汇编语言第一次大作业实验报告

小组成员：袁晨圃、王荤璠、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：

```
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
cnt[MAXN]                // 记录每个单词出现的次数
father[MAXN]             // 记录每个节点的父节点ID，用于回溯
content[MAXN]            // 存储每个节点对应的字符
max_id[MAXN]             // 存储出现次数最多的单词的节点ID
```

  - 关键常量定义：

```
#define MAXN          2500  // 字典树的最大节点数
#define MAXL          100   // 单词的最大长度
#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：

```
int main();      // 读入输入，调用子模块
void alpha();    // 处理字母，构建字典树
void not_alpha(); // 处理非字母，分割单词
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
- **王荤璠**：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：

```
.section .data
son:      .fill MAXN*CHARSET_SIZE, 4, 0  # int son[MAXN][CHARSET_SIZE] = {0}
cnt:      .fill MAXN, 4, 0               # int cnt[MAXN] = {0}
father:   .fill MAXN, 4, 0               # int father[MAXN] = {0}
max_id:   .fill MAXN, 4, 0               # int max_id[MAXN] = {0}
content:  .fill MAXN, 1, 0               # char content[MAXN] = {0}
tot:      .long 0                        # int tot = 0
max_cnt:  .long 0                        # int max_cnt = 0
max_siz:  .long 0                        # int max_siz = 0
ch:       .long 0                        # int ch = 0
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：

```
.globl _start
.extern isalpha, alpha, not_alpha, output, debug, fflush
.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

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

- **字符分类处理**：
  - **判断是否为字母**：

```
if_cond:
    movl    ch(%rip), %edi
    call    isalpha
    testl   %eax, %eax
    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

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

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

- **退出程序**：使用 syscall 直接调用系统退出

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

```
movl  cur_id(%rip), %eax      # 当前节点ID
movl  ch(%rip), %ecx          # 输入字符ASCII值
imull $CHARSET_SIZE, %eax     # 计算二维数组行偏移（CHARSET_SIZE=128）
addl  %ecx, %eax              # 列偏移叠加（cur_id*128 + ch）
leaq  son(%rip), %rdx         # 获取son数组基址
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**

```
cmpl  $0, (%rsi)              # 检查子节点是否存在
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**

```
movl  (%rsi), %eax            # 获取子节点ID（新建或已有）
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：

```
.section .text
        .globl  output
        .type   output, @function

#  void output(int id)
#  %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）

```
        pushq   %rbp
        movq    %rsp, %rbp
        pushq   %rbx                    # callee-saved

        subq    $16, %rsp               # 局部变量留 16 字节
        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：

```
.Lcollect:
        testl   %edi, %edi
        je      .Lreverse

        movl    -4(%rbp), %eax          # eax→rax = cnt（零扩展）
        lea     content(%rip), %rbx
        movzbl  (%rbx,%rdi,1), %ecx     # cl = content[id]

        lea     output_buffer(%rip), %rbx
        movb    %cl, (%rbx,%rax,1)      # output[cnt] = cl

        incl    -4(%rbp)                # ++cnt

        lea     father(%rip), %rbx
        movl    (%rbx,%rdi,4), %edi     # id = father[id]
        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

```
.Lreverse:
        movl    -4(%rbp), %eax          # eax→rax = cnt
        movl    %eax, %edx              # edx→rdx = cnt
        decl    %eax                    # eax→rax = cnt-1

        lea     output_buffer(%rip), %rbx
        leaq    (%rbx,%rax,1), %rsi     # rsi = &output[cnt-1]
        leaq    (%rbx,%rdx,1), %rdi     # rdi = &output[cnt]
        movq    %rbx, %rcx              # rcx = &output[0]

.Lrev_loop:
        cmpq    %rcx, %rsi
        jb      .Lafter_reverse
        movzbl  (%rsi), %eax
        movb    %al, (%rdi)
        decq    %rsi
        incq    %rdi
        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

```
.Lafter_reverse:
        movl    -4(%rbp), %eax
        sall    $1, %eax                # cnt * 2
        lea     output_buffer(%rip), %rbx
        movb    $0, (%rbx,%rax,1)       # '\0'

        /* ---------- 调用 puts ---------- */
        movl    -4(%rbp), %eax
        lea     output_buffer(%rip), %rdi
        leaq    (%rdi,%rax,1), %rdi     # rdi = 输出字符串地址

        subq    $8, %rsp                # 对齐栈 -> 16B
        call    puts@PLT
        addq    $8, %rsp

        popq    %rbx
        leave
        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

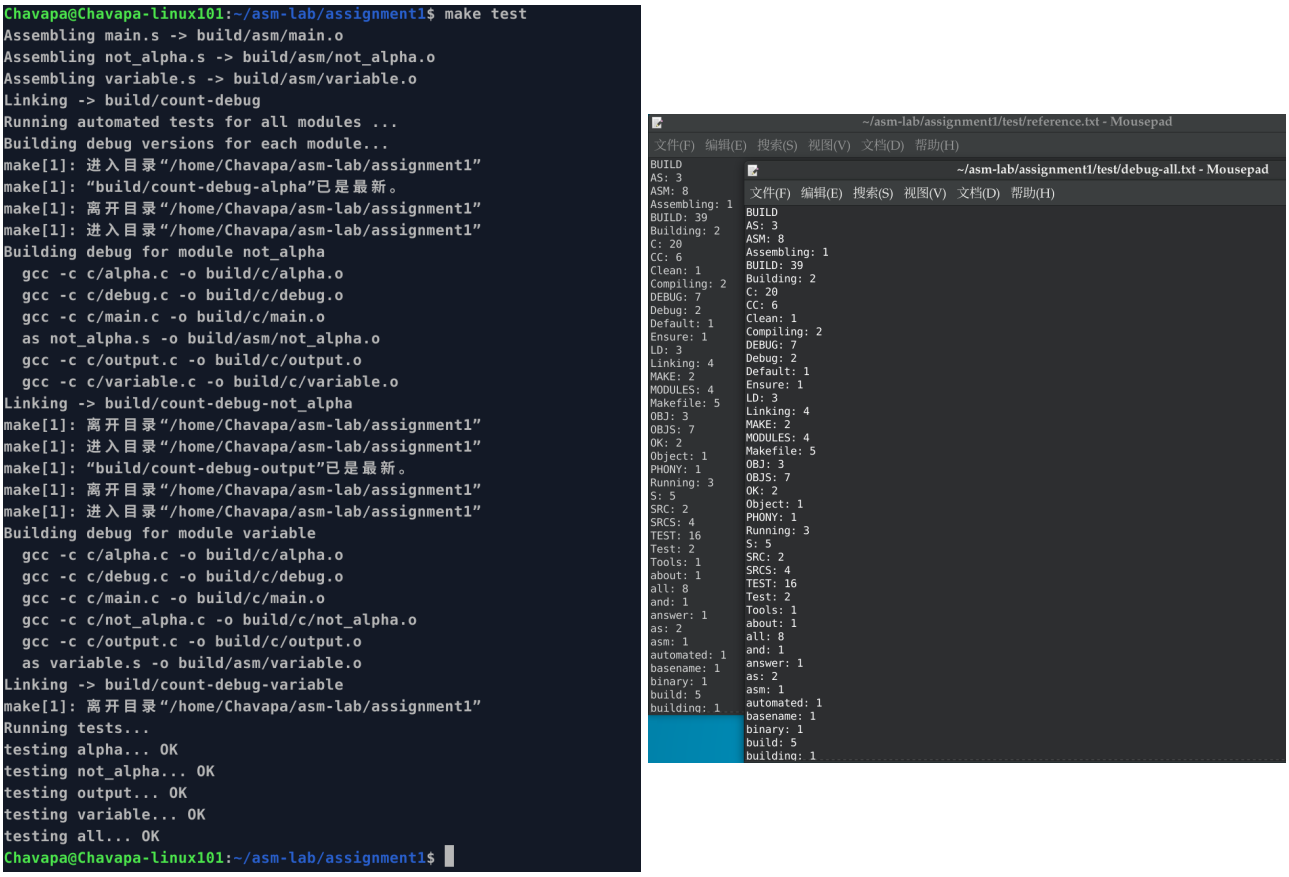

图 1: 运行测试结果截图（左：终端测试；右：输出结果比对）