汇编语言程序设计

第十一节　80X86-64汇编编程

目录

1. **Hello World**
2. 第一个汇编程序
3. 汇编示例程序

Hello World

```
#include <stdlib.h>
#include <stdio.h>

int main()
{
	printf("Hello world\n");
	exit(0);

	return 0;
}
```

- **命令行输入**
  - $ gcc -S -Og helloworld.c
    - This will produce a file named helloworld.s
  - 可以采用不同的编译优化级别 (–O*n*)
    - *n* : the optimization level
- **gcc产生的汇编代码实例**

```
        .file   "hello.c"
        .section        .rodata
.LC0:
        .string   “Hello world”

        .text
        .globl  main
        .type   main, @function
main:
        subq    $8, %rsp
        movl    $.LC0, %edi
        call    puts
        movl    $0, %edi
        call    exit
```

```
gcc -S –Og helloworld.c
```

- 汇编代码中以“.”开头的行都是**汇编指示（Directives）**，如“.file”、“.def”、“.text”等，用以指导汇编器如何进行汇编
  - 其中“.file”、“.def”、”CFI”等均用于调试（可以将其忽略）

- 以“:”结尾的字符串（如“main:”）是用以表示变量或者函数的地址的符号（Symbol）
- 其它均为汇编指令

- 示例：“.globl main”
  - 指示汇编器符号“main”是全局的，这样同一程序的其它模块可以引用它
- “.LC0”则不是全局可见的

```
.text                    #代码段，也可写为.section    .text
.p2align 4, ,15     #指定下一行代码的对齐方式：第1参数表示按2的多少次幂字节
对齐，第2参数表示对齐时额外空间用什么数据来填充，第3字节表示最多允许额外
填充多少字节。
```

- 按 16 字节对齐。

```
.section .rodata            #只读数据段
LC0:
           .string "Hello world”
```

```
.globl   c
.data    #也可写为.section      .data
.align 4
c:
           .long   1
```

请尝试解释一下含义

int c = 1; //初始化的全局变量

目录

1. Hello World

2. **第一个汇编程序**

3. 汇编示例程序

Linux汇编命令

- **as -o my-object-file.o my-file.s**
  - --gstabs //产生带调试信息的object文件

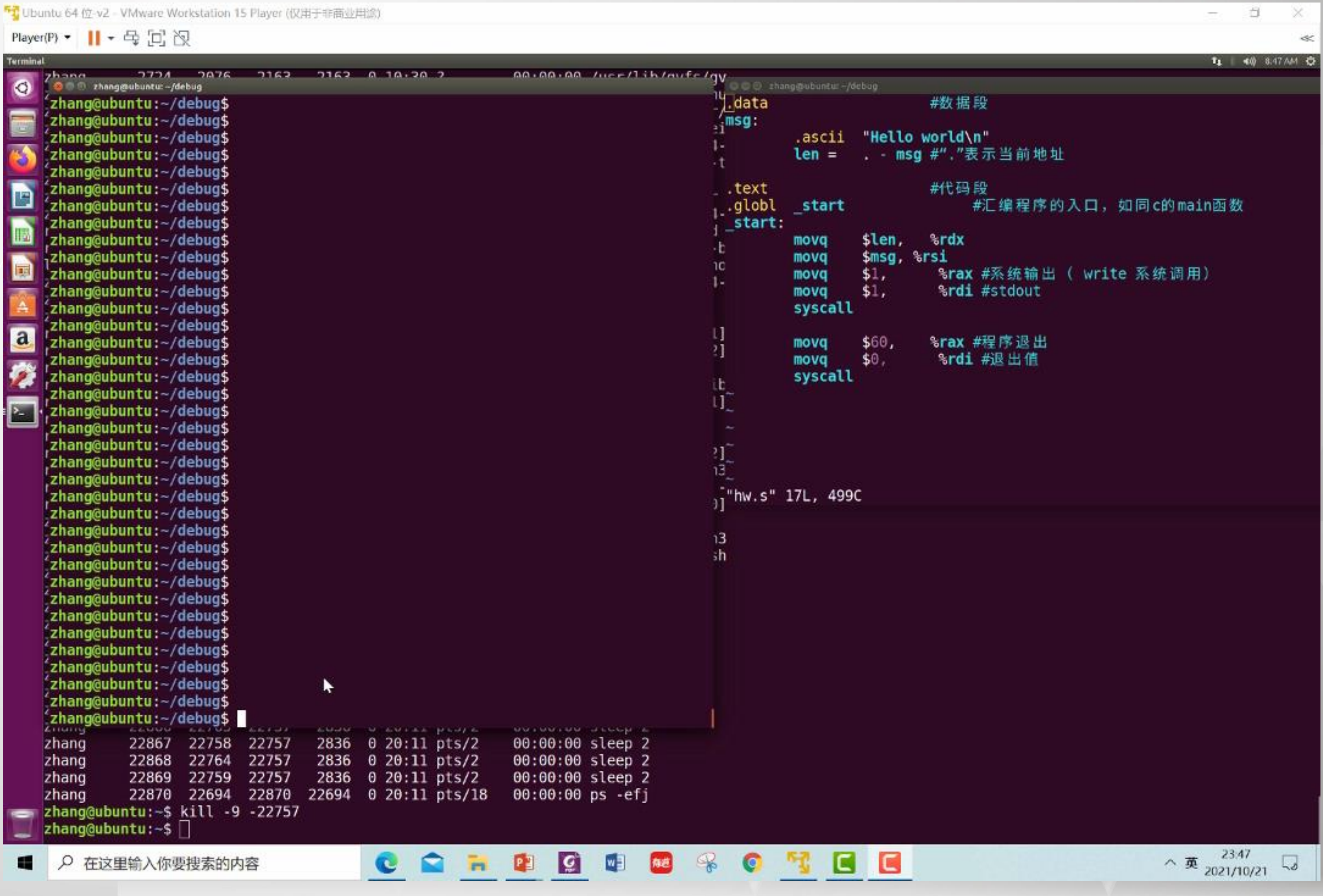
- **ld -o my-exe-file my-object-file.o**
  - 可以有多个.o文件
  - -g //调试信息

Hello World 示例

```
.data                          #数据段
msg:
     .ascii  "Hello world\n"
     len =   . - msg           # “.” 表示当前地址

.text                          #代码段
.globl  _start                 #汇编程序的入口，如同c的main函数
_start:
     movq    $len,   %rdx
     movq    $msg, %rsi
     movq    $1,     %rax #系统输出（ write 系统调用）
     movq    $1,     %rdi #stdout
     syscall

     movq    $60,    %rax #程序退出
     movq    $0,     %rdi #退出值
     syscall
```



```
zhang@ubuntu:~/debug$
zhang@ubuntu:~/debug$
zhang@ubuntu:~/debug$
zhang@ubuntu:~/debug$
zhang@ubuntu:~/debug$
zhang@ubuntu:~/debug$
zhang@ubuntu:~/debug$
zhang@ubuntu:~/debug$
zhang@ubuntu:~/debug$
zhang@ubuntu:~/debug$
zhang@ubuntu:~/debug$
zhang@ubuntu:~/debug$
zhang@ubuntu:~/debug$
zhang@ubuntu:~/debug$
zhang@ubuntu:~/debug$
zhang@ubuntu:~/debug$
zhang@ubuntu:~/debug$
zhang@ubuntu:~/debug$
zhang@ubuntu:~/debug$
zhang@ubuntu:~/debug$
zhang@ubuntu:~/debug$
zhang@ubuntu:~/debug$
zhang@ubuntu:~/debug$
zhang@ubuntu:~/debug$
zhang@ubuntu:~/debug$
zhang@ubuntu:~/debug$
zhang@ubuntu:~/debug$
zhang@ubuntu:~/debug$
zhang@ubuntu:~/debug$
zhang@ubuntu:~/debug$
zhang@ubuntu:~/debug$
zhang@ubuntu:~/debug$
zhang@ubuntu:~/debug$
zhang@ubuntu:~/debug$
zhang@ubuntu:~/debug$
zhang@ubuntu:~/debug$
zhang@ubuntu:~/debug$
```

```
.data                       #数据段
msg:
        .ascii  "Hello world\n"
        len =   . - msg #"."表示当前地址

.text                       #代码段
.globl  _start                   #汇编程序的入口，如同c的main函数
_start:
        movq    $len,   %rdx
        movq    $msg, %rsi
        movq    $1,      %rax #系统输出（ write 系统调用）
        movq    $1,      %rdi #stdout
        syscall

        movq    $60,    %rax #程序退出
        movq    $0,      %rdi #退出值
        syscall

"hw.s" 17L, 499C
```

```
zhang       2724   2076   2163   2163  0 10:30 ?        00:00:00 /usr/lib/gvfs/gv
zhang      22867  22758  22757   2836  0 20:11 pts/2    00:00:00 sleep 2
zhang      22868  22764  22757   2836  0 20:11 pts/2    00:00:00 sleep 2
zhang      22869  22759  22757   2836  0 20:11 pts/2    00:00:00 sleep 2
zhang      22870  22694  22870  22694  0 20:11 pts/18   00:00:00 ps -efj
zhang@ubuntu:~$ kill -9 -22757
zhang@ubuntu:~$ 
```

系统调用

- **X86-64 Linux 下的系统调用是通过中断系统调用（syscall）来实现的**
- **在执行 sycall指令时**
  - 寄存器 rax 中存放的是系统调用的功能号，而传给系统调用的参数则必须按顺序存放到寄存器 rdi， rsi， rdx， r10， r8, r9 中
  - 当系统调用完成之后，返回值可以在寄存器 rax中获得
    - 一般小于0表示错误

部分系统调用列表

Call Code (rax)	System Service	Description
0	**SYS_read**	**Read data**
		rdi = file descriptor (of where to read from) rsi = address of where to store data rdx = count of bytes to read
	If unsuccessful, returns negative value. If successful, returns count of characters actually read.	
1	**SYS_write**	**Write data**
		rdi = file descriptor (of where to write to) rsi = address of data to write rdx = count of bytes to write
	If unsuccessful, returns negative value. If successful, returns count of characters actually written.	

2	**SYS_open**	**Open a file.**

		rdi = address of NULL terminated file name
		rsi = file status flags (typically 0 RD0NLY)

	If unsuccessful, returns negative value. If successful, returns file descriptor.	
3	**SYS_close**	**Close an open file.**
		rdi = file descriptor of open file to close
	If unsuccessful, returns negative value.	
8	**SYS_lseek**	**Reposition the file read/write file offset.**

		rdi = file descriptor (of where to write to)
		rsi = offset
		rdx = origin

	If unsuccessful, returns negative value	

57	**SYS_fork**	**Fork current process.**
59	**SYS_execve**	**Execute a program**
		rdi = Address of NULL terminated string for name of program to execute.
60	**SYS_exit**	**Terminate executing process.**
		rdi = exit status (typically 0)
85	**SYS_creat**	**Open/Create a file.**

		rdi = address of NULL terminated file name
		rsi = file mode flags

	If unsuccessful, returns negative value. If successful, returns file descriptor.	
96	**SYS_gettimeofday**	**Get date and time of day**
		rdi = address of time value structure rsi = address of time zone structure
	If unsuccessful, returns negative value. If successful, returns information in the passed structures.	

处理命令行参数的示例

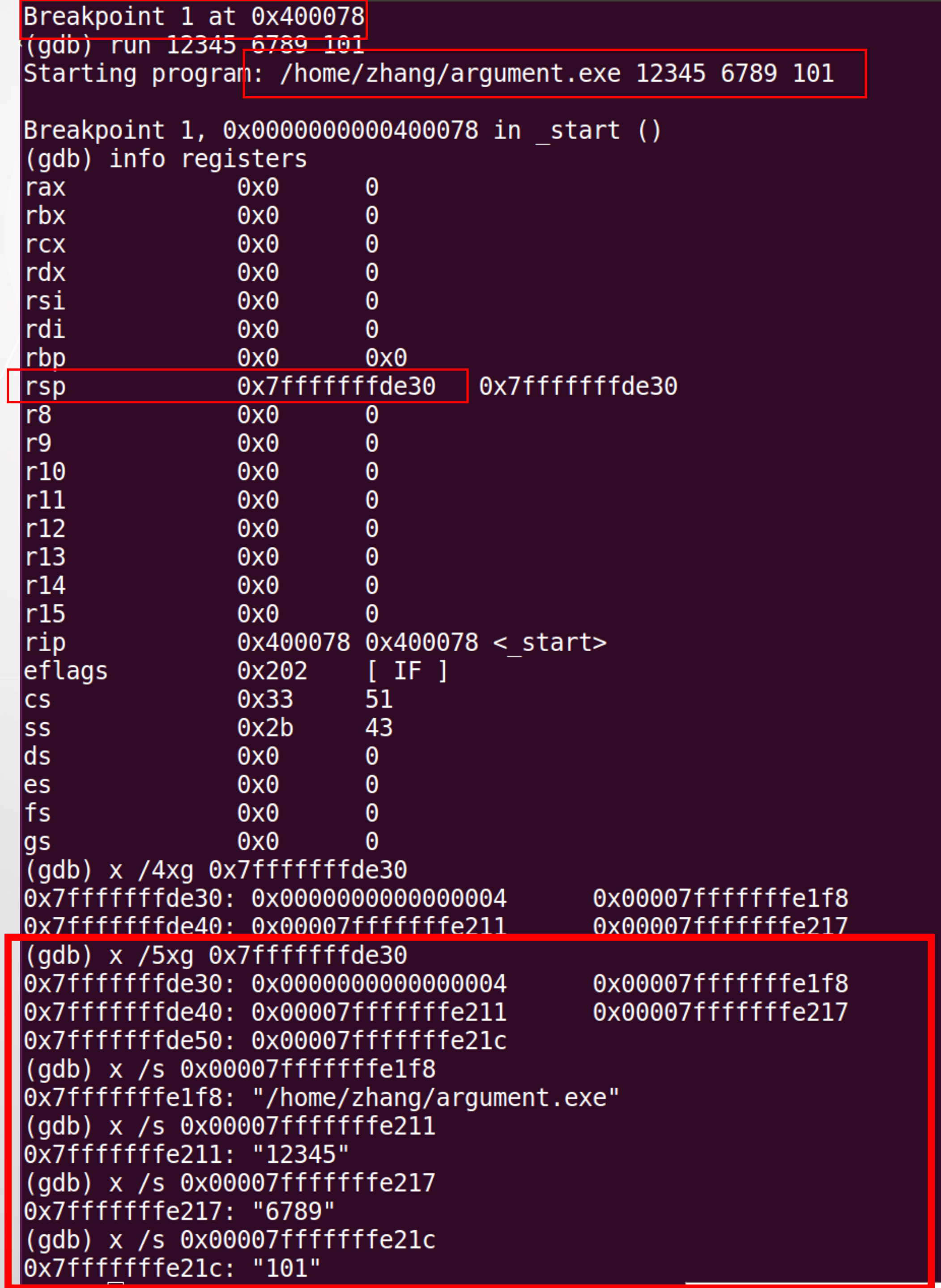
```
.text
.globl _start

_start:
        popq  %rsi              #argc
vnext:
        popq  %rsi              #
        testq  %rsi, %rsi       # 空指针表明结束
        jz      exit            # 即je
        movq  %rsi, %rbx
        xorq   %rdx, %rdx
strlen:
        movb  (%rbx), %al
        incq    %rdx
        incq    %rbx
        testb   %al, %al
        jnz      strlen
        movb  $10, -1(%rbx) #10是换行键
        movq   $1, %rax        # 系统调用号(sys_write)
        movq   $1, %rdi        # 文件描述符(stdout)
        syscall
        jmp vnext
exit:
        movq    $60, %rax #程序退出
        movq    $0,   %rdi #退出值
        syscall
```

相当于C语言形式：int main(int argc, char *argv[])

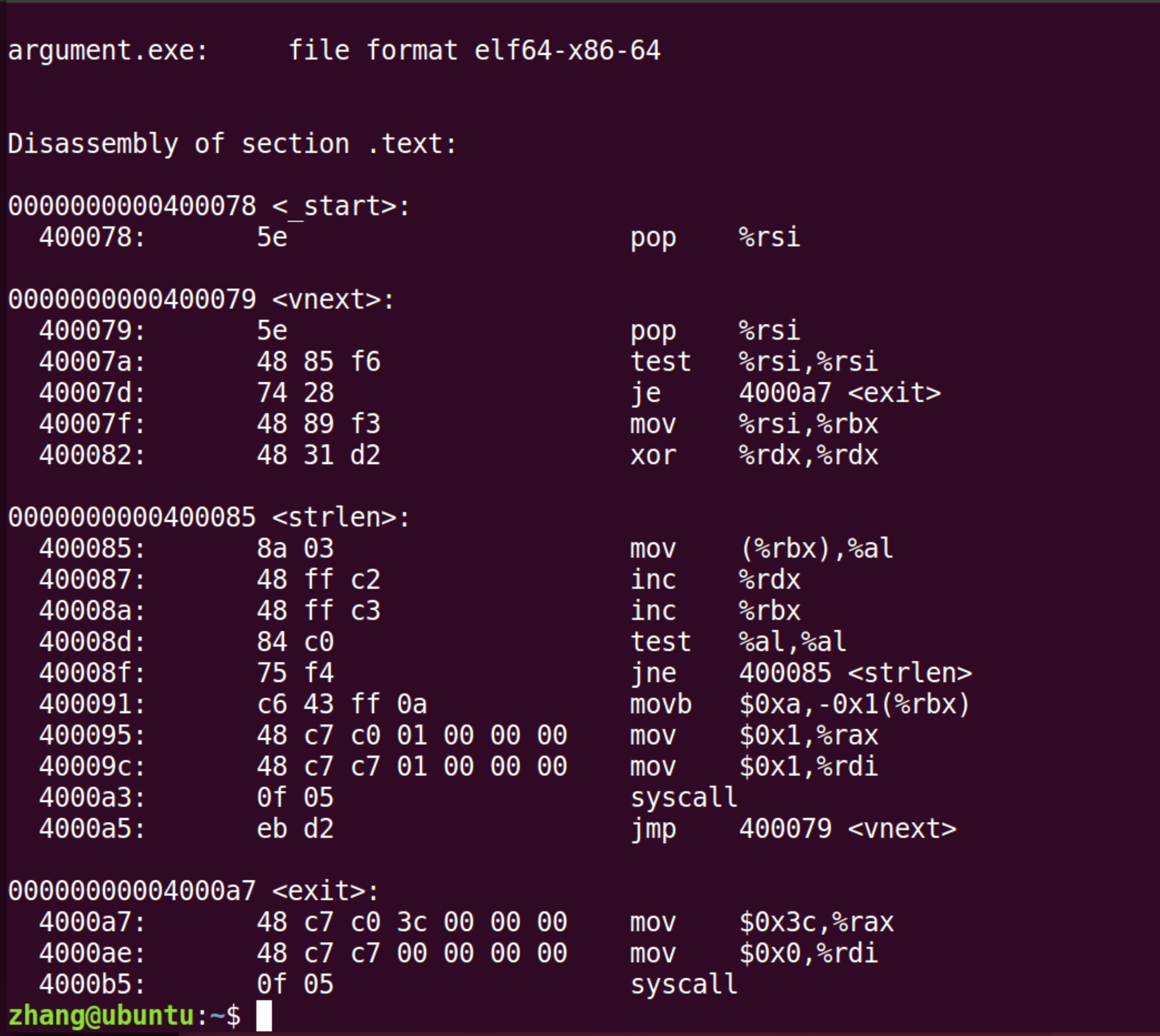
argv[0] = 'progname'
argv[1] = ‘arg1'
argv[2] = ‘arg2'
argv[3] = ‘arg3'
…

当一个可执行程序通过命令行启动时，命令行参数将被保存到栈中



```
Breakpoint 1 at 0x400078
(gdb) run 12345 6789 101
Starting program: /home/zhang/argument.exe 12345 6789 101

Breakpoint 1, 0x0000000000400078 in _start ()
(gdb) info registers
rax            0x0      0
rbx            0x0      0
rcx            0x0      0
rdx            0x0      0
rsi            0x0      0
rdi            0x0      0
rbp            0x0      0x0
rsp            0x7fffffffde30   0x7fffffffde30
r8             0x0      0
r9             0x0      0
r10            0x0      0
r11            0x0      0
r12            0x0      0
r13            0x0      0
r14            0x0      0
r15            0x0      0
rip            0x400078 0x400078 <_start>
eflags         0x202    [ IF ]
cs             0x33     51
ss             0x2b     43
ds             0x0      0
es             0x0      0
fs             0x0      0
gs             0x0      0
(gdb) x /4xg 0x7fffffffde30
0x7fffffffde30: 0x0000000000000004      0x00007fffffffe1f8
0x7fffffffde40: 0x00007fffffffe211      0x00007fffffffe217
(gdb) x /5xg 0x7fffffffde30
0x7fffffffde30: 0x0000000000000004      0x00007fffffffe1f8
0x7fffffffde40: 0x00007fffffffe211      0x00007fffffffe217
0x7fffffffde50: 0x00007fffffffe21c
(gdb) x /s 0x00007fffffffe1f8
0x7fffffffe1f8: "/home/zhang/argument.exe"
(gdb) x /s 0x00007fffffffe211
0x7fffffffe211: "12345"
(gdb) x /s 0x00007fffffffe217
0x7fffffffe217: "6789"
(gdb) x /s 0x00007fffffffe21c
0x7fffffffe21c: "101"
```



```
argument.exe:     file format elf64-x86-64


Disassembly of section .text:

0000000000400078 <_start>:
  400078:       5e                      pop    %rsi

0000000000400079 <vnext>:
  400079:       5e                      pop    %rsi
  40007a:       48 85 f6                test   %rsi,%rsi
  40007d:       74 28                   je     4000a7 <exit>
  40007f:       48 89 f3                mov    %rsi,%rbx
  400082:       48 31 d2                xor    %rdx,%rdx

0000000000400085 <strlen>:
  400085:       8a 03                   mov    (%rbx),%al
  400087:       48 ff c2                inc    %rdx
  40008a:       48 ff c3                inc    %rbx
  40008d:       84 c0                   test   %al,%al
  40008f:       75 f4                   jne    400085 <strlen>
  400091:       c6 43 ff 0a             movb   $0xa,-0x1(%rbx)
  400095:       48 c7 c0 01 00 00 00    mov    $0x1,%rax
  40009c:       48 c7 c7 01 00 00 00    mov    $0x1,%rdi
  4000a3:       0f 05                   syscall
  4000a5:       eb d2                   jmp    400079 <vnext>

00000000004000a7 <exit>:
  4000a7:       48 c7 c0 3c 00 00 00    mov    $0x3c,%rax
  4000ae:       48 c7 c7 00 00 00 00    mov    $0x0,%rdi
  4000b5:       0f 05                   syscall
zhang@ubuntu:~$
```

汇编调用lib_c库函数示例

```
.section        .rodata             #不声明亦可
.LC0:
        .string "Hello world\n”

.text
.globl  _start

_start:
        movl    $.LC0, %edi
        call    puts
        movl    $0, %edi
        call    exit
```

```
#汇编命令
$ as -o hello.o  hello.s
$ ld -lc -dynamic-linker /lib64/ld-linux-x86-64.so.2 -o hello hello.o
```

Linux汇编小结

- **程序结构**
  - 主要包括三个常用的段：

.data	数据段	声明带有初始值的数据
.bss	数据段	声明无需初始化的数据
.text	正文段	程序指令

  - 程序入口地址

    汇编器使用_start符号表示默认的起始点, 此外如果想要汇编内部的符号能够被外部模块访问,需要赋予.globl 属性, 如:

```
.globl          _start
```

数据段 .data

- 声明一个数据元素时, 需要使用Symbol和类型说明
- 示例如下——

```
output:
        .ascii "hello world."
pi:
        .float 3.14

#声明可以在一行中定义多个值, 如:
ages:
        .int 20, 10, 30, 40
```

只读数据段 .section .rodata

类型说明:

.ascii 文本字符串
.asciz 以空字符结尾的字符串
.byte 字节值
.double 双精度浮点值
.float 单精度浮点值
.int 32位整数
.long 32位整数, 和int相同
.octa 16字节整数
.quad 8字节整数
.short 16位整数
.single 单精度浮点数(和float相同)

bss段

- **和data段不同，无需声明特定的数据类型，只需声明为所需目的保留的原始内存部分即可。**

  .comm 声明为未初始化的全局内存区域
  .lcomm 声明为未初始化的局部内存区域

  示例如下——

```
        .section .bss
               .lcomm buffer, 1000
        #该语句把1000字节的内存地址赋予buffer，外部模块不能访问他们
```

- **相比较.data段，.bss段声明的优点是?**

多选题 0.5分

设置

A/B/C三条语句哪个（些）是错的?

A B C

```
        movq $strNum, %rbx
        movq $0, %rdi
popLoop:
        popq %rax
        addb  $48, %al     # 字符0的ASCII码是48
        movb %al, (%rbx,%rdi,1)
        incq  %rdi
        loop popLoop

        movb $NULL, (%rbx,%rdi,1)

        movq   %rdi, %rdx
        movq   %rbx, %rsi
        movq   $1, %rax       # 系统调用号(sys_write)
        movq   $1, %rdi       # 文件描述符(stdout)
        syscall

exit:
        ...
```

```
.equ NULL,  0
.section  .data
intNum:
          .int   1498
.section .bss
          .lcomm strNum, 10
.section .text
.globl _start
_start:
           movl   $intNum, %eax #  A
           movq  0,   %rcx         #  B
           movl   $10, %ebx
divideLoop:
           movl   $0, %edx
           divl     %ebx
           pushq %rdx  #push remainder
           incq     %rcx
           cmpl   %eax, 0          # C
           jne      divideLoop
```

提交

补充算术操作指令(32位指令)

指令	效果	描述
imull S	R[%edx]:R[%eax] = S * R[%eax]	有符号乘(结果64位)
mull S	R[%edx]:R[%eax] = S * R[%eax]	无符号乘(结果64位)
cltd	sign-extend %eax → %edx:%eax	转换为8字节 （指令也可写作CDQ） 还有类似指令cqto

idivl S	R[%edx] = R[%edx]:R[%eax] % S;
	R[%eax] = R[%edx]:R[%eax] / S;
		有符号除法，保存余数和商

divl S	R[%edx] = R[%edx]:R[%eax] % S;
	R[%eax] = R[%edx]:R[%eax] / S;
		无符号除法，保存余数和商

- **.equ 用于把常量值设置为可以在程序中使用的Symbol**
  - .equ factor, 3
- **经过设置之后，数据符号值是不能在程序中改动的**

loop指令步骤：

(1) %rcx = %rcx-1

(2) 判断rcx中的值，不为0则转至标号处执行程序

目录

1. Hello World
2. 第一个汇编程序
3. **汇编示例程序**

.type function_name, @function

***This tells the linker that the symbol *function_name* should be treated as a function.**

递归调用示例

- 阶乘(factorial.s)

```
.section .text
.globl factorial          #this is unneeded unless we want to share it

.globl _start
_start:
    movl $4, %edi         #The factorial takes one argument –
                          #the number we want a factorial of.
    call   factorial      #run the factorial function
    movl %eax, %edi       #factorial returns the answer in %eax, but we
                          #want it in %edi to send it as our exit status

    movq $60, %rax         #exit code
    syscall
```

```
#This is the actual function definition：  factorial  (n)
.type   factorial, @function

factorial:
      movl $1, %eax
      cmpl $1, %edi          #If the number is 1, that is our base case, and
                             #we simply return
      je end_factorial
      pushq %rdi
      decl    %edi           #otherwise, decrease the value
      call     factorial     #call itself
      popq  %rdi
      imull  %edi, %eax      #multiply that by the result of the last call to
                             #factorial; the answer is stored in %eax.
end_factorial:
      ret
```

imull S, D
这里有两个操作数，它将计算S和D的乘积并截断为双字，然后存储在D中；无带符号数与无符号数的区分

C语言调用汇编

```
#include<stdio.h>
extern void stats(int[], int, int *, int *);

int main()
{
    int lst[] = {1, -2, 3, -4, 5, 7, 9, 11};
    int len = 8;
    int sum, ave;

    stats(lst, len, &sum, &ave);
    printf ("Stats:\n");
    printf (" Sum = %d \n", sum);
    printf (" Ave = %d \n", ave);
    return 0;
}
```

?

```
# Function to find the integer sum and integer average for a passed list of signed integers.
# Call:
#       stats(lst, len, &sum, &ave);
# Arguments Passed:
#       1) rdi - address of array
#       2) rsi - length of passed array
#       3) rdx - address of variable for sum
#       4) rcx - address of variable for average
# Returns:
#
.section .text
.globl stats
stats:
        pushq   %r12        # callee saved
        movq    $0, %r11    # index
        movl    $0, %r12d   # sum
```

```
sumLoop:
        movl  (%rdi,%r11,4), %eax  #get lst[i]
        addl    %eax, %r12d             #update sum
        incq    %r11                    #index++
        cmpq  %rsi, %r11
        jb       sumLoop

        movl   %r12d, (%rdx)          #return sum

        movl  %r12d, %eax
        cltd                            #sign-extend %eax —> %edx:%eax
        idivl    %esi
        movl   %eax, (%rcx)            #return average

        #Done, return …
        popq   %r12
        ret
```

文件处理示例

```
# Example program to demonstrate file I/O. This example
# will open/create a file, write some information to the
# file, and close the file. Note, the file name and
# write message are hard-coded for the example.
.section .data
.equ LF, 10 #line feed
.equ NULL,0 #end of string
.equ TRUE,1
.equ FALSE,0
.equ EXIT_SUCCESS,0 #success code
.equ STDIN, 0  #standard input
.equ STDOUT,1 #standard output
.equ STDERR,2 #standard error
.equ SYS_read,0 #read
.equ SYS_write,1 #write
.equ SYS_open, 2 #file open
.equ SYS_close,3 #file close
.equ SYS_fork,57 #fork
```

```
.equ SYS_exit,  60 #terminate
.equ SYS_creat, 85 #file open/create
.equ SYS_time,  201 #get time
.equ O_CREAT,   0x40
.equ O_TRUNC,   0x200
.equ O_APPEND,  0x400
.equ O_RDONLY,  000000 #read only
.equ O_WRONLY,  000001 #write only
.equ O_RDWR,    000002 #read and write
.equ S_IRUSR,   0x100
.equ S_IWUSR,   0x80
.equ S_IXUSR,    0x40
```

```
newline:
    .int LF, NULL
header:
    .ascii  “\nFile Write Example.\n\n\0”
filename:
    .ascii  “url.txt\0”
url:
    .ascii  “http://www.google.com\n\0”

len = . -url - 1

writeDone:
    .ascii  “Write Completed.\n\0”
fileDescrip:
    .quad   0
errMsgOpen:
    .ascii  “Error opening file.\n\0”
errMsgWrite:
    .ascii  “Error writing to file.\n\0”
```

```
.section .text
.globl _start
_start:
    movq $header,%rdi
    call printString
openInputFile:
     movq $SYS_creat, %rax
     movq $filename, %rdi
     movq $S_IRUSR|S_IWUSR, %rsi
     syscall

     cmp $0, %rax
     jl  errorOnOpen

     movq %rax, fileDescrip
```

```
.globl printString
printString:
    pushq %rbx
    movq %rdi, %rbx
    movq $0, %rdx

strCountLoop:
    cmpb $NULL,(%rbx)
    je strCountDone
    incq %rdx
    incq %rbx
    jmp strCountLoop

strCountDone:
    cmpq $0, %rdx
    je prtDone

    movq $SYS_write, %rax
    movq %rdi,%rsi
    movq $STDOUT,%rdi
    syscall

prtDone:
    popq %rbx
    ret
```

```
movq $SYS_write, %rax
movq fileDescriptor, %rdi
movq $url, %rsi
movq $len, %rdx
syscall
cmpq $0, %rax
jl errorOnWrite

movq $writeDone, %rdi
call printString

movq $SYS_close, %rax
movq fileDescriptor, %rdi
syscall
jmp exampleDone
```

```
errorOnOpen:
      movq $errMsgOpen,  %rdi
      call printString
      jmp  exampleDone

errorOnWrite:
      movq $errMsgWrite, %rdi
      call printString
      jmp  exampleDone

exampleDone:
      movq $SYS_exit, %rax
      movq $EXIT_SUCCESS, %rdi
      syscall
```

填空题 0.5分

设置

```
#define M 13
#define N ?
int mat1[M][N];
int mat2[N][M];

int copy_element(long i, long j)
{
    mat1[i][j] = mat2[j][i];
}
```

```
copy_element:
        leaq    (%rsi,%rsi,2), %rax
        leaq    (%rsi,%rax,4), %rax
        addq     %rdi, %rax
        movl     mat2(,%rax,4), %eax
        leaq    (%rdi,%rdi,4), %rdx
        leaq    (%rdi,%rdx,2), %rdx
        addq     %rdx, %rsi
        movl     %eax, mat1(,%rsi,4)
        ret
```

N的数值是 [填空1]

作答