Stack Trace

From OSDev Wiki

A stack trace is debugging output, normally sent to a log file or a debug window that shows the hierarchy of callers that called the current function. A stack trace is generated by analysing the stack to find each stack frame. The addresses of the functions called can be retrieved from each stack frame and the names of the functions displayed.

To implement a stack trace you have to know the structure of the stack frames, which is shown in the article Stack for X86 CDECL.

Walking the stack

Often a stack trace is written in assembly as it involves finding the current value of the EBP register. To write a stack trace routine in a higher-level language you will need to find EBP. This can be done by using a small assembly function or inline assembly, or by using __builtin_frame_address(0) if you use the GCC compiler. On some platform (e.g. x86), the compiler does not necessary save the EBP on the stack. For example, for gcc, use the -fno-omit-frame-pointer to make sure that the EBP is saved. Note that omission of the frame pointer merely causes functions to be missed from the backtrace.

The following C++ code shows how (given the existence of a Trace function) this can be used to walk up the stack:

```
/* Assume, as is often the case, that EBP is the first thing pushed. If not, w
struct stackframe {
  struct stackframe* ebp;
  uint32_t eip;
};
void Debug::TraceStackTrace(unsigned int MaxFrames)
{
    struct stackframe *stk;
    asm ("movl %%ebp,%0" : "r"(stk) ::);
    Trace("Stack trace:\n");
    for(unsigned int frame = 0; stk && frame < MaxFrames; ++frame)
    {
        // Unwind to previous stack frame
        Trace("  0x{0:16}     \n", stk->eip);
        stk = stk->ebp;
    }
}
```

Note that the above code and GDB backtracing require a NULL %ebp, to know when to stop. Otherwise the traces will run off into garbage. To account for this, set up a NULL stack frame before you jump to your C entry point:

```
mov $stack_end, %esp ; Initialize %esp
...
xor %ebp, %ebp       ; Set %ebp to NULL
call kmain           ; According to calling convention, kmain will save %ebp (
```

With this, stack tracers will see the NULL %ebp as the end of the trace.

Assembly Implementation

This assembly implementation for x86 uses the same algorithm as above and similarly relies on a NULL base pointer to be placed near the top of the stack. Rather than print the contents of the stack, however, it builds an array of addresses which can then be resolved into symbol names.

```
; Walks backwards through the call stack and builds a list of return addresses
; Args:
;   * Array of 32-bit addresses.
;   * Maximum number of elements in array.
; Return value: The number of addresses stored in the array.
; Calling convention: cdecl
[global walk_stack]
walk_stack:
    ; Create stack frame & save caller's EDI and EBX.
    push ebp
    mov  ebp,       esp
    sub  esp,       8
    mov  [ebp - 4], edi
    mov  [ebp - 8], ebx
    ; Set up local registers.
    xor  eax,       eax         ; EAX = return value (number of stack frames f
    mov  ebx,       [esp +  8]  ; EBX = old EBP.
    mov  edi,       [esp + 16]  ; Destination array pointer in EDI.
    mov  ecx,       [esp + 20]  ; Maximum array size in ECX.
.walk:
    ; Walk backwards through EBP linked list, storing return addresses in EDI
    test ebx,       ebx
    jz   .done
    mov  edx,       [ebx +  4]  ; EDX = previous stack frame's IP.
    mov  ebx,       [ebx +  0]  ; EBX = previous stack frame's BP.
    mov  [edi],     edx         ; Copy IP.
    add  edi,       4
    inc  eax
    loop .walk
.done:
    ; Restore caller's EDI and EBX, leave stack frame & return EAX.
    mov  edi,       [ebp - 4]
    mov  ebx,       [ebp - 8]
    leave
    ret
```

Resolving Function Names

The next step in producing meaningful output from a stack trace is to find the names of the functions containing the addresses found during the stack walk.

When looking up the name of a function you have to find the biggest address smaller than the value you are looking for. This is because the return address saved by the call is the address of the jsr instruction, which will be offset within the function that is making the call.

To get the information you need to lookup function names you will need to either include debugging symbols in your kernel or load the map file created by your linker into the kernel's memory space. The map file shows the addresses of each of your functions. While you could include the entire map file, it is often quite large and inefficiently stored. Not only this but often functions are not listed in the order that they appear in the object file and the format is not amenable to tracing through to find a specific function.

One possible solution is to pre-process your map file to produce a smaller, more useful format for it. You could do this in a way that allows either binary or linear searching for a particular address. See NobleTech's Web site[1] (http://www.nobletech.co.uk/Products/PenPot/Design/Kernel/Debug/FnNameLookup.aspx) for C# code showing a way of reading the map file produced by GNU ld and outputting a binary file that allows more efficient linear searching for symbols. A binary Win32 console application to do the pre-processing is also available for free from that site. C++ code that can be used in your kernel to look up function names in the pre-processed file format is also shown.