CPS311 Lecture: Exceptions

Last revised September 16, 2019

Objectives:

1. To introduce the concept of interrupts and exception
2. To introduce signal handling in Unix
3. To introduce the use of break/syscall operations

Materials:

1. Ability to demo original version of quadratic equation solver (in Procedures demos) plus revised version with signal handler (in Exceptions demos),
2. Projectable of `setjmp/longjmp` operation
3. Handout showing code added to use a MIPS/Unix signal catcher

I. **Introduction**

A. In our discussion thus far, we have assumed that the flow of control within a program proceeds from instruction to successive instruction, except when explicitly altered by a branch or procedure call instruction.

B. There are, however, two kinds of events that can cause the flow of execution to be altered apart from an explicit instruction in the currently running program.

1. INTERRUPTS can be generated by external devices to indicate that they need service by the CPU. Examples:

a) When a disk controller has been asked to transfer a block of data between memory and the disk, it will issue an interrupt request when the transfer is complete.

b) The controller for a keyboard may issue an interrupt request every time a key is struck.

c) A network interface may issue an interrupt when a packet arrives.

d) The system clock issues an interrupt request at regular intervals so that software can keep track of the time of day and can manage sharing of the CPU resources among different users (time-slicing.)

2. EXCEPTIONS arise from the operation of the currently running program. They are usually (but not always) the result of some flaw in the program.

   Examples:

   a) Memory access violations - attempting to access a region of memory the program is not allowed to access (usually resulting from an error in address computation or an infinite loop.)

   b) Illegal instructions - usually the result of bad code or attempting to execute data.

   c) Invalid arithmetic results - e.g. overflow

3. Interrupts and exceptions differ in that the former are ASYNCHRONOUS while the latter are SYNCHRONOUS.

   a) Interrupts arise from conditions outside the currently running program. Thus, if the same program is run again, it is very unlikely that the same interrupt will occur at the same point in the program.

   b) Exceptions arise from the behavior of the currently running program. Ordinarily, if a given program gives rise to an exception, then it will generate that same exception at the same point in its execution every time it is run on the same input data.

4. It is often - but not always - the case that exceptions arise from some sort of error in the running program.

C. On most computers, dealing with interrupts and exceptions involves features of both the hardware and the operating system.

1. In response to an interrupt or exception, the hardware typically does a forced jump to an operating system routine that is designed to handle the specific condition detected. In so doing, it saves enough information about the context in which the interrupt/exception exception has been dealt with.

2. The operating system routine that is invoked must analyze the specific event and decide on an appropriate action, which may or may not alter the execution of the program that was running at the time the interrupt/exception occurred.

3. In the remainder of this lecture, we will look at how a specific CPU/ operating system combinations deal with interrupts/exceptions: Unix on MIPS. The hardware part of the process is MIPS specific, while the operating system part is similar on all Unix-like systems (including Linux).

II. MIPS/Unix Interrupt/Exception Handling

A. On most systems, including MIPS, interrupts and exceptions are handled the same way by the hardware.

1. On MIPS, two special CPU registers are loaded when an interrupt or exception occurs:

   EPC = PC value at time interrupt/exception occurred
   Cause = code indicating what happened

2. Control goes to a fixed address in memory, which must contain a routine that deciphers the Cause and takes the appropriate action (The routine located at this point is part of the operating system.)

3. In this regard, the MIPS hardware is simpler than many other machines. It is common to find that the hardware branches to a different location in memory, depending on WHICH interrupt or exception occurred.

   Most CPU architectures define a TABLE of addresses (commonly called the interrupt vectors) which specify which routine is invoked for which condition. On MIPS, all interrupts and exceptions go to the same routine, which must in turn determine what happened (from the Cause register) and then initiate appropriate action. (We will discuss the notion of interrupt vectors later in the course.)

B. In the case of an interrupt, the operating system code transfers control to an interrupt handler that is part of the device driver for the specific device that interrupted, which takes it from there.

C. In the case of an exception, Unix converts the exception to a signal which is delivered to the process through the Unix signal mechanism (discussed in the Unix manual pages under the heading signal.)

1. In Unix, signals can be sent from a variety of sources.

   a) An exception within the current program, as discussed here.

   b) User action at the keyboard - e.g. typing ^C to terminate the program, or ^Z to suspend execution.

   c) Another program using the kill system service to send a signal.

2. Each signal has a signal number - a small integer. (On the version of Unix on our mips system, it's in the range 1..64; on Linux

3. The default action for a signal is one of the following - depending on the specific signal:

   a) Terminate the program.

b) Terminate the program with a core dump (a copy of the current state of the memory and registers contained in a file called core, which can then be analyzed by a debugger to determine the cause of the error.)

c) Stop the program - with the possibility of resumption later.

d) Ignore the signal.

4. Demo: original quadratic equation solver - with overflow: (Use B = 65536)

D. The default handling of a signal can be overridden by using the `signal` system service. This system service takes two parameters - an integer that identifies the signal, and the address of a function that is to be called when that signal occurs. Thereafter, any occurrence of that particular signal will be "caught" by the handler.

HANDOUT - Quadratic equation solver extended to handle overflow (setup code that calls signal).

E. Since once a procedure establishes a handler it becomes possible for that handler to be invoked by an exception in any procedure that it calls (directly or indirectly), there is sometimes a need to unwind the stack of pending procedure calls to get back to the procedure that actually established the handler. On Unix, this is handled by `setjmp` and `longjmp`.

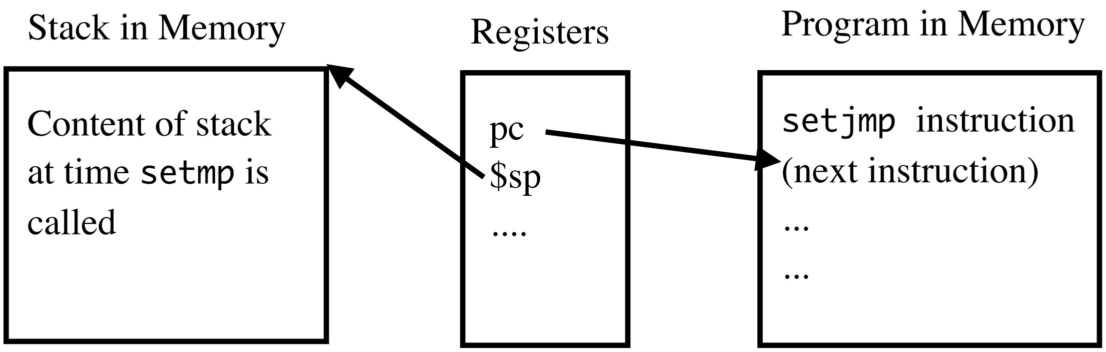
1. HANDOUT - `setjmp` in entry protocol and `longjmp` in handler

2. DEMO: program with overflow handling - same data

3. The operation of `setjmp` and `longjmp`

PROJECT

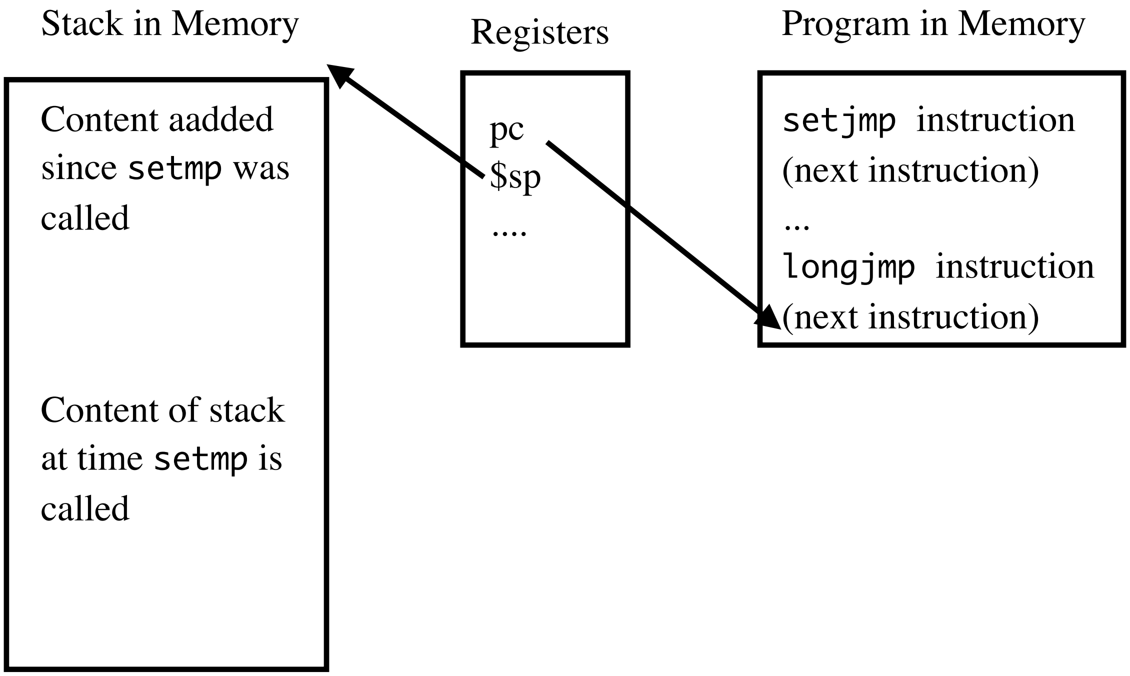
Situation when `setjmp` is called



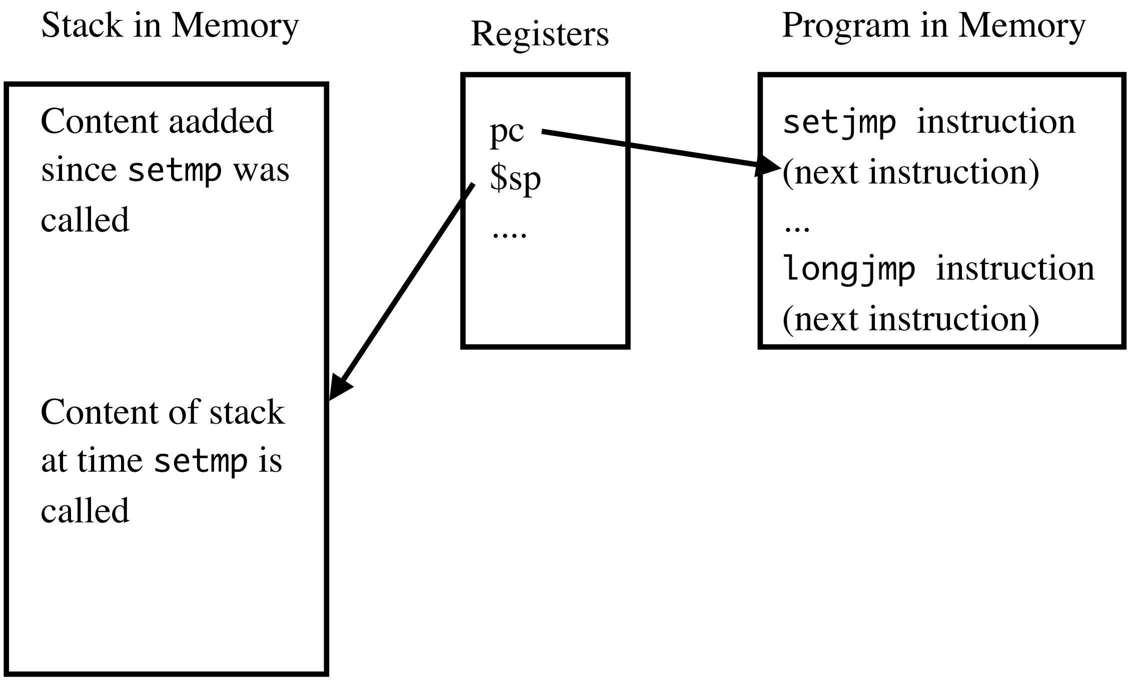
Result of `setjmp`:

- Stack, registers and program are unchanged
- A copy of the content of the registers is saved in the environment passed as a parameter to `setjmp`
- `setjmp` returns 0 to the caller

Situation when `longjmp` is called



Result of `longjmp`



- Registers are restored to the values saved in the environment passed as a parameter to `setjmp`
- `setjmp` returns 1 to the called

III.Deliberately Caused Exceptions

A. Though exceptions normally arise due to program errors detected by the hardware, there are some times when it is useful for the software to be able to create an exception

B. There are three reasons why this is necessary.

1. A program may make use of a deliberately-thrown exceptiojn to handle problems occurring in the program (e.g. exceptions used in the Library project in CPS122.)

2. Debuggers make use of a special instruction that causes a signal(caught by the debugger) in order to allow breakpoints to be set in a progam.

e.g. the "stop in main" command you used in an earlier lab caused dbx to replace the first instruction in main with a break instruction (after saving a copy of the real instruction that was there.) A signal handler established by the debugger, in turn, allows dbx to regain control when execution reaches this point.

On MIPS, this instruction is called break.

3. Calling operating system services.

   a) In order to protect system integrity against erroneous (or malicious) user programs, as you recall from CPS221 most CPU's can run in one of two modes, often called kernel mode and user mode.

      (1) The operating system runs with the CPU in kernel mode.

      (2) All other programs run with the CPU in user mode.

      (3) Certain operations are only permitted when the CPU is in kernel mode - e.g

          - Halting the CPU
          - IO operations
          - Access to certain regions of memory (e.g. that which contains the operating system and other users' programs on a multi-user system.

          Thus, only operating system code can perform these operations.

   b) When an interrupt or exception occurs, the CPU is switched into kernel mode by the hardware. This doesn't result in any danger to system integrity, because the hardware also transfers control to code at a known address, which is part of the operating system (and resides in a region of memory that cannot be altered while the CPU is in user mode.)

c) When a user program needs an operating system service (e.g. an IO operation), it executes a special instruction that causes n exception. The operating system interprets execution of this instruction as a request for a system service, and takes ppropriate action (assuming the program is requesting a service legally.)\

d) Various ISA's call this instruction "`syscall`" or "`trap`" or "`change mode to kernle`". (On MIPS it is called `syscall`).