Helium 2 System Calls

ColdFire Architecture

Neil Klingensmith
7-12-2008

Introduction

This document explains the order of operations for system calls in the Helium 2 operating system. System calls are implemented using software interrupts, allowing the operating system execute privileged instructions such as disabling interrupts with a move to the status register. Information exchange between the operating system and the user code is accomplished with two different protocols: one for system calls that may give control of the CPU to a different thread[1], and another for system calls that do not affect which thread has control of the CPU[2].

There are two classes of responsibilities when executing a system call: the responsibilities of the user thread and the responsibilities of the interrupt handler. The tasks to be performed by a user thread are encapsulated in functions defined in the file `oscalls.c`. These functions organize the data passed to them by the user code (they usually put relevant data in the CPU registers), set up the user stack for the OS call, and execute the trap instruction. Trap handlers (the actual operating system code that runs in supervisor mode) are located in the files that correspond to their function; that is, mutex-related system call code is located in `mutex.c`, memory-related system call code is located in `memory.c`, etc.

System Calls that May Transfer CPU Control to a Different Thread

1. The user thread calls a standard C function located in `oscalls.c` with the relevant information for the system call. The module `oscalls.c` must be linked in to the user program binary file in order for the user-space code to be referenced. The “wrapper” functions in `oscalls.c` are written in assembly language, and may need to be rewritten when using different compilers.
2. The wrapper function pushes all registers onto the user stack using the `movem` instruction. This prepares the thread for being rescheduled by `StartExecution`, which automatically pops all registers off of the user stack.
3. Load the system call parameters into the appropriate CPU registers. These will vary depending on the system call.

[1] The OS call `mutexTake`, for example, may give control of the CPU to a different thread if the other thread already controls the mutex of interest.
[2] The OS call `heapAlloc` will never transfer control of the CPU to a different thread. It will only return the address of a memory block.

4. Call the appropriate trap instruction. **This is the last action taken by the user code.** The trap instruction will cause (a) execution to jump to the appropriate exception handler and (b) the processor to enter supervisor mode. The return address and status register for the user thread will be pushed onto the supervisor stack.
5. The trap handler will disable interrupts using the instruction

```
move.w #0x2700,SR
```

6. The trap handler will save the user stack pointer, status register, and return program counter in the thread's control block using the code

```
move.l usp,a4
movea.l currThread,a2
move.l a4,24(a2)       // Save USP in the currThread object
move.l 4(a6),20(a2)    // Save the SR into currThread->SR
move.l 8(a6),16(a2)    // save the PC into currThread->PC[3]
```

   Note that the source operand effective address for the last two instructions is a location on the supervisor stack relative to the frame pointer A6. These instructions assume that a stack frame has been allocated using the link instruction (done automatically by the CodeWarrior compiler).

7. The trap exception handler will perform the function requested by the user thread. Many of the trap handlers are written purely in assembly language and just call a separate function written in C that actually performs the action requested by the user thread.
8. After the requested action is performed, the trap handler will collapse its stack frame (this must be done explicitly when using the CodeWarrior compiler) and jump to the `StartExecution` routine.
9. StartExecution will determine the next highest priority thread ready to run, place its SR and PC on the supervisor stack, pop its registers off of its user stack, set up its SP and execute an RTE instruction.

System Calls that Will Not Transfer CPU Control to a Different Thread

1. The user thread calls a standard C function located in `oscalls.c` with the relevant information for the system call. The module `oscalls.c` must be linked in to the user program binary file in order for the user-space code to be referenced. The "wrapper" functions in `oscalls.c` are written in assembly language, and may need to be rewritten when using different compilers.

---

[3] As a side note, much of the assembly code in Helium that accesses members of data structures assumes certain offsets for member elements of those elements. For example, this code assumes that the `PC` element of the `currThread` object is 16 bytes from the base address of the object. Consequently, if new member elements are added to the `THREAD` struct, they should be added at the end of the list of members so existing code does not have to be modified. If a new member of type int were added to the THREAD struct before the PC element, the offset of the PC element would change to 20 and this code would not work.

2. The wrapper function pushes all registers onto the user stack using the `movem` instruction.
3. Load the system call parameters into the appropriate CPU registers. These will vary depending on the system call.
4. Call the appropriate trap instruction. The trap instruction will cause (a) execution to jump to the appropriate exception handler and (b) the processor to enter supervisor mode. The return address and status register for the user thread will be pushed onto the supervisor stack.
5. The trap handler will disable interrupts using the instruction

```
move.w #0x2700,SR
```

6. The trap exception handler will perform the function requested by the user thread. Many of the trap handlers are written purely in assembly language and just call a separate function written in C that actually performs the action requested by the user thread. Some trap handlers are shared by multiple OS calls. In these cases, the trap handler will determine which OS call it should execute by examining the contents of register D7 and then call the appropriate C function to execute that OS call.