Gruppe 5 - Übung 5

Von: Ivo Janowitz, Nguyen Anh Quang, Tillman Rossa, Roman Seiler, Alexander Uhl

1. Timespow

Aufgabe 1

- Schreiben Sie eine Lösung für den kurzen Test timespow.s aus der letzten Veranstaltung. Der Code befindet sich im Git Repository sysprog

```
# Systemnahe Programmierung - Testen Sie Ihre Kenntnisse!
# H. Hoegl, 2012-11-08

#   timespow(3, 2) + timespow(2, 3)

    .section .data
    .section .text
    .globl _start

_start:
    pushl $2        # b
    pushl $3        # x
    call  timespow2
    addl  $8, %esp
    pushl %eax
    pushl $3        # b
    pushl $2        # x
    call  timespow2
    addl  $8, %esp
    popl  %ebx
    addl %eax, %ebx
    movl $1, %eax
    int $0x80


# timespow2(x, b)
#   return x * 2^b
#   Trick: x * 2^b = shift argument x left by b bits
#                    shll %cl, %ebx  (shift ebx left by cl bits)
.type timespow2, @function
timespow2:
    _____________________ # 1 Prolog
    _____________________ # 2 Prolog
    _____________________ # 3 Argument holen
    _____________________ # 4 Argument holen
    _____________________ # 5 Schieben
    _____________________ # 6 Ergebnis ablegen
    _____________________ # 7 Epilog
    _____________________ # 8 Epilog
    _____________________ # 9 Zurueckkehren
```

2. Selbst Übung

Aufgabe 2

- Vollziehen Sie das im Kapitel 5 (Bartlett) beschriebene Programm mit dem Debugger gdb nach

3. ToUpper in anderen Sprachen

Aufgabe 3

- Formulieren Sie das Programm aus Kapitel 5 in den Sprachen

ToUpper in C

```
C Code to be added
```

ToUpper in Java

```
Java Code to be addedd
```

ToUpper in Python

```
Python Code to be added
```

4. Aufgaben von Kapitel 5

Know the concepts

- Describe the lifecycle of a file descriptor.

  Todo

- What are the standard file descriptors and what are they used for?

  Todo

- What is a buffer?

  Todo

- What is the difference between the .data section and the .bss section?

  Todo

- What are the system calls related to reading and writing files?

  Todo

Use the concepts

- Modify the toupper program so that it reads from STDIN and writes to STDOUT instead of using the files on the command-line.

```
# -*- indent-tabs-mode: nil -*- (for Emacs)

# PURPOSE: This program converts an input file
# to an output file with all letters
# converted to uppercase.
```

```
#
# PROCESSING:
# 1) Open the input file
# 2) Open the output file
# 4) While we're not at the end of the input file
#    a) read part of file into our memory buffer
#    b) go through each byte of memory
#       if the byte is a lower-case letter,
#       convert it to uppercase
#    c) write the memory buffer to output file


        .section .data

        #######CONSTANTS########
        # system call numbers
        .equ SYS_OPEN, 5
        .equ SYS_WRITE, 4
        .equ SYS_READ, 3
        .equ SYS_CLOSE, 6
        .equ SYS_EXIT, 1

        # options for open (look at
        # /usr/include/asm/fcntl.h for
        # various values. You can combine them
        # by adding them or ORing them)
        # This is discussed at greater length
        # in "Counting Like a Computer"
        .equ O_RDONLY, 0
        .equ O_CREAT_WRONLY_TRUNC, 03101

        # standard file descriptors
        .equ STDIN, 0
        .equ STDOUT, 1
        .equ STDERR, 2

        # system call interrupt
        .equ LINUX_SYSCALL, 0x80
        .equ END_OF_FILE, 0 #This is the return value

        # of read which means we've
        # hit the end of the file
        .equ NUMBER_ARGUMENTS, 2

        .section .bss

        # Buffer - this is where the data is loaded into
        # from the data file and written from
        # into the output file. This should
        # never exceed 16,000 for various
        # reasons.
        .equ BUFFER_SIZE, 500
        .lcomm BUFFER_DATA, BUFFER_SIZE

        .section .text

        # STACK POSITIONS
        .equ ST_SIZE_RESERVE, 8
        .equ ST_FD_IN, -4
        .equ ST_FD_OUT, -8
        .equ ST_ARGC, 0         # Number of arguments
        .equ ST_ARGV_0, 4       # Name of program
        .equ ST_ARGV_1, 8       # Input file name
        .equ ST_ARGV_2, 12      # Output file name
        .globl _start

_start:
        ### INITIALIZE PROGRAM ###
        # save the stack pointer

        movl %esp, %ebp

        # Allocate space for our file descriptors
        # on the stack
        subl $ST_SIZE_RESERVE, %esp

open_files:
open_fd_in:
```

```
                        ### OPEN INPUT FILE ###
                        # open syscall
        movl $SYS_OPEN, %eax
                        # input filename into %ebx
        movl ST_ARGV_1(%ebp), %ebx
                        # read-only flag
        movl $O_RDONLY, %ecx
                        # this doesn't really matter for reading
        movl $0666, %edx
                        # call Linux
        int $LINUX_SYSCALL
store_fd_in:
                        # save the given file descriptor
        movl %eax, ST_FD_IN(%ebp)

open_fd_out:
                                ### OPEN OUTPUT FILE ###
                                # open the file
        movl $SYS_OPEN, %eax
                                # output filename into %ebx
        movl ST_ARGV_2(%ebp), %ebx
                                # flags for writing to the file
        movl $O_CREAT_WRONLY_TRUNC, %ecx
                                # mode for new file (if it's created)
        movl $0666, %edx
                                # call Linux
        int $LINUX_SYSCALL

store_fd_out:
                                # store the file descriptor here
        movl %eax, ST_FD_OUT(%ebp)
                                # BEGIN MAIN LOOP
read_loop_begin:
                                   # READ IN A BLOCK FROM THE INPUT FILE
        movl $SYS_READ, %eax
        movl ST_FD_IN(%ebp), %ebx  # get the input file descriptor
        movl $BUFFER_DATA, %ecx    # the location to read into
        movl $BUFFER_SIZE, %edx    # the size of the buffer
        int $LINUX_SYSCALL         # Size of buffer read is returned in %eax
        cmpl $END_OF_FILE, %eax    # check for end of file marker
                                   # if found or on error, go to the end
        jle end_loop
continue_read_loop:
        ### CONVERT THE BLOCK TO UPPER CASE ###
        pushl $BUFFER_DATA      # location of buffer
        pushl %eax              # size of the buffer
        call convert_to_upper
        popl %eax               # get the size back
        addl $4, %esp           # restore %esp
        ### WRITE THE BLOCK OUT TO THE OUTPUT FILE ###
        movl %eax, %edx         # size of the buffer
        movl $SYS_WRITE, %eax
        movl ST_FD_OUT(%ebp), %ebx  # file to use
        movl $BUFFER_DATA, %ecx     # location of the buffer
        int $LINUX_SYSCALL
        jmp read_loop_begin

end_loop:
                                ###CLOSE THE FILES###
                                # NOTE - we don't need to do error checking
                                # on these, because error conditions
                                # don't signify anything special here
        movl $SYS_CLOSE, %eax
        movl ST_FD_OUT(%ebp), %ebx
        int $LINUX_SYSCALL
        movl $SYS_CLOSE, %eax
        movl ST_FD_IN(%ebp), %ebx
        int $LINUX_SYSCALL
                                ### EXIT ###
        movl $SYS_EXIT, %eax
        movl $0, %ebx
        int $LINUX_SYSCALL


#PURPOSE: This function actually does the
# conversion to upper case for a block
#
#INPUT: The first parameter is the location
```

```
# of the block of memory to convert
# The second parameter is the length of
# that buffer
#
#OUTPUT: This function overwrites the current
# buffer with the upper-casified version.
#
#VARIABLES:
# %eax - beginning of buffer
# %ebx - length of buffer
# %edi - current buffer offset
# %cl - current byte being examined
# (first part of %ecx)

        ### CONSTANTS ##
                                # The lower boundary of our search
        .equ LOWERCASE_A, 'a'
                                # The upper boundary of our search
        .equ LOWERCASE_Z, 'z'
                                # Conversion between upper and lower case
        .equ UPPER_CONVERSION, 'A' - 'a'
                                ### STACK STUFF ###
        .equ ST_BUFFER_LEN, 8   # Length of buffer
        .equ ST_BUFFER, 12      # actual buffer
convert_to_upper:
        pushl %ebp
        movl %esp, %ebp
                        ### SET UP VARIABLES ###
        movl ST_BUFFER(%ebp), %eax
        movl ST_BUFFER_LEN(%ebp), %ebx
        movl $0, %edi

                        # if a buffer with zero length was given
                        # to us, just leave
        cmpl $0, %ebx
        je end_convert_loop
convert_loop:
                        # get the current byte
        movb (%eax,%edi,1), %cl
                        # go to the next byte unless it is between
                        #'a' and 'z'
        cmpb $LOWERCASE_A, %cl
        jl next_byte
        cmpb $LOWERCASE_Z, %cl
        jg next_byte
                        # otherwise convert the byte to uppercase
        addb $UPPER_CONVERSION, %cl
                        # and store it back
        movb %cl, (%eax,%edi,1)

next_byte:
        incl %edi #next byte
        cmpl %edi, %ebx #continue unless
                        # we've reached the end
        jne convert_loop
end_convert_loop:
                        # no return value, just leave
        movl %ebp, %esp
        popl %ebp
        ret

# vim: expandtab sw=8 sts=8
```

ToUpper Buffer Version

- Change the size of the buffer.

```
# -*- indent-tabs-mode: nil -*- (for Emacs)

# PURPOSE: This program converts an input file
# to an output file with all letters
# converted to uppercase.
#
# PROCESSING:
```

```
# 1) Open the input file
# 2) Open the output file
# 4) While we're not at the end of the input file
#    a) read part of file into our memory buffer
#    b) go through each byte of memory
#       if the byte is a lower-case letter,
#       convert it to uppercase
#    c) write the memory buffer to output file


        .section .data

        #######CONSTANTS########
        # system call numbers
        .equ SYS_OPEN, 5
        .equ SYS_WRITE, 4
        .equ SYS_READ, 3
        .equ SYS_CLOSE, 6
        .equ SYS_EXIT, 1

        # options for open (look at
        # /usr/include/asm/fcntl.h for
        # various values. You can combine them
        # by adding them or ORing them)
        # This is discussed at greater length
        # in "Counting Like a Computer"
        .equ O_RDONLY, 0
        .equ O_CREAT_WRONLY_TRUNC, 03101

        # standard file descriptors
        .equ STDIN, 0
        .equ STDOUT, 1
        .equ STDERR, 2

        # system call interrupt
        .equ LINUX_SYSCALL, 0x80
        .equ END_OF_FILE, 0 #This is the return value

        # of read which means we've
        # hit the end of the file
        .equ NUMBER_ARGUMENTS, 2

        .section .bss

        # Buffer - this is where the data is loaded into
        # from the data file and written from
        # into the output file. This should
        # never exceed 16,000 for various
        # reasons.
        .equ BUFFER_SIZE, 500
        .lcomm BUFFER_DATA, BUFFER_SIZE

        .section .text

        # STACK POSITIONS
        .equ ST_SIZE_RESERVE, 8
        .equ ST_FD_IN, -4
        .equ ST_FD_OUT, -8
        .equ ST_ARGC, 0         # Number of arguments
        .equ ST_ARGV_0, 4       # Name of program
        .equ ST_ARGV_1, 8       # Input file name
        .equ ST_ARGV_2, 12      # Output file name
        .globl _start

_start:
        ### INITIALIZE PROGRAM ###
        # save the stack pointer

        movl %esp, %ebp

        # Allocate space for our file descriptors
        # on the stack
        subl $ST_SIZE_RESERVE, %esp

open_files:
open_fd_in:
                        ### OPEN INPUT FILE ###
                        # open syscall
```

```
        movl $SYS_OPEN, %eax
                        # input filename into %ebx
        movl ST_ARGV_1(%ebp), %ebx
                        # read-only flag
        movl $O_RDONLY, %ecx
                        # this doesn't really matter for reading
        movl $0666, %edx
                        # call Linux
        int $LINUX_SYSCALL
store_fd_in:
                        # save the given file descriptor
        movl %eax, ST_FD_IN(%ebp)

open_fd_out:
                                ### OPEN OUTPUT FILE ###
                                # open the file
        movl $SYS_OPEN, %eax
                                # output filename into %ebx
        movl ST_ARGV_2(%ebp), %ebx
                                # flags for writing to the file
        movl $O_CREAT_WRONLY_TRUNC, %ecx
                                # mode for new file (if it's created)
        movl $0666, %edx
                                # call Linux
        int $LINUX_SYSCALL

store_fd_out:
                                # store the file descriptor here
        movl %eax, ST_FD_OUT(%ebp)
                                # BEGIN MAIN LOOP
read_loop_begin:
                                   # READ IN A BLOCK FROM THE INPUT FILE
        movl $SYS_READ, %eax
        movl ST_FD_IN(%ebp), %ebx  # get the input file descriptor
        movl $BUFFER_DATA, %ecx    # the location to read into
        movl $BUFFER_SIZE, %edx    # the size of the buffer
        int $LINUX_SYSCALL         # Size of buffer read is returned in %eax
        cmpl $END_OF_FILE, %eax    # check for end of file marker
                                   # if found or on error, go to the end
        jle end_loop
continue_read_loop:
        ### CONVERT THE BLOCK TO UPPER CASE ###
        pushl $BUFFER_DATA      # location of buffer
        pushl %eax              # size of the buffer
        call convert_to_upper
        popl %eax               # get the size back
        addl $4, %esp           # restore %esp
        ### WRITE THE BLOCK OUT TO THE OUTPUT FILE ###
        movl %eax, %edx         # size of the buffer
        movl $SYS_WRITE, %eax
        movl ST_FD_OUT(%ebp), %ebx  # file to use
        movl $BUFFER_DATA, %ecx     # location of the buffer
        int $LINUX_SYSCALL
        jmp read_loop_begin

end_loop:
                                ###CLOSE THE FILES###
                                # NOTE - we don't need to do error checking
                                # on these, because error conditions
                                # don't signify anything special here
        movl $SYS_CLOSE, %eax
        movl ST_FD_OUT(%ebp), %ebx
        int $LINUX_SYSCALL
        movl $SYS_CLOSE, %eax
        movl ST_FD_IN(%ebp), %ebx
        int $LINUX_SYSCALL
                                ### EXIT ###
        movl $SYS_EXIT, %eax
        movl $0, %ebx
        int $LINUX_SYSCALL


#PURPOSE: This function actually does the
# conversion to upper case for a block
#
#INPUT: The first parameter is the location

# of the block of memory to convert
```

```
# The second parameter is the length of
# that buffer
#
#OUTPUT: This function overwrites the current
# buffer with the upper-casified version.
#
#VARIABLES:
# %eax - beginning of buffer
# %ebx - length of buffer
# %edi - current buffer offset
# %cl - current byte being examined
# (first part of %ecx)

        ### CONSTANTS ##
                                # The lower boundary of our search
        .equ LOWERCASE_A, 'a'
                                # The upper boundary of our search
        .equ LOWERCASE_Z, 'z'
                                # Conversion between upper and lower case
        .equ UPPER_CONVERSION, 'A' - 'a'
                                ### STACK STUFF ###
        .equ ST_BUFFER_LEN, 8   # Length of buffer
        .equ ST_BUFFER, 12      # actual buffer
convert_to_upper:
        pushl %ebp
        movl %esp, %ebp
                        ### SET UP VARIABLES ###
        movl ST_BUFFER(%ebp), %eax
        movl ST_BUFFER_LEN(%ebp), %ebx
        movl $0, %edi

                        # if a buffer with zero length was given
                        # to us, just leave
        cmpl $0, %ebx
        je end_convert_loop
convert_loop:
                        # get the current byte
        movb (%eax,%edi,1), %cl
                        # go to the next byte unless it is between
                        #'a' and 'z'
        cmpb $LOWERCASE_A, %cl
        jl next_byte
        cmpb $LOWERCASE_Z, %cl
        jg next_byte
                        # otherwise convert the byte to uppercase
        addb $UPPER_CONVERSION, %cl
                        # and store it back
        movb %cl, (%eax,%edi,1)

next_byte:
        incl %edi #next byte
        cmpl %edi, %ebx #continue unless
                        # we've reached the end
        jne convert_loop
end_convert_loop:
                        # no return value, just leave
        movl %ebp, %esp
        popl %ebp
        ret

# vim: expandtab sw=8 sts=8
```

ToUpper BSS Version

- Rewrite the program so that it uses storage in the .bss section rather than the stack to store the file descriptors.

```
# -*- indent-tabs-mode: nil -*- (for Emacs)

# PURPOSE: This program converts an input file
# to an output file with all letters
# converted to uppercase.
#
# PROCESSING:
```

```
# 1) Open the input file
# 2) Open the output file
# 4) While we're not at the end of the input file
#    a) read part of file into our memory buffer
#    b) go through each byte of memory
#       if the byte is a lower-case letter,
#       convert it to uppercase
#    c) write the memory buffer to output file


        .section .data

        #######CONSTANTS########
        # system call numbers
        .equ SYS_OPEN, 5
        .equ SYS_WRITE, 4
        .equ SYS_READ, 3
        .equ SYS_CLOSE, 6
        .equ SYS_EXIT, 1

        # options for open (look at
        # /usr/include/asm/fcntl.h for
        # various values. You can combine them
        # by adding them or ORing them)
        # This is discussed at greater length
        # in "Counting Like a Computer"
        .equ O_RDONLY, 0
        .equ O_CREAT_WRONLY_TRUNC, 03101

        # standard file descriptors
        .equ STDIN, 0
        .equ STDOUT, 1
        .equ STDERR, 2

        # system call interrupt
        .equ LINUX_SYSCALL, 0x80
        .equ END_OF_FILE, 0 #This is the return value

        # of read which means we've
        # hit the end of the file
        .equ NUMBER_ARGUMENTS, 2

        .section .bss

        # Buffer - this is where the data is loaded into
        # from the data file and written from
        # into the output file. This should
        # never exceed 16,000 for various
        # reasons.
        .equ BUFFER_SIZE, 500
        .lcomm BUFFER_DATA, BUFFER_SIZE

        .section .text

        # STACK POSITIONS
        .equ ST_SIZE_RESERVE, 8
        .equ ST_FD_IN, -4
        .equ ST_FD_OUT, -8
        .equ ST_ARGC, 0         # Number of arguments
        .equ ST_ARGV_0, 4       # Name of program
        .equ ST_ARGV_1, 8       # Input file name
        .equ ST_ARGV_2, 12      # Output file name
        .globl _start

_start:
        ### INITIALIZE PROGRAM ###
        # save the stack pointer

        movl %esp, %ebp

        # Allocate space for our file descriptors
        # on the stack
        subl $ST_SIZE_RESERVE, %esp

open_files:
open_fd_in:
                        ### OPEN INPUT FILE ###
                        # open syscall
```

```
        movl $SYS_OPEN, %eax
                        # input filename into %ebx
        movl ST_ARGV_1(%ebp), %ebx
                        # read-only flag
        movl $O_RDONLY, %ecx
                        # this doesn't really matter for reading
        movl $0666, %edx
                        # call Linux
        int $LINUX_SYSCALL
store_fd_in:
                        # save the given file descriptor
        movl %eax, ST_FD_IN(%ebp)

open_fd_out:
                                ### OPEN OUTPUT FILE ###
                                # open the file
        movl $SYS_OPEN, %eax
                                # output filename into %ebx
        movl ST_ARGV_2(%ebp), %ebx
                                # flags for writing to the file
        movl $O_CREAT_WRONLY_TRUNC, %ecx
                                # mode for new file (if it's created)
        movl $0666, %edx
                                # call Linux
        int $LINUX_SYSCALL

store_fd_out:
                                # store the file descriptor here
        movl %eax, ST_FD_OUT(%ebp)
                                # BEGIN MAIN LOOP
read_loop_begin:
                                   # READ IN A BLOCK FROM THE INPUT FILE
        movl $SYS_READ, %eax
        movl ST_FD_IN(%ebp), %ebx  # get the input file descriptor
        movl $BUFFER_DATA, %ecx    # the location to read into
        movl $BUFFER_SIZE, %edx    # the size of the buffer
        int $LINUX_SYSCALL         # Size of buffer read is returned in %eax
        cmpl $END_OF_FILE, %eax    # check for end of file marker
                                   # if found or on error, go to the end
        jle end_loop
continue_read_loop:
        ### CONVERT THE BLOCK TO UPPER CASE ###
        pushl $BUFFER_DATA      # location of buffer
        pushl %eax              # size of the buffer
        call convert_to_upper
        popl %eax               # get the size back
        addl $4, %esp           # restore %esp
        ### WRITE THE BLOCK OUT TO THE OUTPUT FILE ###
        movl %eax, %edx         # size of the buffer
        movl $SYS_WRITE, %eax
        movl ST_FD_OUT(%ebp), %ebx  # file to use
        movl $BUFFER_DATA, %ecx     # location of the buffer
        int $LINUX_SYSCALL
        jmp read_loop_begin

end_loop:
                                ###CLOSE THE FILES###
                                # NOTE - we don't need to do error checking
                                # on these, because error conditions
                                # don't signify anything special here
        movl $SYS_CLOSE, %eax
        movl ST_FD_OUT(%ebp), %ebx
        int $LINUX_SYSCALL
        movl $SYS_CLOSE, %eax
        movl ST_FD_IN(%ebp), %ebx
        int $LINUX_SYSCALL
                                ### EXIT ###
        movl $SYS_EXIT, %eax
        movl $0, %ebx
        int $LINUX_SYSCALL


#PURPOSE: This function actually does the
# conversion to upper case for a block
#
#INPUT: The first parameter is the location

# of the block of memory to convert
```

```
# The second parameter is the length of
# that buffer
#
#OUTPUT: This function overwrites the current
# buffer with the upper-casified version.
#
#VARIABLES:
# %eax - beginning of buffer
# %ebx - length of buffer
# %edi - current buffer offset
# %cl - current byte being examined
# (first part of %ecx)

        ### CONSTANTS ##
                                # The lower boundary of our search
        .equ LOWERCASE_A, 'a'
                                # The upper boundary of our search
        .equ LOWERCASE_Z, 'z'
                                # Conversion between upper and lower case
        .equ UPPER_CONVERSION, 'A' - 'a'
                                ### STACK STUFF ###
        .equ ST_BUFFER_LEN, 8   # Length of buffer
        .equ ST_BUFFER, 12      # actual buffer
convert_to_upper:
        pushl %ebp
        movl %esp, %ebp
                        ### SET UP VARIABLES ###
        movl ST_BUFFER(%ebp), %eax
        movl ST_BUFFER_LEN(%ebp), %ebx
        movl $0, %edi

                        # if a buffer with zero length was given
                        # to us, just leave
        cmpl $0, %ebx
        je end_convert_loop
convert_loop:
                        # get the current byte
        movb (%eax,%edi,1), %cl
                        # go to the next byte unless it is between
                        #'a' and 'z'
        cmpb $LOWERCASE_A, %cl
        jl next_byte
        cmpb $LOWERCASE_Z, %cl
        jg next_byte
                        # otherwise convert the byte to uppercase
        addb $UPPER_CONVERSION, %cl
                        # and store it back
        movb %cl, (%eax,%edi,1)

next_byte:
        incl %edi #next byte
        cmpl %edi, %ebx #continue unless
                        # we've reached the end
        jne convert_loop
end_convert_loop:
                        # no return value, just leave
        movl %ebp, %esp
        popl %ebp
        ret

# vim: expandtab sw=8 sts=8
```

HeyNow

- Write a program that will create a file called heynow.txt and write the words “Hey diddle diddle!” into it.

```
ToBeDone
```