Today: System and Kernel Calls

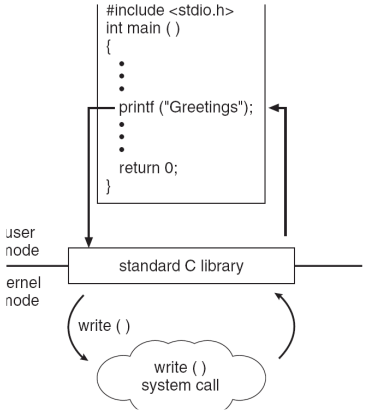
- System calls
- System calls in Minix
- Kernel calls in Minix
- Lab 1: implementing system and kernel calls

System Calls

- Programming interface to the services provided by the OS
- Typically written in a high-level language (C or C++)
- Mostly accessed by programs via a high-level Application Program Interface (API) rather than direct system call use
- Three most common APIs are Win32 API for Windows, POSIX API for POSIX-based systems (including virtually all versions of UNIX, Linux, and Mac OS X), and Java API for the Java virtual machine (JVM)
- Why use APIs rather than system calls?

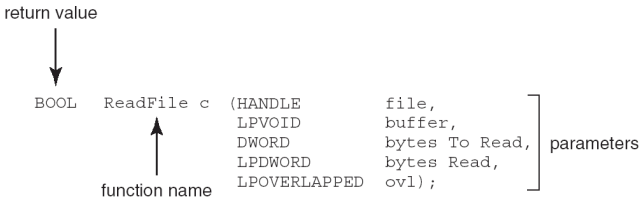
Standard C Library Example

- C program invoking printf() library call, which calls write() system call



Example of Standard API

- Consider the ReadFile() function in the
- Win32 API—a function for reading from a file



- A description of the parameters passed to ReadFile()
  - HANDLE file—the file to be read
  - LPVOID buffer—a buffer where the data will be read into and written from
  - DWORD bytesToRead—the number of bytes to be read into the buffer
  - LPDWORD bytesRead—the number of bytes read during the last read
  - LPOVERLAPPED ovl—indicates if overlapped I/O is being used

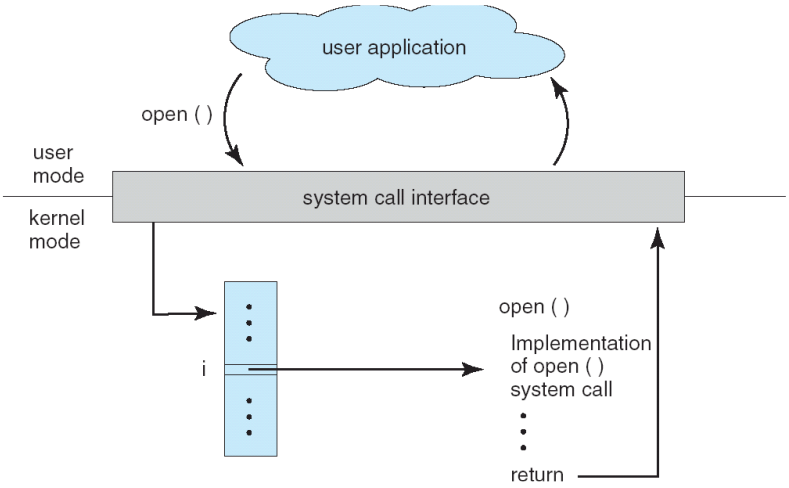
Generic System Call Implementation

- A single hardware operation such as TRAP raises the priority level and begins execution from a table of functions specified at boot time
- TRAP has an integer parameter, and the system call number.
- parameters to the system call are on the stack or in registers
- The kernel source includes a large table of functions, together with a limit specifying the maximum value of the trap parameter
- These functions may be defined anywhere in the kernel

System Call Implementation

- Typically, a number associated with each system call
  - System-call interface maintains a table indexed according to these numbers
- The system call interface invokes intended system call in OS kernel and returns status of the system call and any return values
- The caller need know nothing about how the system call is implemented
  - Just needs to obey API and understand what OS will do as a result call
  - Most details of OS interface hidden from programmer by API
    - Managed by run-time support library (set of functions built into libraries included with compiler)

API – System Call – OS Relationship



System Call Parameter Passing

- Often, more information is required than simply identity of desired system call
  - Exact type and amount of information vary according to OS and call
- Three general methods used to pass parameters to the OS
  - Simplest: pass the parameters in *registers*
    - In some cases, may be more parameters than registers
  - Parameters stored in a *block,* or table, in memory, and address of block passed as a parameter in a register
    - This approach taken by Linux and Solaris
  - Parameters placed, or *pushed,* onto the *stack* by the program and *popped* off the stack by the operating system
  - Block and stack methods do not limit the number or length of parameters being passed

What is POSIX

- Portable Operating System Interface, is a family of standards specified by the IEEE for maintaining compatibility between operating systems
- POSIX defines the application programming interface (API), along with command line shells and utility interfaces, for software compatibility with variants of Unix and other operating systems.

Example of Posix Calls

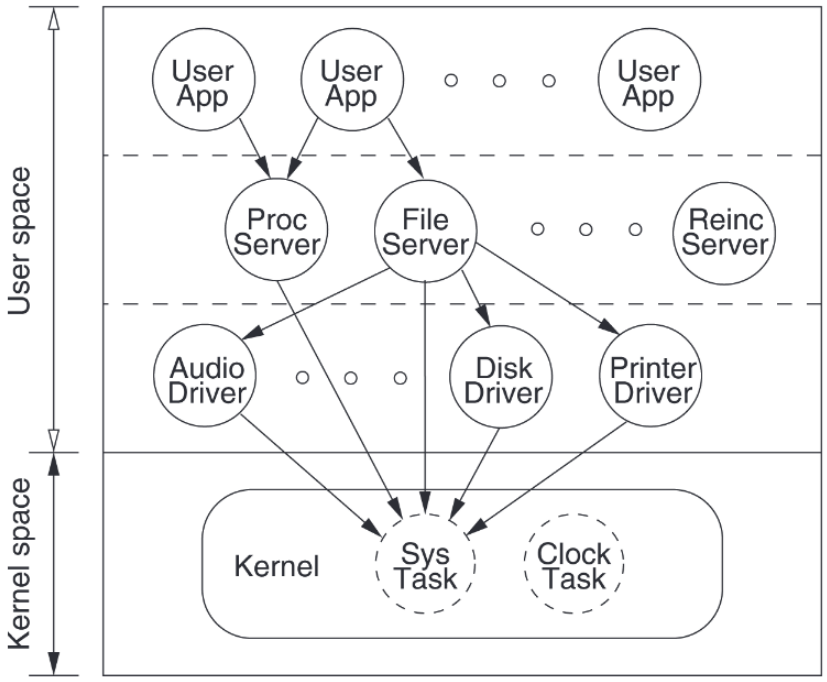
- Calls in POSIX include:
  - networking calls such as `socket`, `connect`, `bind`, `listen`, `accept`, `send`, `recv`, shutdown,
  - calls for mapping files to memory, such as `void * mmap(void *start, size_t length, int prot, int flags, int fd, off_t offset)` and int munmap(void *start, size_t length)
  - posix threads calls such as `pthread_create`–`poll` or `select` to check open file descriptors for I/O availability

Posix calls

- may be implemented as system calls or library functions
- include generic math and string functions
- For a complete list, see the link
  - https://pubs.opengroup.org/onlinepubs/009695399/index.html

Remember:

Minix3 is layered



Key Minix System Calls

- Process management: `fork`, `wait` and `waitpid`, `execve`, `exit`, `brk`, `getpid`.
- Signal handling: `sigaction`, `sigpending`, `kill`, `alarm`, `pause`.
- time: `time`, `stime`, `times`.

Key Minix System Calls

- File management: `open`, `creat`, `mknod`, `close`, `read`, `write`, `seek`, `stat` and `fstat`, `dup2`, `pipe`, `access`, `rename`, `fcntl`, `ioctl`
- Directory and file system management: `mkdir`, `rmdir`, `link`, `unlink` (remove), `mount`, `umount`, `chdir`, `chroot`.
- Protection: `chmod`, `getuid`, `setuid`, `chown`.

Full list of System Calls

minix/include/minix/callnr.h

3 contributors

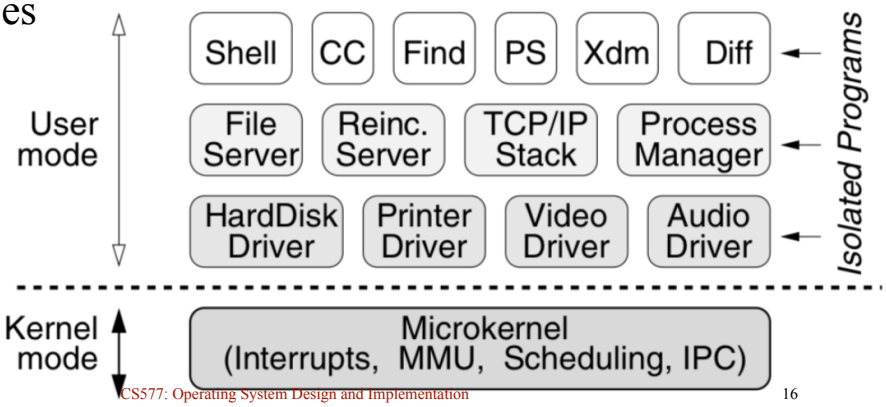
115 lines (108 sloc) 3.1 KB Raw Blame History

```
#define NCALLS          114   /* number of system calls allowed */

/* In case it isn't obvious enough: this list is sorted numerically. */
#define EXIT              1
#define FORK              2
#define READ              3
#define WRITE             4
#define OPEN              5
#define CLOSE             6
#define WAIT              7
#define CREAT             8
#define LINK              9
#define UNLINK           10
#define WAITPID          11
#define CHDIR            12
#define TIME             13
#define MKNOD            14
#define CHMOD            15
#define CHOWN            16
#define BRK              17
#define PREV_STAT        18
#define LSEEK            19
#define MINIX_GETPID     20
#define MOUNT            21
#define UMOUNT           22
#define SETUID           23
#define GETUID           24
#define STIME            25
#define PTRACE           26
#define ALARM            27
#define PREV_FSTAT       28
#define PAUSE            29
#define UTIME            30
```

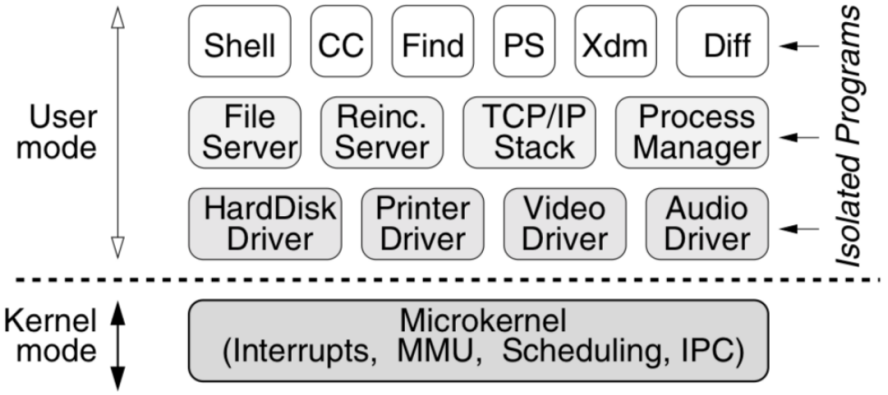

Recap

- Multiserver Operating System
  - Run device drivers outside the kernel
  - Many OS components also run as user-level processes



Recap

- No direct link from User to Kernel except via servers or drivers!

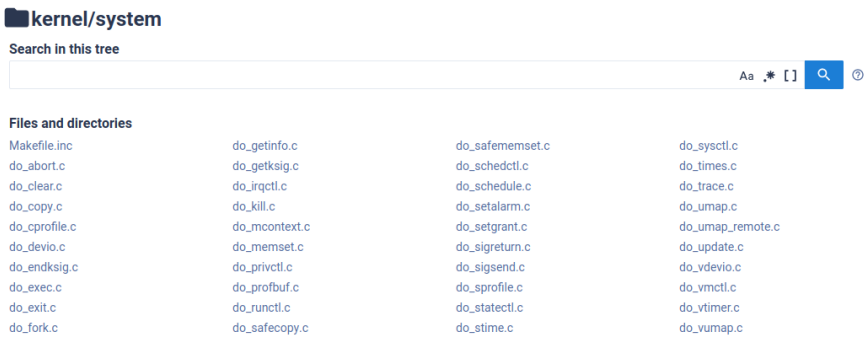


Minix System Call Implementation

- No way directly from user to kernel except via servers!
  - TRAP goes into the kernel as before, but there is effectively only one true system call (more later)
  - The caller's stack/registers have parameters indicating which function to call in which server
  - each server has its own set of system calls
    - the file system server provides system calls for accessing files
    - the process manager manages processes

Minix System Calls

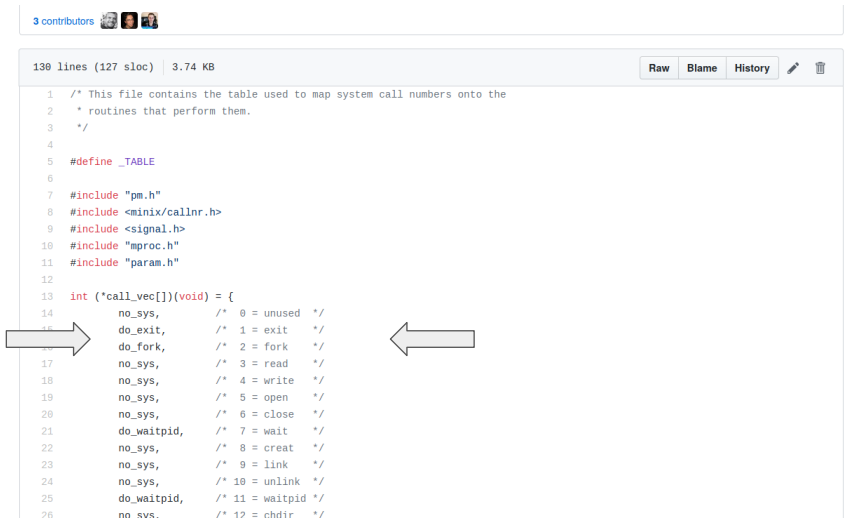
- In Minix some system calls just map the call to another kernel calls. For example fork is created in the process manager and mapped to SYS_FORK
  - System Calls implementation lies in servers
  - Kernel Call implementation lies in kernel/system



A Syscall life in Minix: fork as an example

- User calls fork()
- A TRAP occurs, but that trap goes to the Process Management server

A Syscall life in Minix: fork as an example



3 contributors

130 lines (127 sloc) | 3.74 KB — Raw | Blame | History

```
/* This file contains the table used to map system call numbers onto the
 * routines that perform them.
 */

#define _TABLE

#include "pm.h"
#include <minix/callnr.h>
#include <signal.h>
#include "mproc.h"
#include "param.h"

int (*call_vec[])(void) = {
        no_sys,         /*  0 = unused  */
        do_exit,        /*  1 = exit    */
        do_fork,        /*  2 = fork    */
        no_sys,         /*  3 = read    */
        no_sys,         /*  4 = write   */
        no_sys,         /*  5 = open    */
        no_sys,         /*  6 = close   */
        do_waitpid,     /*  7 = wait    */
        no_sys,         /*  8 = creat   */
        no_sys,         /*  9 = link    */
        no_sys,         /* 10 = unlink  */
        do_waitpid,     /* 11 = waitpid */
        no_sys,         /* 12 = chdir   */
```


But how to get the Syscall number?

minix/sys/sys/syscall.h

- An automatically generated file
  - Remember, Minix is now POSIX compatible!

```
20
21  /* syscall: "fork" ret: "int" args: */
22  #define SYS_fork        2
```

System and Kernel Calls

- Control moved to the server
- One of two scenarios, depending on syscall
  - Server implements the entire syscall, with no further calls to the kernel
  - Server needs to invoke/change/set something in the kernel
- fork() needs to call the kernel!
  - Thus, the long route!

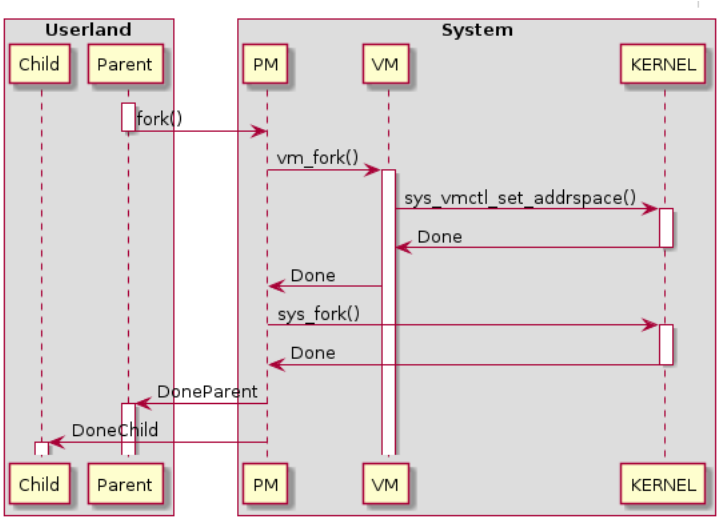
fork() in Minix: do_fork()

- Actual fork implementation in **minix/kernel/system/do_fork.c**
- The actual pipeline
  - User calls *fork(2)* (definition in minix/lib/libc/sys-minix/fork.c)
  - This in turn calls *_syscall()* to call the correct server + the correct function (definition in minix/lib/libc/sys-minix/syscall.c)
  - *_syscall* calls *sendrec* (defined in include/minix/ipc.h)
  - Finally, *sendrec* calls the interrupt vector using *_do_kernel_call_orig* which is a function that is architecture dependent, (for example implementation in lib/libc/arch/i386/sys-minix/_ipc.S)
    - Traps into the kernel
    - passes the message pointer to kernel in the %eax register

Notes

- Most communication between different servers, kernel and servers, and any subsystems is done via the predefined constants in minix/include/minix/com.h
- But a fork does not deal with just the process
  - Memory needs to be managed

A high-level call graph for fork



Kernel Calls

- Some system calls have a corresponding kernel call
  - fork() and sys_fork
- But reverse is not true
- Some kernel call meant for "internal" operations between kernel and system processes
  - sys_devio() kernel call to read or write I/O ports
    - kernel call invoked by a device driver
  - Message/IPC primitives: send, receive, notify can be thought of a type of kernel call

Kernel Calls

- Kernel calls and IPC are restricted to system processes
- System calls can be invoked by user processes
- /usr/system.conf is the config file that describes restrictions

Lab 1

- Goal: How to implement a system call, a kernel call and a system process?
- Handed out in GitHub Classroom
- Turn in via GitHub

- Kernel programming
  - Expect kernel panics!

Lab 1

- Class motto:
  - **"Your kernel may panic, but you shouldn't"**

- Lab 1: A Gentle Introduction to Kernel Programming
  - Part 1: Highly-scripted guided tour
  - Part 2: Use part 1 to complete assignment

Part 1

- "Guided tour" to implement a system call, a kernel call and a server process
  - Be sure to complete all the steps to understand how all of these work
  - Code and instructions are in the "samples" folder
  -

Part 2

- Implement a new Minix server: calc()
  - implements two services: add() and multiply
  - add() is a system call and handled by calc() in user space
  - multiply() is a system call and also a kernel call.
    - user process calls multipy(), which comes to calc()
    - calc() invokes kernel call for multiply() to get result and return results to user process.