

汇编语言第一次大作业实验报告

小组成员：袁晨圃、王萃璠、曾子恒、王永生

2025 年 4 月 28 日

1 实验总体思路及过程

- 总体思路：采用字典树（Trie 树）对输入内容进行处理，找到出现次数最多的单词并输出
- 数据结构设计：

- 使用字典树存储单词，每个节点表示一个字符
- 使用数组实现树结构，避免指针操作的复杂性
- 全局数据存储在 variable.s 中，主要包括以下数组：

```
1 son[MAXN][CHARSET_SIZE] // 存储当前节点的子节点，son[i][ch]表示节点i的字符ch子节点的ID
2 cnt[MAXN]                // 记录每个单词出现的次数
3 father[MAXN]             // 记录每个节点的父节点ID，用于回溯
4 content[MAXN]            // 存储每个节点对应的字符
5 max_id[MAXN]             // 存储出现次数最多的单词的节点ID
```

- 关键常量定义：

```
1 #define MAXN          2500 // 字典树的最大节点数
2 #define MAXL          100  // 单词的最大长度
3 #define CHARSET_SIZE 128   // 字符集大小（ASCII）
```

- 实验过程：

- 为分工合作，首先使用组员们熟悉的 C 语言实现了对输入内容的处理，将代码分为 4 大功能模块，分别是：

```
1 int main();      // 读入输入，调用子模块
2 void alpha();    // 处理字母，构建字典树
3 void not_alpha(); // 处理非字母，分割单词
4 void output(int id); // 从叶子节点回溯到根节点，输出单词
```

- 处理流程：

1. 逐个读取输入字符
2. 若是字母，则调用 alpha() 将其加入当前单词
3. 若不是字母，则调用 not_alpha() 处理当前单词结束
4. 所有输入处理完毕后，输出出现次数最多的单词

- 4 位小组成员每人负责将一个模块的代码用汇编语言实现，最后整合运行
- 小组成员的协作通过 github 进行代码共享，本次实验还通过 makefile 部署了自动化开发流程，自动对各模块进行编译，测试和合并运行

2 实验分工

- 袁晨圃：main.s 函数编写，代码整合测试
- 王萃璠：alpha.s 函数编写，实验报告撰写
- 曾子恒：not_alpha.s 函数编写，实验报告撰写
- 王永生：output.s 函数编写，实验报告撰写

3 汇编程序设计方法

3.1 总体设计思路

- 模块化分层设计：通过不同的函数模块实现不同的功能，清晰划分责任
- 数据逻辑分离：利用 variable.s 文件集中管理全局数据结构，其他文件通过 extern 引用
- 数据结构选择：
 - 采用字典树（Trie 树）实现单词存储和频率统计，时间复杂度、空间复杂度均为 $O(n)$ ，其中 n 为文本长度
 - 使用线性数组实现树结构，避免复杂的指针操作
 - variable.s 中定义的核心数据结构：

```
1 .section .data
2 son:      .fill MAXN*CHARSET_SIZE, 4, 0 # int son[MAXN][CHARSET_SIZE] = {0}
3 cnt:      .fill MAXN, 4, 0              # int cnt[MAXN] = {0}
4 father:   .fill MAXN, 4, 0              # int father[MAXN] = {0}
5 max_id:   .fill MAXN, 4, 0              # int max_id[MAXN] = {0}
6 content:  .fill MAXN, 1, 0              # char content[MAXN] = {0}
7 tot:      .long 0                       # int tot = 0
8 max_cnt:  .long 0                       # int max_cnt = 0
9 max_siz:  .long 0                       # int max_siz = 0
10 ch:      .long 0                       # int ch = 0
11 cur_id:  .long 0                       # int cur_id = 0
```

- 编程风格：
 - 使用位置无关代码（PIC）技术访问全局变量
 - 充分利用 64 位寄存器和地址计算指令
 - 严格遵循函数调用约定（保存被调用者保存寄存器）
 - 合理利用 C 标准库函数，如 getchar()、puts() 等

3.2 分模块具体设计

3.2.1 main.s

1. 主程序设计

- 使用外部符号声明，引入需要用到的函数和全局变量：

```
1 .globl _start
2 .extern isalpha, alpha, not_alpha, output, debug, fflush
3 .extern ch, max_id, max_siz
```

2. 主循环处理输入

- 循环读取字符：调用 `getchar` 读取输入，遇 EOF 结束循环

```
1 while_cond:
2     call    getchar
3     movl    %eax, ch(%rip) # let ch = getchar()
4     cmpl    $-1, %eax
5     je      while_end      # jump ch == EOF ? while_end : if_cond
```

- 字符分类处理：

- 判断是否为字母：

```
1 if_cond:
2     movl    ch(%rip), %edi
3     call    isalpha
4     testl   %eax, %eax
5     jz      else_stmt      # jump isalpha(ch) ? if_stmt : else_stmt
```

- 字母：调用 `alpha` 处理字母字符，构建单词
- 非字母：调用 `not_alpha` 处理非字母字符，用于分割单词

3. 结果输出阶段

- 遍历高频词数组：按 `max_siz` 遍历 `max_id` 数组，调用 `output` 逐项输出高频词

```
1     movq    $0, %rsi # let counter %rsi = 0
2 for_loop:
3     cmpl    max_siz(%rip), %esi
4     jge     for_end # if %esi >= max_siz then break
5     leaq    max_id(%rip), %rdi # let %rdi = &max_id
6     movl    (%rdi,%rsi,4), %edi # let %edi = max_id[%esi]
7     movslq   %edi, %rdi # sign extend %edi to %rdi
8     push    %rsi      # 保存循环计数器
9     call    output     # call output(max_id[%esi])
10    pop     %rsi      # 恢复循环计数器
11    inc     %rsi      # %esi++
12    jmp     for_loop
```

4. 收尾与调试

- 调试：调用 `debug` 打印内部数据结构（如树节点）
- 刷新输出：`fflush(NULL)` 强制写入流，确保缓冲区内容被输出

```
1     xor     %edi, %edi
2     call    fflush     # call fflush(NULL)
```

- 退出程序：使用 `syscall` 直接调用系统退出

```
1    movl    $60, %eax    # syscall number for exit
2    xorl    %edi, %edi    # exit code 0
3    syscall
```

3.2.2 alpha.s

1. 函数入口准备

- 栈帧建立： `pushq %rbp + movq %rsp, %rbp` 创建新栈帧
- 外部依赖：通过 `.extern` 声明使用外部全局变量（树结构核心数据）

2. 子节点地址计算

```
1    movl    cur_id(%rip), %eax    # 当前节点ID
2    movl    ch(%rip), %ecx    # 输入字符ASCII值
3    imull   $CHARSET_SIZE, %eax    # 计算二维数组行偏移 (CHARSET_SIZE=128)
4    addl    %ecx, %eax    # 列偏移叠加 (cur_id*128 + ch)
5    leaq    son(%rip), %rdx    # 获取son数组基址
6    leaq    (%rdx,%rax,4), %rsi    # 最终地址: &son[cur_id][ch]
```

- 核心目的是定位 `son[cur_id][ch]` 在内存中的地址

3. 子节点存在性检查

```
1    cmpl    $0, (%rsi)    # 检查子节点是否存在
2    jne     .skip_creation    # 存在则跳过创建
```

- 作用：判断当前字符是否已存在子节点路径

4. 新节点创建逻辑

- 计数器递增： `incl tot` 全局唯一 ID 生成
- 父节点记录： `father[tot] = cur_id` 记录新节点的父节点
- 字符存储： `content[tot] = ch` 存储当前字符到新节点
- 子节点注册： `son[cur_id][ch] = tot` 绑定父子关系

5. 当前节点更新

```
1    movl    (%rsi), %eax    # 获取子节点ID (新建或已有)
2    movl    %eax, cur_id(%rip)    # 更新cur_id为子节点
```

- 状态转移：将当前节点指针移动到子节点，准备处理下一字符

6. 函数返回

- 栈帧恢复： `leave + ret` 恢复调用前环境

3.2.3 not_alpha.s

1. 栈帧与寄存器保护（行 1-4）

- 建立栈帧并保存 rbx/r12/r13 寄存器，遵循 x64 调用约定
- 使用 PIC（位置无关代码）模式访问全局变量，通过 @GOTPCREL(%rip) 实现

2. 有效性检查（行 5-9）

- 检查 cur_id 是否为 0（空状态）
- 若为 0 直接跳转到退出流程，避免无效节点操作

3. 词频统计（行 10-14）

- 从 cnt 数组中读取 cur_id 对应的词频值
- 将词频值加 1 后写回内存，完成计数器更新

4. 最大词频更新（行 15-22）

- 分支 1（新最大值）：
 - 若当前词频超过 max_cnt，更新 max_cnt 为新值
 - 清空 max_siz（重置极值数组索引）
- 分支 2（等于当前最大值）：
 - 将 cur_id 写入 max_id 数组末尾
 - 递增 max_siz 记录新的极值数量

5. 状态重置（行 23-25）

- 将 cur_id 重置为 0，准备接收下一个单词的插入操作

6. 退出清理（行 26-29）

- 恢复保存的寄存器
- 销毁栈帧并返回调用点

3.2.4 output.s

1. 函数基本结构与内存布局

- 输出缓冲区定义及常量声明：

```
1 .equ    MAXL, 1000
2
3 .section .bss
4     .lcomm output_buffer, MAXL*2
5
6 # 全局符号，由C文件提供
7     .extern content          # char  content[]
8     .extern father          # int   father[]
9     .extern puts
```

- 函数签名定义：

```
1 .section .text
2     .globl  output
3     .type   output, @function
4
5 # void output(int id)
6 # %edi 为参数 id
```

2. 栈帧与寄存器保护

- 建立栈帧并保存 %rbx（被调用者保存寄存器），分配 16 字节局部空间
- 初始化局部变量 cnt=0（记录字符收集数量）

```
1     pushq   %rbp
2     movq    %rsp, %rbp
3     pushq   %rbx                # callee-saved
4
5     subq    $16, %rsp           # 局部变量留 16 字节
6     movl    $0, -4(%rbp)        # int cnt = 0;
```

3. 字符反向收集

- 循环条件：当 id ≠ 0 时持续遍历（father 数组实现树结构回溯）
- 操作流程：

```
1 .Lcollect:
2     testl   %edi, %edi
3     je      .Lreverse
4
5     movl    -4(%rbp), %eax       # eax→rax = cnt（零扩展）
6     lea     content(%rip), %rbx
7     movzbl  (%rbx,%rdi,1), %ecx  # cl = content[id]
8
9     lea     output_buffer(%rip), %rbx
10    movb     %cl, (%rbx,%rax,1)   # output[cnt] = cl
11
12    incl     -4(%rbp)            # ++cnt
13
14    lea     father(%rip), %rbx
15    movl     (%rbx,%rdi,4), %edi  # id = father[id]
16    jmp      .Lcollect
```

- 特性：收集的字符顺序为从叶子到根的反序（例如树中路径为 3→2→1，收集顺序为 [3], [2], [1]），需要后续反转

4. 字符串反转

- 指针初始化：
 - rsi 指向最后一个字符（output_buffer + cnt-1）

– rdi 指向结束符位置 (output_buffer + cnt)

- 反转过程：通过双指针从两端向中间交换字符，最终得到正序字符串

```
1 .Lreverse:
2     movl    -4(%rbp), %eax          # eax→rax = cnt
3     movl    %eax, %edx              # edx→rdx = cnt
4     decl    %eax                    # eax→rax = cnt-1
5
6     lea     output_buffer(%rip), %rbx
7     leaq    (%rbx,%rax,1), %rsi      # rsi = &output[cnt-1]
8     leaq    (%rbx,%rdx,1), %rdi      # rdi = &output[cnt]
9     movq    %rbx, %rcx              # rcx = &output[0]
10
11 .Lrev_loop:
12     cmpq    %rcx, %rsi
13     jb      .Lafter_reverse
14     movzbl  (%rsi), %eax
15     movb    %al, (%rdi)
16     decq    %rsi
17     incq    %rdi
18     jmp     .Lrev_loop
```

5. 终止符与输出

- 安全设计：在 output_buffer[cnt*2] 写入 \0（确保缓冲区溢出时仍能终止）
- 栈对齐优化：调用 puts 前调整栈指针保证 16 字节对齐
- 输出地址：将反转后的字符串起始地址 (output_buffer + cnt) 传给 puts

```
1 .Lafter_reverse:
2     movl    -4(%rbp), %eax
3     sall    $1, %eax                # cnt * 2
4     lea     output_buffer(%rip), %rbx
5     movb    $0, (%rbx,%rax,1)       # '\0'
6
7     /* ----- 调用 puts ----- */
8     movl    -4(%rbp), %eax
9     lea     output_buffer(%rip), %rdi
10    leaq    (%rdi,%rax,1), %rdi      # rdi = 输出字符串地址
11
12    subq    $8, %rsp                 # 对齐栈 -> 16B
13    call    puts@PLT
14    addq    $8, %rsp
15
16    popq    %rbx
17    leave
18    ret
```

4 编程调试心得体会

- C 语言的代码在对单模块的调试中发挥了重要作用。通过 `make debug-(模块名)` 命令可以将 C 语言编写的该模块替换为相应的汇编.s 文件，在其他函数仍是 C 语言的情况下进行测试，实现单模块的调试。这是一种效率较高的 debug 方式。
- 各个成员在代码编写时都遇到了“位数”的问题，实验目标是实现 64 位的程序，makefile 的编译指令中也是 64 位的编译指令。但各组员编写的时候都写成了 32 位程序，导致测试的时候出现错误。因此，在编写汇编代码时，需要注意指令的位数，以及数据类型是否正确。
- 在程序调试过程中，我们遇到了一个特别有趣的问题：程序执行正常，但没有任何输出显示在终端上。经过调试发现，程序的逻辑是正确的，但输出内容似乎“消失”了。经过仔细分析 main.s 的代码，我们发现问题出在程序的退出方式上：

```
1      movl    $60, %eax    # syscall number for exit
2      xorl    %edi, %edi   # exit code 0
3      syscall
```

这段代码使用了直接的系统调用来终止程序，而没有通过 C 运行时库的 `exit()` 函数。问题在于，直接使用 `syscall` 退出程序会绕过 C 运行时的清理过程，导致标准输出流的缓冲区没有被刷新，输出内容仍然滞留在缓冲区中。

我们通过两种方法解决了这个问题：

1. 在 `syscall` 之前调用 `fflush(NULL)` 手动刷新所有输出缓冲区
2. 替换直接系统调用，改为调用 C 库的 `exit()` 函数，让它来处理程序终止时的缓冲区刷新

这个经验让我们深刻理解了汇编代码与 C 运行时环境之间的交互，特别是关于输出缓冲机制和程序正确终止的重要性。

5 源代码及测试文件

本项目的 github 仓库地址为：<https://github.com/ucas-asm2025-team/assignment1>，其中包含了所有的源代码和测试文件。

6 运行测试结果截图

```
Chavapa@Chavapa-linux101:~/asm-lab/assignment1$ make test
Assembling main.s -> build/asm/main.o
Assembling not_alpha.s -> build/asm/not_alpha.o
Assembling variable.s -> build/asm/variable.o
Linking -> build/count-debug
Running automated tests for all modules ...
Building debug versions for each module...
make[1]: 进入目录 "/home/Chavapa/asm-lab/assignment1"
make[1]: "build/count-debug-alpha"已是最新。
make[1]: 离开目录 "/home/Chavapa/asm-lab/assignment1"
make[1]: 进入目录 "/home/Chavapa/asm-lab/assignment1"
Building debug for module not_alpha
gcc -c c/alpha.c -o build/c/alpha.o
gcc -c c/debug.c -o build/c/debug.o
gcc -c c/main.c -o build/c/main.o
as not_alpha.s -o build/asm/not_alpha.o
gcc -c c/output.c -o build/c/output.o
gcc -c c/variable.c -o build/c/variable.o
Linking -> build/count-debug-not_alpha
make[1]: 离开目录 "/home/Chavapa/asm-lab/assignment1"
make[1]: 进入目录 "/home/Chavapa/asm-lab/assignment1"
make[1]: "build/count-debug-output"已是最新。
make[1]: 离开目录 "/home/Chavapa/asm-lab/assignment1"
make[1]: 进入目录 "/home/Chavapa/asm-lab/assignment1"
Building debug for module variable
gcc -c c/alpha.c -o build/c/alpha.o
gcc -c c/debug.c -o build/c/debug.o
gcc -c c/main.c -o build/c/main.o
gcc -c c/not_alpha.c -o build/c/not_alpha.o
gcc -c c/output.c -o build/c/output.o
as variable.s -o build/asm/variable.o
Linking -> build/count-debug-variable
make[1]: 离开目录 "/home/Chavapa/asm-lab/assignment1"
Running tests...
testing alpha... OK
testing not_alpha... OK
testing output... OK
testing variable... OK
testing all... OK
Chavapa@Chavapa-linux101:~/asm-lab/assignment1$
```

```
~/asm-lab/assignment1/test/reference.txt - Mousepad
文件(F) 编辑(E) 搜索(S) 视图(V) 文档(D) 帮助(H)
BUILD
AS: 3
ASM: 8
Assembling: 1
BUILD: 39
Building: 2
C: 20
CC: 6
Clean: 1
Compiling: 2
DEBUG: 7
Debug: 2
Default: 1
Ensure: 1
LD: 3
Linking: 4
MAKE: 2
MODULES: 4
Makefile: 5
OBJ: 3
OBJ3: 7
OK: 2
Object: 1
PHONY: 1
Running: 3
S: 5
SRC: 2
SRCS: 4
TEST: 16
Test: 2
Tools: 1
about: 1
all: 8
and: 1
answer: 1
as: 2
asm: 1
automated: 1
basename: 1
binary: 1
build: 5
building: 1

~/asm-lab/assignment1/test/debug-all.txt - Mousepad
文件(F) 编辑(E) 搜索(S) 视图(V) 文档(D) 帮助(H)
BUILD
AS: 3
ASM: 8
Assembling: 1
BUILD: 39
Building: 2
C: 20
CC: 6
Clean: 1
Compiling: 2
DEBUG: 7
Debug: 2
Default: 1
Ensure: 1
LD: 3
Linking: 4
MAKE: 2
MODULES: 4
Makefile: 5
OBJ: 3
OBJ3: 7
OK: 2
Object: 1
PHONY: 1
Running: 3
S: 5
SRC: 2
SRCS: 4
TEST: 16
Test: 2
Tools: 1
about: 1
all: 8
and: 1
answer: 1
as: 2
asm: 1
automated: 1
basename: 1
binary: 1
build: 5
building: 1
```

图 1: 运行测试结果截图 (左: 终端测试; 右: 输出结果比对)