Workshop 4 - System Calls

Team Cardinal
CSCI 3453 Spring 2017

March 31, 2017

1 System Calls

1.1 Overview

In the simplest terms, a system call is merely a way for a program to request that the operating system kernel perform some action. Because the operations governed by the kernel are of a privileged nature, a program cannot always directly perform the necessary actions. When these instances occur, the program must make a query to the kernel to act as an intermediary and act on its behalf.

System calls are a consequence of an operating system that is built on Dual-Mode Operation. Dual-Mode is a way to describe the context in which an instruction is being executed. In user mode, an instruction must be checked if the process running it has the appropriate permissions. On the flip side, in kernel mode an operation is allowed to execute without any check on permissions. In essence, kernel mode allows unfettered access to actions which are otherwise restricted. The major flaw of this design being that Dual-Mode forces us to draw a line between what constitutes a protected kernel operation and what does not. Because of this we must have a way for a user program to interface with the kernel. System calls are one way for a user level program to transfer control to temporarily transfer control to the kernel.

1.2 Mode Transitions

Obviously if we are running a system in dual-mode we need a way to transfer from user to kernel mode and vice-versa from kernel to user. Transfer of control can happen in more than one way and it it vital to know when and why the transitions have occurred.

1.2.1 User to Kernel

System Calls: The primary distinction between a system call and other modes of transfer is that a user processes makes a voluntary request to switch modes while other modes are involuntary. In many cases this call is a special instruction known as a trap instruction which changes the next instruction of the process to begin at a pre-defined block. A different trap instruction is used for each system call and therefore acts similar to a function call with the major distinction being that the kernel is now tasked with executing that function.

Upon completion the kernel restores the program counter of the process to the instruction immediately following the trap and restores any contextual data such as register values etc.

Interrupts: Interrupts that come from external sources cause a temporary transfer of control from the user mode to kernel mode. When a 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 it's bounds. In these cases, the processes 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 to allow the process to 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. In addition this makes internal interrupts synchronous (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 a 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. This assembly instruction plus the system call number passed to the function are used to go to 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. Questions here

Conclusion

Appendix A Pintos System Calls

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

void exit (int status) NO_RETURN;
        /* 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