System Calls

Ray Seyfarth

August 4, 2011

System calls

- A system call is a special function call which changes the CPU's privilege level to enable more capabilities
- A user process cannot do privileged instructions
  - No in or out instructions
  - No changing of CPU mapping registers
- Instead a user process makes a system call
- The system call is a part of the kernel of the operating system
- It verifies that the user should be allowed to do the requested action and then does the action

Outline

1. 32 bit system calls
2. 64 bit system calls
3. C wrapper functions

32 bit Linux system calls

- Each system call is identified by an integer defined in "/usr/include/asm/unistd_32.h"
- The system call number is placed in eax
- Parameters are placed in registers ebx, ecx, edx, esi, edi, and ebp
- Process uses the software interrupt number 0x80 to make the system call
- Return value in eax

```
        segment .data
hello:  db      "Hello world!",0x0a
        segment .text
        ...
        mov     eax, 4          ; syscall 4 is write
        mov     ebx, 1          ; file descriptor
        lea     ecx, [hello]    ; array to write
        mov     rdx, 13         ; write 13 bytes
        int     0x80
```

64 bit Linux system calls

- System call number defined in "/usr/include/asm/unistd_64.h"
- System call number is placed in rax
- Parameters rdi, rsi, rdx, r10, r8 and r9.
- Process uses syscall instruction
- Return value in rax.

```
        segment .data
hello:  db      "Hello world!",0x0a
        segment .text
        global  _start
_start: mov     eax, 1          ; syscall 1 is write
        mov     edi, 1          ; file descriptor
        lea     rsi, [hello]    ; array to write
        mov     edx, 13         ; write 13 bytes
        syscall
        mov     eax, 60         ; syscall 60 is exit
        xor     edi, edi        ; exit(0)
        syscall
```

C wrapper functions

- Every system call is available through a C “wrapper function”
- A wrapper function might do very little other than shuffle registers
- Some wrappers offer a little extra convenience
- Wrapper functions are described in section 2 of the on-line manual
    - Use “`man 2 write`” to learn about the write system call

```
        segment .data
msg:    db      "Hello World!",0x0a ; String to print
len:    equ     $-msg               ; Length of the string
        segment .text
        global  main
        extern  write, exit
main:
        mov     edx, len            ; Arg 3 is the length
        mov     rsi, msg            ; Arg 2 is the array
        mov     edi, 1              ; Arg 1 is the fd
        call    write
        xor     edi, edi            ; 0 return = success
        call    exit
```

Open system call

```
int open ( char *pathname, int flags [, int mode ] );
```

- `pathname` is a null-terminated string
- `flags` is a collection of options or'ed together
- `mode` is the permissions to grant if a file is created

flags	meaning
0	read-only
1	write-only
2	read and write
0x40	create if needed
0x200	truncate the file
0x400	append

Permissions for files

- There are 3 basic permissions: read, write and execute
- There are 3 categories of users: user (owner), group and other
- Each of the 3 categories gets a 0 or 1 for each basic permission
- Octal works well for permissions
- 640o is an octal number granting read and write permission to the user, read permission to the group and no permission to others

Code to open a file

- Open system call returns a small non-negative integer identifying the opened file
- It returns -1 on error and sets errno

```
        segment .data
fd:     dd      0
name:   db      "sample",0
        segment .text
        extern  open
        lea     rdi, [name] ; pathname
        mov     esi, 42     ; read-write | create
        mov     rdx, 600o   ; read-write for me
        call    open
        test    eax, 0
        jz      error       ; failed to open
        mov     [fd], eax
```

Read and write system calls

```
int read ( int fd, void *data, long count );
int write ( int fd, void *data, long count );
```

- fd is the file descriptor returned by open
- data is a pointer to some memory to send or receive data
- count is the number of bytes to read or write
- The data can be any type
- These functions return the number of bytes read or written
- read returns 0 on end-of-file
- They both return -1 on errors and set errno
- Use perror to print a text description based on errno

Lseek system call

```
long lseek ( int fd, long offset, int whence );
```

- `offset` is a byte offset from whence
- If `whence` is 0, `offset` is the byte position
- If `whence` is 1, `offset` is relative to the current position
- If `whence` is 2, `offset` id relative to the end of the file
- `lseek` returns the current position
- Using `whence` = 2 and `offset` = 0, `lseek` returns the file size

Reading an entire file

```
mov     edi, [fd]
xor     esi, esi    ; set offset to 0
mov     edx, 2      ; set whence to 2
call    lseek       ; determine file size
mov     [size], rax
mov     edi, rax
call    malloc      ; allocate an array for the file
mov     [data], rax
mov     edi, [fd]
xor     esi, esi    ; set offset to 0
xor     edx, edx    ; set whence to 0
call    lseek       ; seek to start of file
mov     edi, [fd]
mov     esi, [data]
mov     edx, [size]
call    read        ; read the entire file
```

The close system call

```
int close ( int fd );
```

- You should make a habit of closing files when no longer needed
- They will be closed when the process ends
- No data is buffered in the user process, so data written to unclosed files will be written
- Closing will reduce overhead in the kernel
- There is a per-process limit on open files
- Use "`ulimit -a`" to see your limits