Files

Systems Programming

Michael Sonntag
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
- 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
  - Note: 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

- Note: 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!)
- Note: 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
    - Note: Guarantees that all file data was sent to the device. This still is no guarantee it is permanently stored (internal buffers)!
    - Note: 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
```

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 only 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)

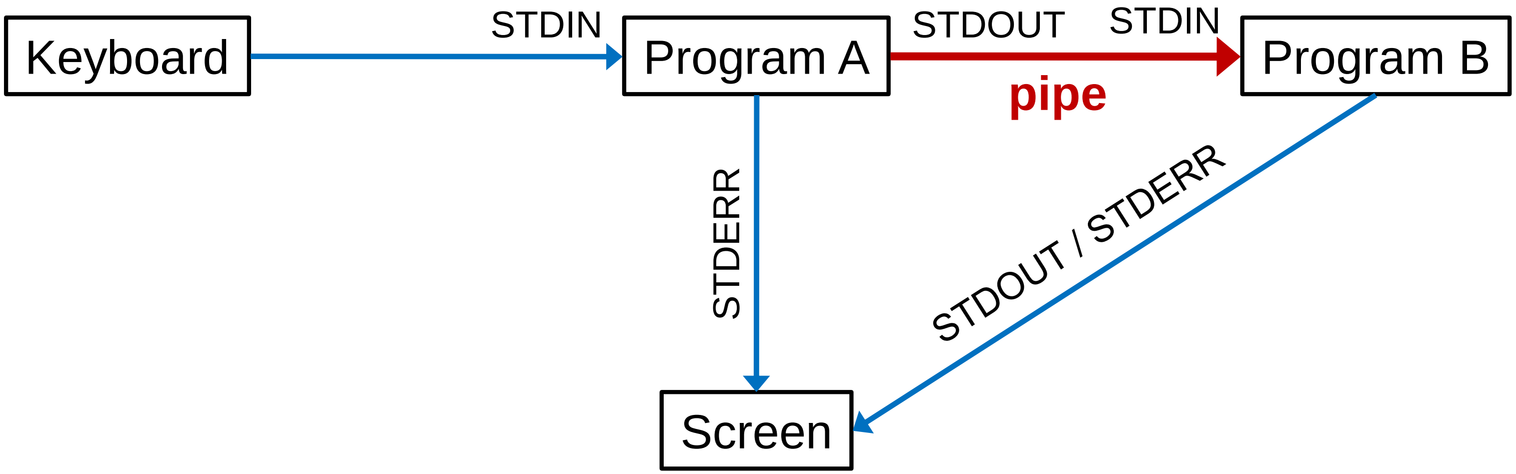
- The paradigm of UNIX is to treat all input/output systems as files
  - Network connections
  - Serial port
  - Audio devices
  - Harddisks
  - etc.

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

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)