Files

Systems Programming

JOHANNES KEPLER
UNIVERSITÄT LINZ

Michael Sonntag
Institute of Networks and Security

INSTITUTE
OF NETWORKS
AND SECURITY

The File concept of Unix

- Files are accessed as a sequential stream of bytes
- Opening a file returns a file descriptor (which is a number)
  - Might also be a pointer – but for the user it is **completely opaque**!
  - **MUST** be retained – access to a file is **ONLY** possible through this!
- File permissions
  - Three modes: read, write, execute (rwx)
    - (110) means permission to read and write but not to execute
  - Three different sets
    - user: every file has an owner
    - group: every file belongs to a single group (of arbitrary users)
    - others: users that are neither owner nor in the group
  - Octal encoding
    - Convert binary permission sets to octal numbers
    - Combine them to one number and prefix with 0
  - 0754 = owner can read, write & execute the file,
    group can read and execute,
    everyone else can only read

user	group	other
rwx	r-x	r--
111	101	100
7	5	4

Dealing with Files

- Lifecycle of all files:
  - Open file **& check for success**
  - Read from / write to file **& check for success every time**
  - Close file
    - & check for success → but this is rare and there is little you can do
- All files are closed when the program terminates
  - Unfortunately, there is little guarantee how → data loss possible!
- Opening a file is not possible yourself → ask the OS to do it for you
  - For this you need the file name (including the path - or it will use the current directory of the program)
    - This course: statically defined (data section) or program parameter
- Reading and writing is not possible directly, i.e. from a register
  - You can only read to / write from memory
  - So we need a buffer there

Opening a file

- RAX: 2 (=sys_open system call)
- RDI: Address of filename (must be nul-terminated, i.e. a C string)
- RSI: Flags (read, write, read&write, append…)
  - Must contain one of O_RDONLY, O_WRONLY, or O_RDWR
  - 0..N creation flags O_CREAT, O_TMPFILE, O_TRUNC...
  - 0..N status flags: O_APPEND, O_ASYNC...
- RDX: Mode = permissions for file (when creating one)
  - Use 0777 at most (we use 0666)
    - Linux: also 04000=SUID, 02000=SGID, 01000=Sticky are possible
- Return value RAX: File descriptor
  - >=0: Success – file descriptor
  - <0: Error – negative of error number
    - Example: -13 = Error number 13 = EACCESS = Access not allowed
- Remember: this is a syscall, so RCX and R11 are overwritten

See also http://man7.org/linux/man-pages/man2/open.2.html

Reading from a file

- RAX: 0 (=sys_read)
- RDI: File descriptor
- RSI: Address of buffer to be filled with file data
  - Must contain space for RDX bytes!
  - This is a binary buffer, so there is **no** termination (nul or other)
- RDX: Number of bytes to read at most
  - The OS will **always try** to give you that much data, but there is no guarantee: file is not long enough, network problem…
- Return value RAX: Number of characters **actually read**
  - >0: Success – RAX bytes placed in buffer
  - =0: Success – End of file reached (& no data read)
    - No data available, but not EOF → call blocks (or returns error number EAGAIN; see option O_NONBLOCK)!
  - <0: Error – Negative of error number

Writing to a file

- RAX: 1 (=sys_write)
- RDI: File descriptor
- RSI: Address of buffer with data to be written to file
  - Must contain at least RDX bytes!
- RDX: Number of bytes to write
  - The OS will **always try** to write the full amount, but there is no guarantee: disk full, network problem…
- Return value RAX: Number of characters **actually written**
  - >=0: Success – RAX bytes written to file
    - But not necessarily yet on disk – might be in OS buffer only!
    - Might also block if O_NONBLOCK is not set
  - <0: Error – Negative of error number

Closing a file

- RAX: 3 (=sys_close)
- RDI: File descriptor
- Return value RAX
  - =0: Success
  - <0: Error – Negative of error number

- Writing to a file on a network filesystem might report writing errors only on closing the file (but not on the individual write, as storing the data in the local buffer succeeds!)
- Attention: closing a file is no guarantee that the data is on the disk
  - Use fsync before (RAX 75, RDI file handle), but this may block
    - Guarantees that all file data was sent to the device. This still is no guarantee it is permanently stored (internal buffers)!
    - Also: no guarantees about the file entry (=directory content)

System calls for file manipulation

System call	RAX (cmd.)	RDI (parameter 1)	RSI (parameter 2)	RDX (parameter 3)	RAX (return value)
SYS_OPEN	2	Pointer to filename	Flags (O_RDONLY, …)	Create mode (e.g. 0666)	File descriptor or error number
SYS_READ	0	File descriptor	Pointer to data buffer	Max. number of bytes to read	Actual number of bytes read or error number
SYS_WRITE	1	File descriptor	Pointer to data buffer	Number of bytes to write	Actual number of bytes written or error number
SYS_CLOSE	3	File descriptor	---	---	0 (success) or error number

Buffers – Space for data

- Buffers must be reserved “somehow”:
  - Static: define in assembler file
  - Stack: reduce RSP
    - Not recommended except for very small buffers
  - Heap: explicit memory reservation (see later)
    - Recommended for large buffers
- Static buffers: declare in section BSS
  - BSS: On many (=not all!) systems initialized to all zeros

```
.section .bss
.lcomm my_buffer, 500      # Create a symbol for the start address
                           # Note: No „$“ for the length!

        ……

# RDI already contains the file descriptor
movq $0, %rax              # Read from file into buffer
movq $my_buffer, %rsi      # Store start address of buffer
movq $500, %rdx            # Store length of buffer
syscall                    # Read from file; result in RAX
```

Example: toupper.s – Helper constants

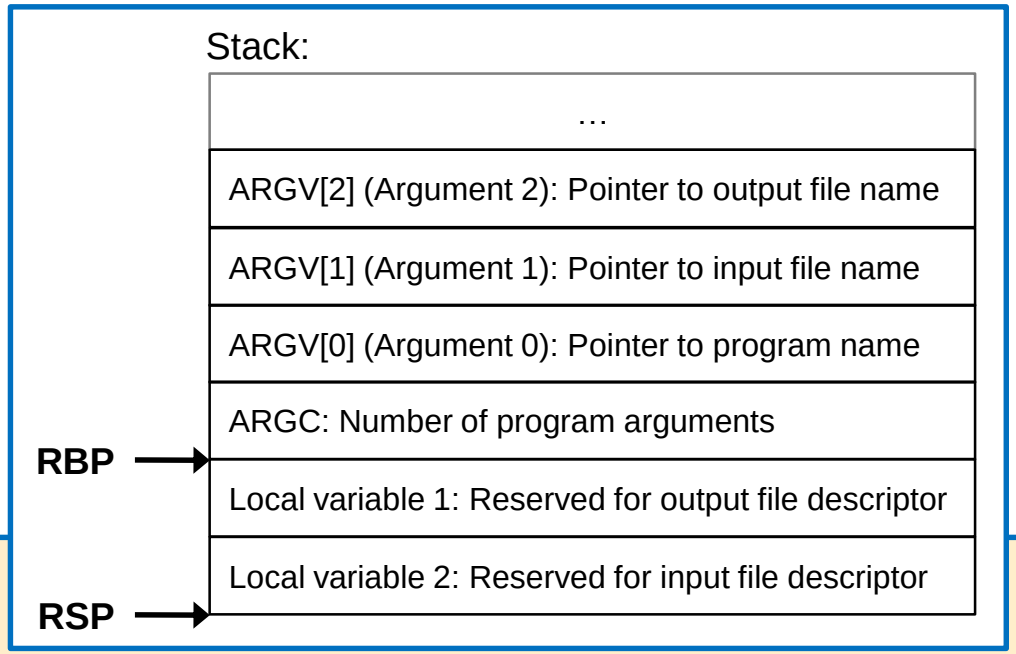
```
# System call numbers
.equ SYS_OPEN, 2
.equ SYS_READ, 0
.equ SYS_WRITE, 1
.equ SYS_CLOSE, 3
.equ SYS_EXIT, 60

.equ O_RDONLY, 0                      # Open file options - read-only
.equ O_CREAT_WRONLY_TRUNC, 03101      # Open file options - these are:
                                      # CREAT  - create file if not exising
                                      # WRONLY - only write to this file
                                      # TRUNC  - destroy current contents

.equ O_PERMS, 0666                    # Read & Write perms. for everyone

# End-of-file result status
.equ END_OF_FILE, 0     # This is the return value of read()
                        # which means we've hit the end of
                        # the file
```

Example: toupper.s – Data buffer

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

Example: toupper.s – Helper constants

Remember: OS puts command line arguments (actually only pointers to those strings) on stack




```
    .section .text

    # STACK POSITIONS
    .equ ST_SIZE_RESERVE, 16 # Space for local variables
    # Note: Offsets are RBP-based, which is set immediately at program start
    .equ ST_FD_IN, -16       # Local variable for input file descriptor
    .equ ST_FD_OUT, -8       # Local variable for output file descriptor
    .equ ST_ARGC,    0       # Number of arguments (integer)
    .equ ST_ARGV_0,  8       # Name of program (address = pointer to string)
    .equ ST_ARGV_1, 16       # Input file name (address = pointer to string)
    .equ ST_ARGV_2, 24       # Output file name (address = pointer to string)
```

Example: toupper.s (4)

```
_start:
    ###INITIALIZE PROGRAM###
    movq %rsp, %rbp              # Create new stack frame
    subq $ST_SIZE_RESERVE, %rsp # Allocate file descriptor space on stack

    ###CHECK PARAMETER COUNT###
    cmpq $3, ST_ARGC(%rbp)
    je open_files

    movq $-1, %rdi               # Our return value for parameter problems
    movq $SYS_EXIT, %rax
    syscall
```

Example: toupper.s (5)

```
open_files:
open_fd_in:
    ###OPEN INPUT FILE###
    movq ST_ARGV_1(%rbp), %rdi   # Input filename into %rdi
    movq $O_RDONLY, %rsi         # Read-only flag
    movq $O_PERMS, %rdx          # This doesn't really matter for reading
    movq $SYS_OPEN, %rax         # Specify "open"
    syscall                      # Call Linux

    cmpq $0, %rax                # Check success
    jl exit                      # In case of error simply terminate
store_fd_in:
    movq  %rax, ST_FD_IN(%rbp)   # Save the returned file descriptor
```

Example: toupper.s (6)

```
open_fd_out:
    ###OPEN OUTPUT FILE###
    movq ST_ARGV_2(%rbp), %rdi          # Output filename into %rdi
    movq $O_CREAT_WRONLY_TRUNC, %rsi    # Flags for writing to the file
    movq $O_PERMS, %rdx                 # Permissions for new file (if created)
    movq $SYS_OPEN, %rax                # Open the file
    syscall                             # Call Linux

    cmpq $0, %rax                       # Check success
    jl close_input                      # In case of error close input file
                                        # (already open!)
store_fd_out:
    movq %rax, ST_FD_OUT(%rbp)          # Store the file descriptor
```

Example: toupper.s (7)

```
read_loop_begin:
    ###READ IN A BLOCK FROM THE INPUT FILE###
    movq ST_FD_IN(%rbp), %rdi      # Get the input file descriptor
    movq $BUFFER_DATA, %rsi        # The location to read into
    movq $BUFFER_SIZE, %rdx        # The size of the buffer
    movq $SYS_READ, %rax
    syscall                        # Size of buffer read is returned in %rax
    ###EXIT IF WE'VE REACHED THE END###
    cmpq $END_OF_FILE, %rax        # Check for end of file marker (or error)
    je end_loop                    # If found, go to the end
    jl close_output                # On error just terminate

continue_read_loop:
    ###CONVERT THE BLOCK TO UPPER CASE###
    movq $BUFFER_DATA, %rdi        # Location of the buffer
    movq %rax, %rsi                # Size of the buffer
    pushq $-1                      # Dummy value for stack alignment
    pushq %rax                     # Store bytes read for write check
    call convert_to_upper
```

Example: toupper.s (8)

```
write_out_begin:
    ###WRITE THE BLOCK OUT TO THE OUTPUT FILE###
    movq ST_FD_OUT(%rbp), %rdi     # File to use
    movq $BUFFER_DATA, %rsi        # Location of  buffer
    movq %rax, %rdx                # Buffer size (=number of bytes read)
    movq $SYS_WRITE, %rax
    syscall                        # Note: Check how much was written!

    ###CHECK WRITE SUCCESS###
    popq %rbx                      # Retrieve number of bytes read
    addq $8, %rsp                  # Remove stack alignment space
    cmpq %rax, %rbx                # Compare number read to written
    jne close_output               # If not the same, terminate program

    ###CONTINUE THE LOOP###
    jmp read_loop_begin
```

Example: toupper.s (9)

```
end_loop:                   # No special error handling, so success and error
close_output:               # are the same: we just close both files
    ###CLOSE THE FILES###
    # NOTE - we don't need to do error checking on these, because
    #        error conditions don't signify anything special here
    movq ST_FD_OUT(%rbp), %rdi
    movq $SYS_CLOSE, %rax
    syscall

close_input:
    movq ST_FD_IN(%rbp), %rdi
    movq $SYS_CLOSE, %rax
    syscall

exit:
    ###EXIT###
    movq $0, %rdi
    movq $SYS_EXIT, %rax
    syscall
```

Example: toupper.s (10)

```
#####FUNCTION convert_to_upper
#PURPOSE:    This function actually does the conversion to upper case for a block
#INPUT:      The first parameter (rdi) is the location of the block of memory to convert
#            The second parameter (rsi) is the length of that buffer
#OUTPUT:     This function overwrites the current buffer with the upper-casified version.
#VARIABLES:
#            %rax  - beginning of buffer
#            %rbx  - length of buffer (old value must be saved!)
#            %rdi  - current buffer offset
#            %r10b - current byte being examined (%r10b is the first byte of %r10)
# Note: This variable assignment is for exemplary purposes only and very suboptimal!

    .equ  LOWERCASE_A, 'a'          # The lower boundary of our search
    .equ  LOWERCASE_Z, 'z'          # The upper boundary of our search
    .equ  UPPER_CONVERSION, 'A' - 'a'
                                    # Conversion: Difference upper/lower case
convert_to_upper:
    pushq %rbp                      # Prepare stack
    movq  %rsp, %rbp
    pushq %rbx                      # Save RBX
    ###SET UP VARIABLES###
    movq  %rdi, %rax
    movq  %rsi, %rbx
    movq  $0, %rdi
```

Example: toupper.s (11)

```
    # If a buffer with zero length was given us, just leave
    cmpq $0, %rbx
    je end_convert_loop
convert_loop:
    movb (%rax,%rdi,1), %r10b     # Get the current byte
    # Go to the next byte unless it is between 'a' and 'z'
    cmpb $LOWERCASE_A, %r10b
    jl next_byte
    cmpb $LOWERCASE_Z, %r10b
    jg next_byte
    # Otherwise convert the byte to uppercase
    addb $UPPER_CONVERSION, %r10b
    movb %r10b, (%rax,%rdi,1)     # And store it back
next_byte:
    incq %rdi                     # Next byte
    cmpq %rdi, %rbx               # Continue unless we've reached the end
    jne convert_loop
end_convert_loop:
    movq %rdi, %rax         # Store number of chars converted as return value
    popq %rbx
    movq %rbp, %rsp
    popq %rbp
    ret
```

Standard file descriptors

- Three file descriptors are already open per default
  - STDIN
    - Represents input read from keyboard
    - End of input → press <CTRL-d>
    - File descriptor **0**
  - STDOUT
    - Represents output written to screen
    - File descriptor **1**
  - STDERR
    - Represents error output written to screen
    - File descriptor **2**

- Do NOT close them; they cannot be reopened!
  - Unless you really know what you (want to) do...
    - E.g. for daemons/services running in the background

Unix file paradigm

- The default behavior of most UNIX programs is to
  - Read input from standard input (STDIN)
  - Write output to standard output (STDOUT)
  - Write error output to standard error (STDERR)

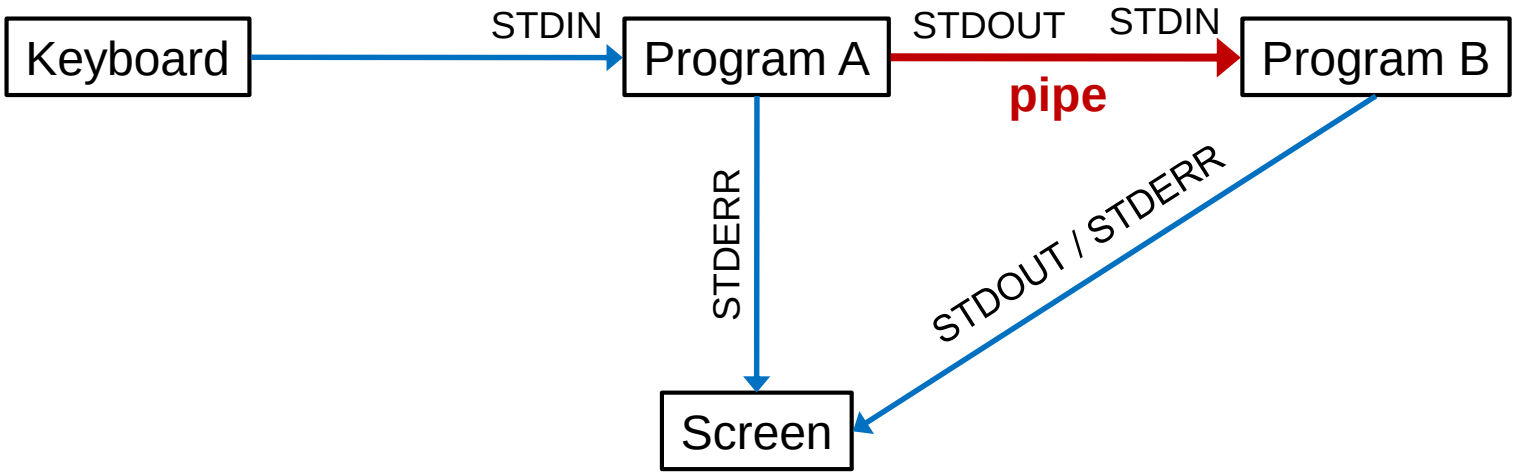
- The paradigm of UNIX is to treat all input/output systems as files
  - Network connections
  - Serial port
  - Audio devices
  - Harddisks
  - etc.

Redirecting Input/Output

- Redirect stdin to file (=read input from file as if it was typed in manually)
  - `sort < list.input`
- Redirect stdout to file (= output is written into file **instead** of printed on screen)
  - `ls > ls.output`
- Redirect stderr to file
  - `ls 2> ls.error`
- Redirect stderr to stdout
  - `ls 2>&1`
- Redirect stdout to stderr
  - `ls 1>&2`
- Redirect stdout and stderr to file
  - `ls &> ls.output`
- Redirect and append stdout to file
  - `ls >> ls.output`
- Redirect and append stderr to file
  - `ls 2>> ls.error`

Pipes

- Pipes
  - They connect programs, similar to a physical pipeline
  - Feed output from program A directly as input to program B
    - Connects STDOUT from first program to STDIN from second program
  - Often | used as pipe symbol
  - **`cat file.txt | sort | uniq`**



Comparing files

- Comparing files (e.g. assignment exemplary output):
  - Command-line only (or you would need to find other tools)
    - Such exist for all OS, but they are mostly for much more complicated tasks!
  - **`cmp -b`** ***`your_file exemplary_file`***
    - Return value: 0 = identical, 1 = different
    - -b also prints the differing bytes
  - **`diff -u`** ***`your_file exemplary_file`***
    - -u also shows the “surrounding” – a few lines before and after
      - Useful for “text” files only

Comparing files - Example

- **diff -u *maximum.s maximum_new.s***

```
--- maximum.s      2017-10-18 15:39:14.000000000 +0200
+++ maximum_new.s  2019-03-26 13:51:15.546264824 +0100
@@ -5,7 +5,7 @@
          #VARIABLES: The registers have the following uses:
          #
          # %rdx - Holds the index of the data item being examined
-         # %rdi - Largest data item found
+         # %rdi - Largest data item found until now
          # %rax - Current data item
          #
          # The following memory locations are used:
@@ -24,11 +24,11 @@
          .globl _start
 _start:
          movq $0, %rdx                # move 0 into the index register
-         movq data_items(,%rdx,8), %rax # load the first byte of data
          movq %rax, %rdi              # since this is the first item, %rax is
                                       # the biggest

 start_loop:                           # start loop
+         movq data_items(,%rdx,8), %rax # load the first byte of data
          cmpq $0, %rax                # check to see if we've hit the end
          je loop_exit
          incq %rdx                    # load next value
```

Changed line

Removed line

Inserted line

JOHANNES KEPLER
UNIVERSITÄT LINZ

INSTITUTE
OF NETWORKS
AND SECURITY

https://www.ins.jku.at

THANK YOU FOR YOUR ATTENTION!

Slides by: Michael Sonntag
michael.sonntag@ins.jku.at
+43 (732) 2468 - 4137
S3 235 (Science park 3, 2nd floor)

JOHANNES KEPLER
UNIVERSITÄT LINZ
Altenberger Straße 69
4040 Linz, Österreich
www.jku.at