CR0 寄存器的 PG 位的含义是什么”。这种细节的问题一般都是出现在正文中，而不会直接出现在目录中，因此你就不能直接通过目录来定位相应的内容了。根据你是否第一次接触 CR0，查阅这个问题会有不同的方法：

- 如果你已经知道 CR0 是个 control register，你可以直接在目录里面查看“control register”所在的章节，然后在这些章节的正文中寻找“CR0”。
- 如果你对 CR0 一无所知，你可以使用阅读器中的搜索功能，搜索“CR0”，还是可以很快地找到“CR0”的相关内容。不过最好的方法是首先使用搜索引擎，你可以马上知道“CR0 是个 control register”，然后就可以像第一种方法那样查阅手册了。

不过有时候，你会发现一个概念在手册中的多个地方都有提到。这时你需要明确你要关心概念的哪个方面，通常一个概念的某个方面只会在手册中的一个地方进行详细的介绍。你需要在这多个地方中进行进一步的筛选，但至少你已经过滤掉很多与这个概念无关的章节了。筛选也是有策略的，你不需要把多个地方的所有内容全部阅读一遍才能进行筛选，小标题，每段的第一句话，图表的注解，这些都可以帮助你很快地了解这一部分的内容大概在讲什么。这不就是高中英语考试中的快速阅读吗？对的，就是这样。如果你觉得目前还缺乏这方面的能力，

现在锻炼的好机会来了。

搜索和筛选信息是一个 trail and error 的过程, 没有什么方法能够指导你在第一遍搜索就能成功, 但还是有经验可言的。搜索失败的时候, 你应该尝试使用不同的关键字重新搜索。至于怎么变换关键字, 就要看你对问题核心的理解了, 换句话说, 怎么问才算是切中要害。这不就是高中语文强调的表达能力吗? 对的, 就是这样。

事实上, 你只需要具备一些基本的交际能力, 就能学会查阅资料, 这和资料的内容没有关系, 来一本"民法大全", "XX 手机使用说明书", "YY 公司人员管理记录", 照样是这么查阅。"查阅资料"是一种与领域无关的基本能力, 无论身处哪一个行业都需要具备, 如果你不想以后工作的时候被查阅资料的能力影响了自己的前途, 从现在开始就努力锻炼吧!

思考题 3: 查阅资料小挑战

查阅各种资料, 回答以下问题:

- **查阅 i386 手册** 理解了科学查阅手册的方法之后, 请你尝试在 i386 手册中查阅以下问题所在位置, 并把需要阅读的范围写到实验报告里面:
 - ◆ EFLAGS 寄存器中的 CF 位是什么意思?
 - ◆ ModR/M 字节是什么?
 - ◆ mov 指令的具体格式是怎么样的?
- **shell 命令** 完成 PA1 的内容之后, nemu 目录下的所有 .c 和 .h 文件总共有多少行代码? 你是使用什么命令得到这个结果的? 和框架代码相比, 你在 PA1 中编写了多少行代码? 你可以把这条命令写入 **Makefile** 中, 随着实验进程的推进, 你可以很方便地统计工程的代码行数, 例如敲入 **make count** 就会自动运行统计代码行数的命令。再来个难一点的, 除去空行之外, nemu 目录下的所有 .c 和 .h 文件总共有多少行代码?
- **Make 文件** 打开工程目录下的 **Makefile** 文件, 你会在 **CFLAGS** 变量中看到 gcc 的一些编译选项。请解释 gcc 中的 **-Wall** 和 **-Werror** 有什么作用? 为什么要使用 **-Wall** 和 **-Werror**?

温馨提示

PA1 到此结束。

任务自查表

序号	是否已完成
必做任务 1	
必做任务 2	
必做任务 3	
必做任务 4	
必做任务 5	
必做任务 6	
必做任务 7	
选做任务 1	
选做任务 2	

实验提交说明

提交地址

阶段性工程提交至自建 git 远程仓库

最终工程提交学部虚拟仿真实验平台

提交方式

1. 实验报告在线填写（在第一栏写上题目“**实验 1：简易调试器实验报告**”）；
2. 教师将通过脚本自动检查运行结果，因此除非实验手册特别说明，不要修改工程中的任何脚本，包括 Makefile、test.sh。
 - 我们会清除中间结果，使用原来的编译选项重新编译(包括 -Wall 和 -Werror)，若编译不通过，本次实验你将得 0 分(编译错误是最容易排除的错误，我们有理由认为你没有认真对待实验)。
3. 学生使用 `make submit` 命令将整个工程打包，导出到本地机器，再通过虚拟仿真平台“附件”功能上传。
 - `make submit` 命令会用你的学号来命名压缩包，然后修改压缩包的名称为“学号-PA1.zip”。此外，不要修改压缩包内工程根目录的命名。为了防止出现编码问题，压缩包中所有文件名都不要包含中文。

git 版本控制

请使用 git 管理你的项目，并合理地手动提交的 git 记录。请你不定期查看自己的 git log，检查是否与自己的开发过程相符。git log 是独立完成实验的最有力证据，完成了实验内容却缺少合理的 git log，会给抄袭判定提供最有力的证据。

如果出现 git head 损毁现象，说明你拷贝了他人的代码，也按抄袭处理。

实验报告内容

你必须在实验报告中描述以下内容：

- **实验进度。按照“任务自查表”格式在线制作表格并填写。缺少实验进度的描述，或者描述与实际情况不符，将被视为没有完成本次实验。**

● 思考题

你可以自由选择报告的其它内容。你不必详细地描述实验过程, 但我们鼓励你在报告中描述如下内容:

- 你遇到的问题和对这些问题的思考、解决办法
- 实验心得

如果你实在没有想法, 你可以提交一份不包含任何想法的报告, 我们不会强求。但请不要

- 大量粘贴讲义内容
- 大量粘贴代码和贴图, 却没有相应的详细解释(让我们明显看出是凑字数的)

来让你的报告看起来十分丰富, 编写和阅读这样的报告毫无任何意义, 你也不会因此获得更多的分数, 同时还可能带来扣分的可能。