

Workshop 4 - System Calls

Team Cardinal
CSCI 3453 Spring 2017

March 31, 2017

1 System Calls

1.1 Overview

System calls are a consequence of an operating system built upon Dual-Mode Operation. The dual-mode paradigm allows the operating system to place instructions within two different execution contexts: "user mode" and "kernel mode". In user mode, an instruction must be checked if the process running it has the appropriate permissions. On the flip side, operations in kernel mode can execute without additional permissions. In other words, kernel mode allows unfettered access to actions which are otherwise restricted. Dual-mode operations force programmers to draw a line between what constitutes a protected kernel operation and what does not. For this to be possible, the operating system provides user processes with system calls to interface with the kernel. More specifically, user-mode processes use system calls to query the kernel to act as an intermediary. To aid in understanding the execution and control flow of system calls for our upcoming second Pintos project, this document provides a technical overview of system call transitions and the corresponding interrupt that enables the aforementioned transitions.

1.2 Mode Transitions

1.2.1 User to Kernel

System Calls: The primary distinction between a system call and the other mode transitions is that a user process makes a voluntary request to switch modes whereas the other mode transitions are involuntary. In many cases, this request is a special instruction known as a **trap instruction** which changes the next instruction of the process to begin at a pre-defined block. Different trap instructions are used for each system call and thus behave similarly to function calls. However, note that the kernel is where the desired actions of the system call are executed. Upon completion, the kernel restores the program counter of the process to the instruction immediately following the trap and then restores any contextual data (such as register values).

Interrupts: Interrupts that come from external sources cause a temporary transfer of control from the user mode to kernel mode. When an interrupt occurs then that event must

be dealt with. The only way for this to occur is for the kernel to regain control of the processor and take action by running the appropriate interrupt handler.

Processor Exceptions: Processor exceptions are generated by hardware and are typically caused by a process trying to perform some type of undefined or restricted instruction such as dividing by zero or accessing memory outside of its bounds. In these cases, the process hands control to the kernel to run an exception handler which may or may not stop execution of the process depending on what the exception is.

1.2.2 Kernel to User

Kernel to User transfer is somewhat more straightforward. The kernel is tasked with starting new processes and after the kernel has completed its job, control is shifted back to user mode so the process can begin. After the kernel is finished handling an interrupt or system call, it restores the program counter of the interrupted process and switches back to user mode so that the process can resume.

1.3 Internal Interrupts

As opposed to external interrupts which are generated by events outside the CPU, internal interrupts are caused directly by CPU instructions. Furthermore, internal interrupts are **synchronous** (i.e. synchronized with CPU instructions) as opposed to external interrupts which are not synchronized with the CPU and can occur at any point during the execution of an instruction. System calls therefore manifest as internal interrupts because there exists a direct link between where the instruction is called in the program and when the interrupt occurs.

1.4 System Calls in Pintos

A major task in Pintos P2-User Programs will be implementing system calls to for process management and file system operations. These include functionality such as starting and exiting user processes, and operations for working with files such as reading/writing, creating and deleting files. A full list of the system calls that must be implemented can be found in section 3.3.4 of the Pintos Manual, or Appendix A of this document.

Some of the system calls directly access the file system. It is important to provide a synchronization mechanism that disallows race conditions on data from the file system. For example, in the write, read, and exec system calls, a global lock should be acquired that maintains single access to shared file system resources. This synchronization will allow any number of user programs to make system calls at once and consequently access the file system across threads. Synchronization of the system calls that access the file system will make the Pintos abstract file system thread-safe and "bullet-proof" to build upon for subsequent projects.

1.4.1 Assembly Instructions

One question that remains is how to invoke the system call after it has been made from the user space. Pintos accomplishes this task in a way consistent with the simulated IA-32 architecture that it runs on. Pintos defines four macros in `/lib/user/syscall.c`, one for 0, 1, 2, or 3 arguments that are passed to that particular system call. Each argument is pushed onto the stack before the actual system call is invoked by the `'int $0x30'` assembly instruction. Both the assembly instruction and the system call number that was passed into the function determine the correct position in an interrupt vector table. Each entry in the vector table points to the code needed to appropriately handle the event associated with the system call number. An example of one system call function is shown below:

```
/* Invokes syscall NUMBER, passing argument ARG0, and returns the
   return value as an 'int'. */
#define syscall1(NUMBER, ARG0) \
    ({ \
        int retval; \
        asm volatile \
            ("pushl_%[arg0];_pushl_%[number];_int_$0x30;_addl_$8,_%esp" \
             : "=a" (retval) \
             : [number] "i" (NUMBER), \
               [arg0] "g" (ARG0) \
             : "memory"); \
        retval; \
    })
```

In this scenario, a user program might make a system call such as `wait(pid)`. Because the `wait` system call takes only one argument, the aforementioned `syscall1` would then be called using the parameters `NUMBER` which is the system call number corresponding to `wait`, and the `pid` which is the process id to wait on. Looking at the assembly code above, we can see that `ARG0` (the system call argument) and then `NUMBER` (the system call number) are pushed onto the stack before the `'int $0x30'` instruction is executed to direct the system to the correct code. The system call numbers have already been defined in Pintos as an enumerated type and a complete list of system call numbers can be found in `src/lib/syscall-nr.h`.

Open Questions

1. Since file system code must be treated as a critical section within system calls (Pintos Manual: pg. 32), how should Pintos' synchronization mechanisms be employed to temporarily inhibit user process concurrency?
2. What synchronization construct should be used to track the state of a child process directly executed by its parent process? In other words, what synchronization constructs are required for the system calls `exec` and `wait`?
- 3.

Conclusion

In order for an operating system to do much of anything useful it must be able to run programs written by users. This cannot be accomplished without having a mechanism in place for user level processes to communicate with the kernel. System calls act as the entry point for this communication and provide a number of benefits. By having a well defined set of system calls, kernel and user level code can be developed independently and still transfer between one another as long as they abide by the criteria set forth by the system call API. Furthermore, system calls act as a line of defense for the kernel. System calls define what actions user programs are allowed to request and they must be implemented in such a way that user programs cannot infiltrate the operation of the kernel inappropriately. This will be a critical bit of functionality in Pintos P2 and foresight must be used so that Pintos is able to handle a user program probing the kernel in any number of fringe cases.

Appendix A Pintos System Calls

```
void halt (void) NORETURN;
    /* Terminates Pintos by calling power_off() */

void exit (int status) NORETURN;
    /* Terminates current user program */

pid_t exec (const char *file);
    /* Runs an executable file pointed at by the function argument */

int wait (pid_t);
    /* Cause the calling process to wait until the process given by pid_t
       dies */

bool create (const char *file, unsigned initial_size);
    /* Creates a new file called file with size initial_size */

bool remove (const char *file);
    /* Deletes file */

int open (const char *file);
    /* Opens file */

int filesize (int fd);
    /* Returns size of file in bytes */

int read (int fd, void *buffer, unsigned length);
    /* Read a specified number of bytes from file fd into buffer. */

int write (int fd, const void *buffer, unsigned length);
    /* Writes specified number of bytes from buffer into file fd */

void seek (int fd, unsigned position);
    /* Changes position for where the next byte is to be read from */

unsigned tell (int fd);
    /* Return the position of next byte to be read or written from file fd
       */

void close (int fd);
    /* Close file pointed at by fd */
```

References

- Pintos Source Code
 - lib/user/syscall.h
 - src/lib/syscall-nr.h
- Operating Systems: Principles and Practices Second Edition by Thomas Anderson and Michael Dahlin
- Pintos Manual by Ben Pfaff
- IA-32 Intel Architecture Software Developer's Manual Volume 1: Basic Architecture