《汇编语言》课程
大作业报告

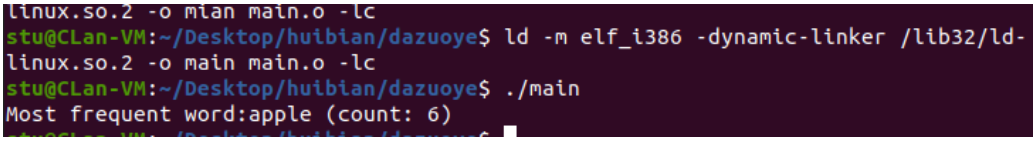
1 汇编程序编写思路与源代码

由于本次作业所写的代码较长, 于是完整代码添加在和本报告统一目录下的文件中, 命名为 main.s, 在本报告中, 对部分关键代码段进行讲解

1.1 代码运行输入与运行结果

```
apple banana apple banana orange orange pear
avacado apple apple pineapple grape banana apple
coconut pear grape apple
```

程序运行截图如下



1.2 代码编写思路

在本次实验中, 我们小组的主要思路是根据 sys_open 和 sys_read 的性质, 一次性读入所有的字符, 并且利用 ecx 寄存器来保存输入文件的字符数, 并且利用剩余字符数来判断是否读入完成.

对于单词的出现和计数, 我们定义了一个 36*100 字节的结构体, 其中前 32 个字节用于存储单词, 后四个字节用于存储单词出现的次数

我们判断单词的结束是通过判断当前读入字符是否是换行符一类的分隔符来判定的, 如果读入的字符是这类分隔符, 则结束当前单词的读入, 并把这个单词写入我们分配的内存中

值得一提的是, 在出现两个分隔符相邻在一起时, 我们应该如何处理? 我们次啊用的方法是, 利用寄存器 ebx 的值来记录我们程序当前读入的状态, 如果 ebx 为 0, 则表示我们正在输入一个单词, 如果 ebx 为 1, 则表示我们已经输入完了一个单词, 具体读入的情况和 ebx 的值如下

1. **ebx=0, 读入一个字母　说明这时正在读入一个单词, 而这个新读入的字母是这个单词的一部分**
2. **ebx=1, 读入一个字母　说明上一个单词的读入已经完成, 而这个新的字母是下一个单词的首字母**
3. **ebx=0, 读入一个分隔符　说明一个单词的读入未结束, 而读到这个分隔符说明这个单词已经结尾, 将 ebx 置为 1, 用来标志开启新一轮单词的读入**
4. **ebx=1, 读入一个分隔符　还没有开启一个单词的读入, 读到一个分隔符, 此时不考虑, 直接读入下一个字符**

每次我们读到一个单词, 将这个单词存在 tempword 中, 我们需要将这个单词与 word 中的单词进行比较, 如果有相同的单词, 则将这个单词的 word_count 加一, 否则将这个单词添加入 word 中 (这些添加利用 str_copy 来实现), 并将他的计数记为 1

至于出现次数最频繁的单词, 我们在读入的过程中, 只需要将当前单词的计数和当前最大计数进行比较, 如果大于当前最大计数, 则更新当前最大计数和当前最大单词

1.3 代码段讲解

下面对一些关键的代码段进行部分讲解

1.3.1 结构体定义

```
    .equ word_text, 0  # 单词字符串（32字节）
    .equ word_count, 32 # 出现次数（4字节）
    .equ word_size, 36 # 结构体总大小 = 36字节
```

1.3.2 求字符串长度

```
strlen:
    movl $0, %eax
.strlen_loop:
    cmpb $0, (%edi, %eax)
    je .strlen_done
    incl %eax
    jmp .strlen_loop
.strlen_done:
    ret
```

利用 eax 来计算首地址为 edi 的字符串的长度, 通过比较 edi 指针所指的当前字符是否为'0' 来判断字符串是否结束, 如果结束, 则返回 eax+1

字符串长度在本程序中的运用主要是在主函数中, 在输出结果时在终端打印正确的字符数

1.3.3 字符串赋值

```
str_copy:
    pushl %esi
    pushl %edi
.str_copy_loop:
    movb (%esi), %al
    movb %al, (%edi)
    incl %esi
    incl %edi
    testb %al, %al
    jnz .str_copy_loop
    popl %edi
    popl %esi
    ret
```

将 esi 中的字符串按字节逐个复制到 edi 中, 并且利用栈来保存字符串的起始地址

1.3.4 字符串比较

```
str_compare:
    pushl %ebx
    pushl %edi
.str_compare_loop:
    movb (%ebx), %al
    cmpb (%edi), %al
    jne .str_compare_not_equal
    testb %al, %al
    jz .str_compare_equal
    incl %ebx
    incl %edi
    jmp .str_compare_loop
.str_compare_equal:
    xorl %eax, %eax
    jmp .str_compare_done
.str_compare_not_equal:
    movl $1, %eax
.str_compare_done:
    popl %edi
    popl %ebx
    ret
```

将 ebx 和 edi 中的字符逐个按字节比较, 同样利用栈来保存字符串的起始地址
相等时返回 0, 否则返回 1

1.3.5 检查读入的字符类型

```
# 检查是否是字母
is_alpha:
    cmpb $'a', %al
    jl .not_lower
    cmpb $'z', %al
    jle .is_alpha_yes
.not_lower:
    cmpb $'A', %al
    jl .is_alpha_no
    cmpb $'Z', %al
    jg .is_alpha_no
.is_alpha_yes:
    xorl %eax, %eax     # ZF=1
    ret
.is_alpha_no:
    movl $1, %eax       # ZF=0
    ret

# 检查是否是标点
is_punctuation:
```

```
    xorl %eax, %eax
    ret
    movl $1, word_count(%edi)
```

检查读入的字符是否是字母, 如果是字母, 则返回 0, 否则返回 1, 对于检查分隔符, 直接返回 1

1.3.6 存储单词

这段代码是本程序的关键, 具体思路在上一部分已经提及, 在此不多赘述

```
save_word:
    pushl %esi
    pushl %edi
    pushl %ebx
    pushl %ecx

    # 遍历检查单词是否已存在
    movl $words, %esi       # ESI = 单词数组起始地址
    movl total_word, %ecx   # ECX = 当前单词数
    leal temp_word, %edi    # EDI = 要查找的单词地址

.search_loop:
    testl %ecx, %ecx
    jz .add_new_word        # 遍历完未找到，添加新单词

    # 计算当前单词地址（EAX = (total_word - ECX) * word_size）
    movl total_word, %eax
    subl %ecx, %eax
    movl $word_size, %edx
    mull %edx
    leal words(%eax), %ebx  # EBX = 当前单词结构体地址

    # 比较单词（参数: EBX=数组中的单词, EDI=temp_word）
    pushl %ecx
    pushl %edi
    pushl %ebx
    call str_compare
    addl $8, %esp           # 清理栈（保留ECX）
    testl %eax, %eax
    popl %ecx
    jz .word_found          # 找到相同单词

    decl %ecx
    jmp .search_loop

.word_found:
    # 单词已存在，增加计数
    incl word_count(%ebx)

    # 检查是否需要更新 max_word/max_count
```

```
    movl word_count(%ebx), %eax
    cmpl max_count, %eax
    jle .done

    # 更新最高频单词
    movl %eax, max_count
    leal word_text(%ebx), %esi
    leal max_word, %edi
    call str_copy
    jmp .done

.add_new_word:
    # 添加新单词（确保数组未满）
    movl total_word, %eax
    cmpl $100, %eax
    jge .overflow

    # 计算存储位置（EAX * word_size）
    movl $word_size, %edx
    mull %edx
    leal words(%eax), %edi

    # 复制单词到数组
    leal temp_word, %esi
    call str_copy

    # 初始化计数为1 movl $1, word_count(%edi)

    addl $32, %edi
    movl $1, (%edi)
    subl $32, %edi

    # 如果是第一个单词，直接设为 max_word
    cmpl $0, max_count
    jne .skip_first
    movl $1, max_count
    leal max_word, %edi
    call str_copy

.skip_first:
    # 增加总单词数
    incl total_word

.done:
    popl %ecx
    popl %ebx
    popl %edi
    popl %esi
    ret
```

```
.overflow:
    # 处理数组已满的情况
    jmp .done
```

要注意这里的寻址方式, 因为 edi 所指向的是一个结构体单元, 所以如果要对单词数目进行访问, 寻址方式应该是 32(edi), 才能正确的对结构体中后 4 个字节的整型数据进行访问

1.3.7 分割单词

这段代码同样是本程序的关键部分, 这段代码将一个完整的文本段落分为一个个单词存入 word 堆中, 而具体的分割方式在思路部分也已经提及, 在此从简

```
split_words:
    movl $buffer, %esi
    movl %eax, %ecx     # ECX = 读取的字节数
    xorl %ebx, %ebx     # EBX = 是否在单词中 (0/1)
    leal temp_word, %edi

.split_next_char:
    cmpl $0, %ecx
    jle .split_done

    movb (%esi), %al

    # 检查是否是字母
    push %eax
    call is_alpha
    test %eax, %eax
    pop %eax
    jz .handle_alpha

    # 检查是否是标点
    push %eax
    call is_punctuation
    test %eax, %eax
    pop %eax
    jz .handle_punct

    # 其他字符跳过
    jmp .split_skip_char

.handle_alpha:
    cmpl $1, %ebx
    je .start_new_word
    # 继续当前单词
    movb %al, (%edi)
    incl %edi
    jmp .split_skip_char

.start_new_word:
```

```
    movl $0, %ebx
    leal temp_word, %edi
    movb %al, (%edi)
    incl %edi
    jmp .split_skip_char

.handle_punct:
    testl %ebx, %ebx
    jne .split_skip_char
    # 结束当前单词
    movb $0, (%edi)
    call save_word
    movl $1, %ebx
    jmp .split_skip_char

.split_skip_char:
    incl %esi
    decl %ecx
    jmp .split_next_char

.split_done:
    # 处理缓冲区末尾的单词
    cmpl $1, %ebx
    jne .split_exit
    movb $0, (%edi)
    call save_word

.split_exit:
    ret
```

根据在思路部分的介绍, 一共有四种读入的情况, 根据不同情况对当前的单词处理

1.3.8 输出字符串个数准备

```
int_to_str:
    pushl %eax
    pushl %ebx
    pushl %ecx
    pushl %edx
    movl $0, %ecx       # 计数器
    movl $10, %ebx      # 除数

.convert_loop:
    xorl %edx, %edx
    divl %ebx           # %eax = %eax / 10, %edx = %eax % 10
    addb $'0', %dl      # 转换为 ASCII 字符
    movb %dl, num_buffer(%ecx)
    incl %ecx
    cmpl $0, %eax
```

```
    jne .convert_loop

    # 反转字符串
    pushl %ecx
    movl $0, %eax
    decl %ecx
.reverse_loop:
    cmp %eax, %ecx
    jle .reverse_done
    movb num_buffer(%eax), %dl
    movb num_buffer(%ecx), %dh
    movb %dl, num_buffer(%ecx)
    movb %dh, num_buffer(%eax)
    incl %eax
    decl %ecx
    jmp .reverse_loop

.reverse_done:
    incl %eax
    movb $0, num_buffer(%eax) # 字符串结束符
    popl %ecx
    movl %ecx, %edx         # 字符串长度
    popl %edx
    popl %ecx
    popl %ebx
    popl %eax
    ret
```

在 x86-32 语言中, 我们不能直接输出一个整型, 于是我们需要进行一步操作, 来将整型 max_count 转化为字符串, 存在 num_buffer 中, 另外值得一提的是, 我们采用的求余的转化过程中, 需要将第一次得到的结果进行反转才能得到正确的结果

1.3.9 主函数

```
_start:
    # 打开文件
    movl $5, %eax       # sys_open
    movl $filename, %ebx
    movl $0, %ecx       # O_RDONLY
    int $0x80

    cmpl $0, %eax
    jl .open_error

    movl %eax, file_fd

    # 读取文件
    movl $3, %eax       # sys_read
    movl file_fd, %ebx
```

```
    movl $buffer, %ecx
    movl $4096, %edx
    int $0x80

    movl %eax, %edx
    movb $0, buffer(%eax) # 在读取的字节数后添加 \0

    cmpl $0, %eax
    jle .read_error

    # 分割单词
    movl %eax, %ecx
    call split_words

    # 输出结果
    call print_result

    # 关闭文件
    movl $6, %eax       # sys_close
    movl file_fd, %ebx
    int $0x80

    # 退出
    movl $1, %eax
    movl $0, %ebx
    int $0x80

.open_error:
    movl $4, %eax       # sys_write
    movl $1, %ebx
    movl $error_open, %ecx
    movl $error_open_len, %edx
    int $0x80
    jmp .exit_error

.read_error:
    movl $4, %eax       # sys_write
    movl $1, %ebx
    movl $error_read, %ecx
    movl $error_read_len, %edx
    int $0x80

.exit_error:
    movl $1, %eax
    movl $1, %ebx
    int $0x80

print_result:
    # 输出前缀 "Most frequent word: '"
    movl $4, %eax       # sys_write
```

```
    movl $1, %ebx       # stdout
    movl $result_prefix, %ecx
    movl $19, %edx      # 字符串长度
    int $0x80

    # 输出最高频单词（max_word）
    movl $max_word, %edi
    call strlen
    movl %eax, %edx
   movl $4, %eax
   movl $1, %ebx
   movl $max_word, %ecx
# movl $32, %edx      # 最大32字节（实际长度需计算）
   int $0x80

    # 输出后缀 "' (count: "
    movl $4, %eax
    movl $1, %ebx
    movl $result_suffix, %ecx
    movl $10, %edx
    int $0x80

    # 输出数字（max_count）
    movl max_count, %eax
    decl %eax
    call int_to_str
    movl $num_buffer, %edi
    call strlen
    movl %eax, %edx
    movl $4, %eax
    movl $1, %ebx
    movl $num_buffer, %ecx
    int $0x80

    # 输出结尾 ")\n"
    movl $4, %eax
    movl $1, %ebx
    movl $result_end, %ecx
    movl $2, %edx
    int $0x80
    ret
```

在主函数之中, 我们进行了外部文件的读入, 以及各种输出的及逆行, 另外调用了 strlen 来计算字符串的长度, 使我们最终按照字符串长度正确地输出

2 编程调试心得体会

2.1 调试方法

在本次实验中, 我们利 gdb 工具, 利用 gdb n 配合 info r 来进行单步调试, 来检测各个寄存器的值是否是我们期望的结果, 利用 break 添加断点来进行整段的调试

2.2 心得体会

1. 在本次实验的调试过程中, 我们仅仅了解到可以利用 info r 来查看各个寄存器的值, 但是各个寄存器的值在很大部分的时间内都是地址, 而我们目前位置找不到命令来对地址所指向的内容进行访问, 查看. 内存也是同理, 例如在本次实验中, 我们没有办法对 word 或者 tempword 以及 num_buffer 进行查看, print 的结果也不符合我们的预期, 我们有待对 gdb 的进一步深入学习与实践
2. 在本次实验中我们深刻感受到了重复利用多个寄存器所带来的麻烦以及栈的强大, 我们在调用函数的时候, 一定一定要记得利用栈来对寄存器的值进行保存, 以免出现我们不想看到的寄存器内容的覆盖
3. 我们也更深层次的意识到模块化对于设计任务的重要性, 进行模块化能够有效的进行分工, 并且在极大程度上降低调试的复杂程度与过程

3 小组分工

1. 薛翼舟：实验报告撰写, 分割单词部分编写与调试 debug
2. 赵夕然: 存储单词段的编写, 代码整体框架构思与 debug
3. 葛景源: strlen, str_compare 等对字符串操作函数的编写
4. 彭锡铭: 主函数编写与子函数整合以及整型转字符串的编写