System Calls in Nachos

Matt Peters
Robert W. Hill

September 1, 1998

Abstract

This document describes the system call interface for the *Nachos instructional Operating System*. Code excerpts are provided, and an example system call is detailed.

1 Introduction

A *Nachos* user program operates on a simulated **MIPS** processor. At the same time, the *Nachos* operating system operates on a host processor (in this case, an **x86**). Since the operating system and the user program are operating on different architectures, the system call interface is slightly different than the normal interface.

2 Overview

This section provides an overview to the flow of control during a system call in the **Nachos** operating system. This flow is shown in Figure 1 below.

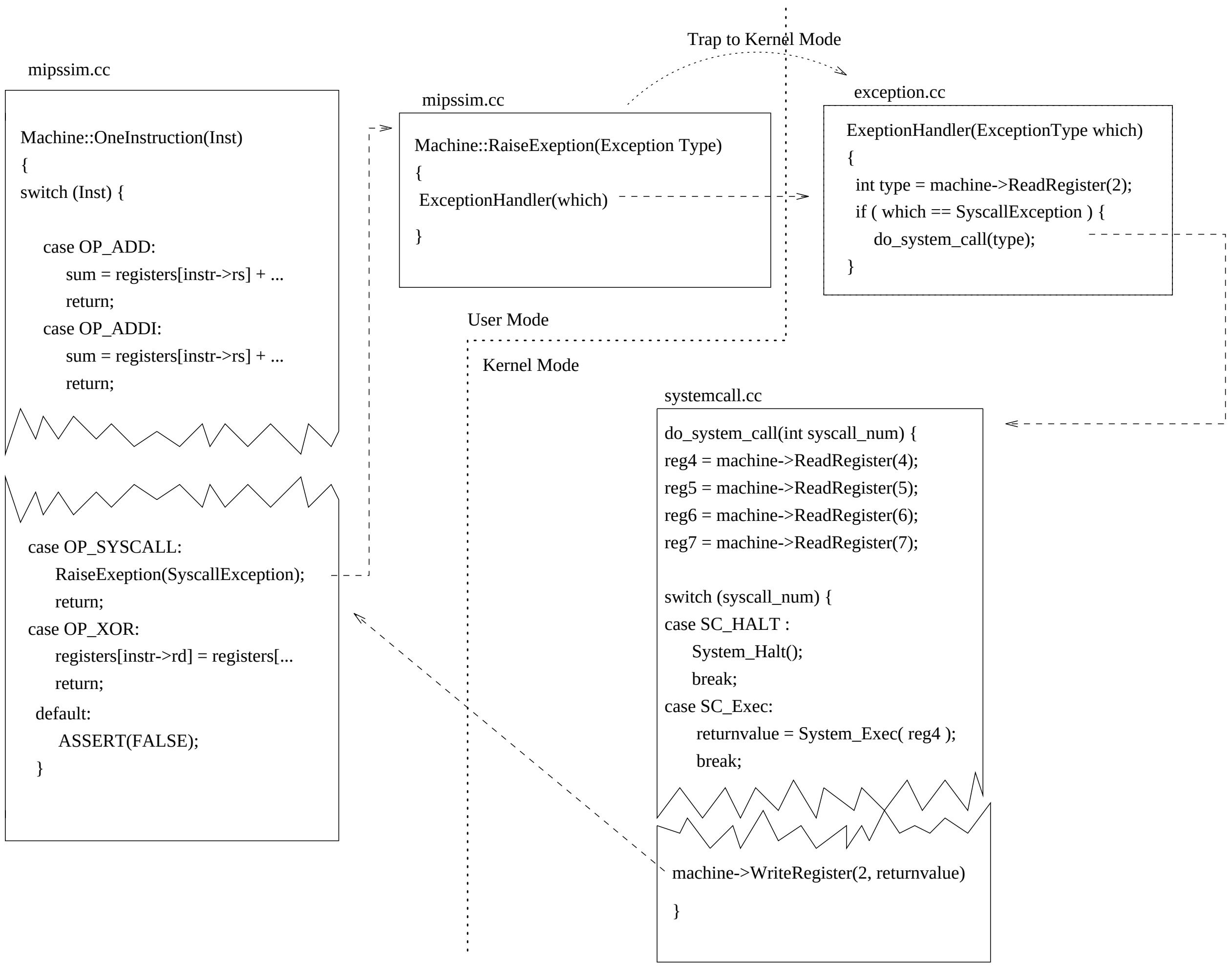
2.1 Arguments

In the simulated **MIPS** machine, arguments to system calls are passed in predefined registers. The user code packs the appropriate arguments into the registers prior to executing the system call. The contents of the user registers prior to the system call are shown in Figure 2.

It should be noted that the system call number is always put into register **2**.

2.2 Trapping

The simulated **MIPS** machine provides a **syscall** instruction. Execution of this instruction results in an unconditional transfer of control to an error handling routine. In the *Nachos* operating system, this routine is **RaiseException**. **RaiseException** disables interrupts



```
Machine::OneInstruction(Inst)
{
switch (Inst) {

    case OP_ADD:
        sum = registers[instr->rs] + ...
        return;
    case OP_ADDI:
        sum = registers[instr->rs] + ...
        return;

  case OP_SYSCALL:
      RaiseExeption(SyscallException);
      return;
  case OP_XOR:
      registers[instr->rd] = registers[...
      return;
   default:
       ASSERT(FALSE);
   }
```

```
Machine::RaiseExeption(Exception Type)
{
 ExceptionHandler(which)
}
```

```
ExeptionHandler(ExceptionType which)
{
  int type = machine->ReadRegister(2);
  if ( which == SyscallException ) {
      do_system_call(type);
}
```

```
do_system_call(int syscall_num) {
reg4 = machine->ReadRegister(4);
reg5 = machine->ReadRegister(5);
reg6 = machine->ReadRegister(6);
reg7 = machine->ReadRegister(7);

switch (syscall_num) {
case SC_HALT :
     System_Halt();
     break;
case SC_Exec:
      returnvalue = System_Exec( reg4 );
      break;

 machine->WriteRegister(2, returnvalue)

  }
```

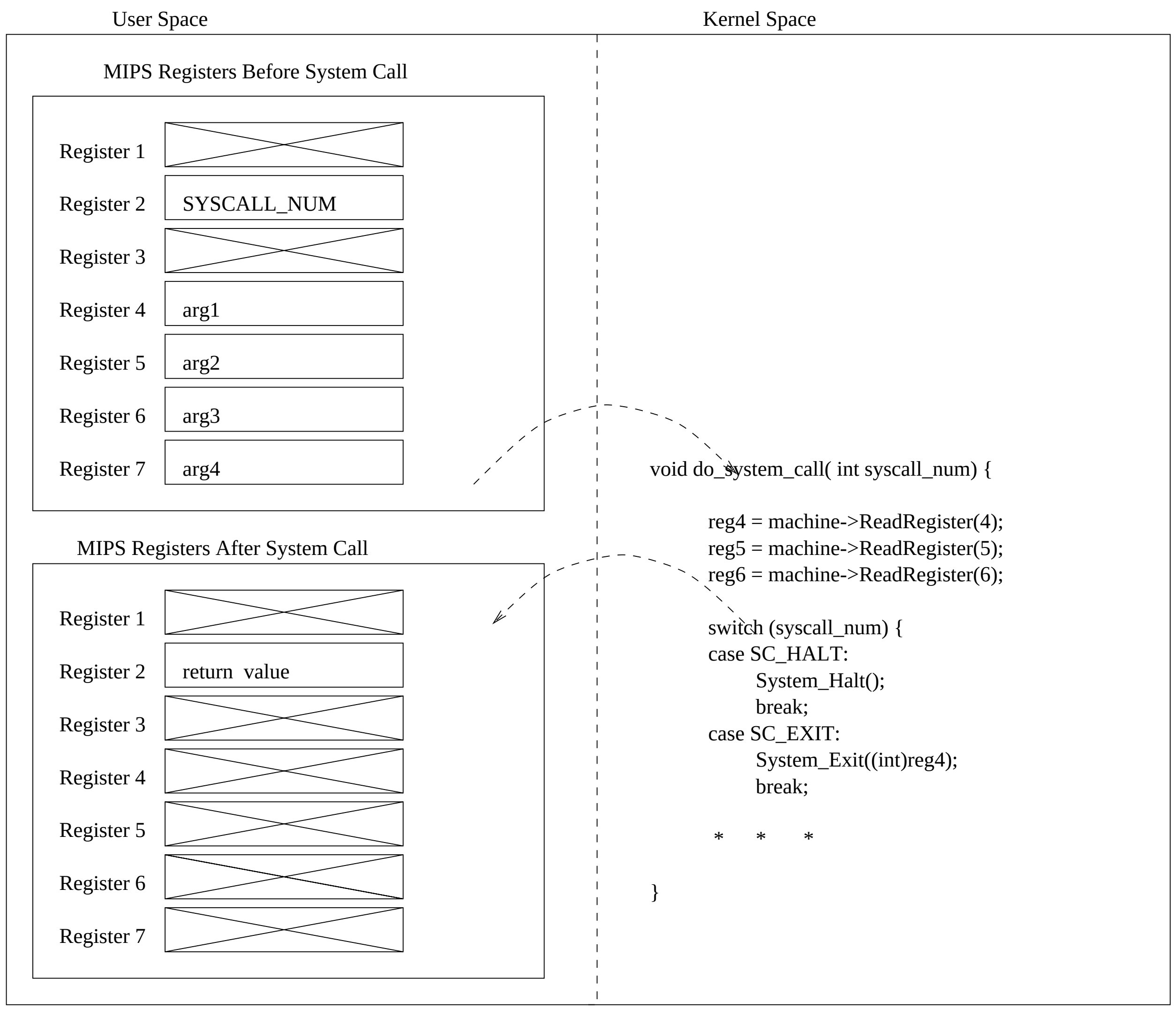
Figure 1: Control flow for system trap in Nachos

and stores the address of the exception in a special register. This value is only used in the case of a page fault. Then, a call is made to the **ExceptionHandler** routine. **ExceptionHandler** checks the type of the exception (page fault, math error, system call, etc) and dispatches the appropriate routine. In the case of a system call, the routine **do_system_call** is invoked.

The operation of **do_system_call** is straightforward. The contents of the user registers **4,5,6**, and **7** are stored in temporary variables, and based on the system call number (as passed in register **2**), the appropriate **System_** *function* is called. For example, if the contents of register **2** is **SC_Exec**, then the **System_Exec** function is called.

2.3 Returning

On return from a system call, the return value of the **System_** function is recorded in register **2**. If this value is negative, then it represents an error. The user stub then saves the negative



```
void do_system_call( int syscall_num) {

        reg4 = machine->ReadRegister(4);
        reg5 = machine->ReadRegister(5);
        reg6 = machine->ReadRegister(6);

        switch (syscall_num) {
        case SC_HALT:
                System_Halt();
                break;
        case SC_EXIT:
                System_Exit((int)reg4);
                break;

         *     *     *

}
```

Figure 2: Registers prior to and after a system call

value in the global variable **errno**, and returns -1 to the user via register **2**.

It should be noted that the **errno** variable provided by **Nachos** follows the **Unix** standard for returning errors from system calls.

3 Putting it all together: The read system call

Given a short program involving the **read** system call, this section will walk through the process of a system trap.

```
//
// Sample User program
```

```
//
int main(void) {
    char buf[80];
    int fd;       //file descriptor

    //
    // open the file, do some stuff, etc...
    //
    if ( read(fd,buf,79) != -1 ) {
       return( 0 );
    } else {
       return( 1 );
    }
}
```

During compilation of the user program, the reference to the **Read** system call is replaced by a jump to the subroutine in the code stub below. This stub represents a transfer of control from the user program to the operating system, as described above.

```
//
// From: start.s Line 144
//
       .globl Read
       .ent Read
Read:
       addiu $2,$0,SC_Read
       syscall
       bgez $2,$ReadPos
       subu $3,$0,$2
       sw $3,errno
       li $2,-1
       j $ReadDone
$ReadPos:
       sw $0, errno
$ReadDone:
       j $31
       .end Read
```

In the code above, the value **SC_Read** (a predefined system call number) is loaded into register **2**. Then the program executes the **syscall** instruction. In the simulated **MIPS** machine, a large switch statement is used to parse instructions from the user program. The **syscall** case is reproduced below.

On return from the system call, the stub checks the value of register **2**. If the value is negative, it stores the absolute value into the global program variable **errno** and stores -1 into register **2** (i.e. it returns -1 to the program). If the value is non-negative, it falls through.

```
//
// From: mipssim.cc Line 546
//
 case OP_SYSCALL:
        RaiseException(SyscallException, 0);
        break;
```

It should be noted that the **syscall** instruction raises a *SyscallException* which will be interpreted by the exception handler, as shown below. Note that the **RaiseException** call exists only to store the address of the exception prior to turning interrupts off. In this way, a page fault can be properly serviced.

```
//
// From: machine.cc Line 100
//
void
Machine::RaiseException(ExceptionType which, int badVAddr)
{
    registers[BadVAddrReg] = badVAddr;
    DelayedLoad(0, 0);                  // finish anything in progress
    interrupt->setStatus(SystemMode);
    ExceptionHandler(which);            // interrupts are enabled at this point
    interrupt->setStatus(UserMode);
}
```

Finally, the **ExceptionHandler** is called (below). This code is responsible for dispatching the appropriate handler for the exception type. In the case of a **read**, the **do_system_call** routine will be called, with the SC_READ value from register **2** as the type argument.

```
//
// From: exception.cc Line 50
//
void
ExceptionHandler(ExceptionType which)
{
    int type = machine->ReadRegister(2);

    if (which == SyscallException) {
        do_system_call(type);
```

The purpose of **do_system_call** is to recover the contents of the user-packed registers and dispatch the appropriate system call based on the **syscall_num**. In the case of **read**, the contents of registers **4,5** and **6** are used to provide the file descriptor, the user space pointer, and the number of bytes to read, respectively.

```
void do_system_call(int syscall_num) {
  int reg4, reg5, reg6, reg7, returnvalue;
  // these are the argument registers used by the system call
  // functions.
  reg4 = machine->ReadRegister(4);
  reg5 = machine->ReadRegister(5);
  reg6 = machine->ReadRegister(6);
  reg7 = machine->ReadRegister(7);

  switch (syscall_num) {
    case SC_Halt:
      System_Halt();
      break;
    case SC_Exit:
      System_Exit((int)reg4);
      break;

       <<removed>>

    case SC_Read:
      returnvalue = System_Read ((int) reg4, (char*) reg5, (int) reg6);
      break;

 }
```

4 Conclusion

This document has presented a brief overview of the **Nachos** system call interface. This interface is based in part on the simulated **MIPS** architecture, and in part on the host **x86** architecture. Readers interested in more information on the **MIPS** architecture should consult *Mips Risc Architecture* by Gerry Kane.