James Molloy's Tutorial Known Bugs

From OSDev Wiki

Several sources - including this Wiki - point to James Molloy's Roll your own toy UNIX-clone OS (http://www.jamesmolloy.co.uk/tutorial_html/) Tutorial as a starting point. This is fine, but the tutorial has some well-known weak points that cause trouble for people again and again. It's not uncommon that well-established members traced back mysterious bugs to early parts of their operating systems based on this tutorial. Nonetheless, it's one of the best introductory tutorials out there even if it has the occasional landmine. This article is meant to preempt issues arising from following the tutorial and to aid those that have encountered such problems. It is generally recommended to be sceptical of its advise on how to design your kernel and compare its information against this wiki. Some issues are quite subtle and only experts will recognize them.

Contents

Before you follow the tutorial

Main article: Bare Bones

It is recommended that you follow the this wiki's standard tutorial Bare Bones before you begin with the tutorial. This ensures you get the a proper cross-compiler and use the proper compilation options. If you have already followed the tutorial, please compare your current build environment against the recommended practices covered by Bare Bones.

Problem: Not using a cross-compiler

> *Main article:* GCC Cross-Compiler

This tutorial was written years before it was recognized as standard practice to use a cross-compiler. As such, you should disregard most of the build instructions in Chapter 1 Environment Setup and instead follow Bare Bones. You should use the Bare Bones linker script instead as well as the the boot assembly from Bare Bones. Floppies are an obsolete technology and it is advisable to create a bootable cdrom image instead.

Problem: __cdecl calling convention

The tutorial states that the `__cdecl` calling convention is used. This is, however, a Windows term. Your cross-compiler uses a similar calling convention but it is called the System V ABI for i386. It is advisable to understand this calling convention in depth, especially parts about how the parameters on the stack are clobbered and how structure parameters are passed. This will be very useful later and will help you avoid a later subtle bug. The function call example in 2.3 neglects to add 12 to esp following the call instruction, so the three parameters are never popped.

Problem: CFLAGS

The tutorial recommends using these compilation options `-nostdlib -nostdinc -fno-builtin -fno-stack-protector`, but this is not the recommended practice. The tutorial neglects to pass the important `-ffreestanding` option. See Bare Bones on how to correctly compile C kernel files and how to correctly link the kernel.

Problem: Not using libgcc

> *Main article:* libgcc

The tutorial disables libc and libgcc through the `-nodefaultlibs` option (implied by `-nostdlib`) but neglects to add back libgcc during the link.

Problem: Not setting a stack

> *Main article:* Stack#Setup the stack

The tutorial neglects to set a stack in the initial boot file and relies on the bootloader using an appropriate stack. You should instead declare your own stack as an array and use that instead, such that you have control of the situation.

Problem: main function

This isn't a regular `main` function: The name `main` is actually a special case in C and it would be inadvisable to call it that. You should call it something like `kernel_main` instead.

Problem: Data types

The tutorial uses non-standard data types such as `u32int` while the international C standard (99 revision) introduces standard fixed-width data types like `uint32_t` that you should use instead. Simply include `<stdint.h>` which comes with your cross-compiler and works even in freestanding mode. This is the reason you should not pass the `-nostdinc` option.

Problem: Inline Assembly

Main article: Inline Assembly/Examples

The tutorial uses inline assembly, which is notoriously hard to get exactly correct. The smallest error can emit assembly that fully works until one day the optimizer does things differently. While the inline assembly looks correct to me at a glance, please compare the inline assembly with the examples on this wiki.

Problem: Missing functions

The GCC documentation mentions that the `memset`, `memcpy`, `memmove` and `memcmp` functions must always be present. The compiler uses these automatically for certain optimization purposes and even code that doesn't use them can automatically generate calls to them. You should add them at your earliest convenience.

Problem: Interrupt handlers corrupt interrupted state

This article previously told you to know the ABI. If you do you will see a huge problem in the interrupt.s suggested by the tutorial: It breaks the ABI for structure passing! It creates an instance of the `struct registers` on the stack and then passes it by value to the `isr_handler` function and then assumes the structure is intact afterwards. However, the function parameters on the stack belongs to the function and it is allowed to trash these values as it sees fit (if you need to know whether the compiler actually does this, you are thinking the wrong way, but it actually does). There are two ways around this. The most practical method is to pass the structure as a pointer instead, which allows you to explicitly edit the register state when needed - very useful for system calls, without having the compiler randomly doing it for you. The compiler can still edit the pointer on the stack when it's not specifically needed. The second option is to make another copy the structure and pass that.

Problem: ISR 17 and 21 have error codes

The interrupt handling code in the downloadable code have a bug where it handles ISR 17 and 21 by pushing a fake error code, but the CPU does push error codes here.

Problem: struct registers::esp is useless

The `struct registers` structure has a esp member that is one of the values pushed by pusha. This value is, however, ignored by popa for obvious reasons. You should rename it to `useless_value` and rename `useresp` to `esp` instead. The value is useless because it has to do with the current stack context, not what was interrupted.

Problem: __attribute__((packed))

This attribute packs the associated structure. This is useful in a few cases, such as the IDT and GDT code (actually just the IDT, GDT and TSS pointers). However, the tutorial tends to randomly attach it to every struct parameter, even where it isn't even needed. It is only needed where you badly want aligned structure members, it doesn't do anything if all the structure members were already naturally aligned. Otherwise, the compiler will automatically insert gaps between structure members so each begins at its own natural alignment.

Problem: cli and sti in interrupt handlers

The interrupt.s file invokes the `cli` and `sti` in the interrupt handler to disable and enable interrupts, as if the author didn't know whether the interrupt handlers were run with interrupts on or off. You can control whether they are run with interrupts on and off by simply deciding it in your IDT entry for the particular interrupt. The `sti` during the interrupt handler end is also useless as `iret` loads the eflags value from the stack, which contains a bit telling whether interrupts are on or off; in other words the interrupt handler automatically restores interrupts whether or not interrupts were enabled before this interrupt.

Problem: kmalloc isn't properly aligned

Each data type in C has its own natural alignment. For instance, on the ABI that you are using an int is a signed 32-bit value that must be 32-bit aligned in memory (4 byte alignment). The same applies for structures, where the alignment of the whole structure is the maximum alignment of all its members. It is undefined behavior to access an unaligned value. For instance, you could decide you want an int at a particular unaligned (for an int) memory address and construct a pointer to it. When you attempt to write an int value to that pointer, undefined behavior happens. Furthermore, SIMD registers have alignment needs that are bigger than their individual components.

The `kmalloc` function in 6.4.1 only 1-byte aligns or page-aligns its memory address. This means you can only reliably use it allocate memories for chars (size 1), but not any larger types unless you use page-alignment. A proper malloc implementation returns pointers that are aligned such that they are suitable for all the common types, for instance it could be 64-bit (8-byte) aligned. You'll also want to modify the parameters such that it uses `size_t` appropriately rather than `u32int`.

Additionally the check if the address is page aligned is wrong.

```
if (align == 1 && (placement_address & 0xFFFFF000)) // If the address is not already page-aligned
```

should be

```
if (align == 1 && (placement_address & 0x00000FFF)) // If the address is not already page-aligned
```

Problem: Paging Code

The paging code isn't terribly good and it is worth it to fully understand paging and design it all yourself. Paging code tends to be quite ugly, but it'll probably be decent after your fifth design revision. There is no need to always re-enable paging in `switch_page_directory`, it is likely best to have a special function the first time paging is enabled. The Inline Assembly in 6.4.5. doesn't need to be volatile as it is simply reading a memory value, which has no side-effects and it is acceptable if the compiler optimizes it away if the value is never used.

Problem: Heap Code

It is probably best that you write your own heap implementation.

There is an operator precedence bug in `find_smallest_hole()` that will cause bad allocations and memory overwrites if attempting to fork in user mode later on.

To fix the problem, change this:

```
if ((location+sizeof(header_t)) & 0xFFFFF000 != 0)
```

to this:

```
if ((location+sizeof(header_t) & 0xFFFFF000) != 0)
```

See the section on user mode below for more details.

Problem: VFS Code

The name of files on Unix are stored in the directory entries rather than the inode itself (`struct fs_node` here), this allows Unix files to have multiple names and even none if the file is deleted but the inode is not yet closed in all programs.

Problem: multiboot.h

It's advisable to get a copy of `multiboot.h` from the GRUB source code rather than copied from the tutorial. Beware, the copy in the GRUB documentation is out of date, use one from an official release.

Problem: Multitasking

It is strongly recommended that you write your own implementation of this and disregard the tutorial. The tutorial attempts to implement forking kernel threads by searching for magic values on the stack, which is insanity. If you wish to create a new kernel thread, simply decide which registers it should have and point its stack pointer at its freshly allocated stack. It will then start executing at your desired entry point. The part where it disables paging is bad and you should just map the source and destination physical frames at appropriate virtual addresses and memcpy with paging on at all times. Section 9.3 in particular is insanity and has blown up at least one well-established hobby operating system.

Inline Assembly optimiser problem with GCC 4.8

As mentioned above, writing Inline Assembly can be tricky. The original Inline Assembly is this:

```
  asm volatile("         \
    cli;                 \
    mov %0, %%ecx;       \
    mov %1, %%esp;       \
    mov %2, %%ebp;       \
    mov %3, %%cr3;       \
    mov $0x12345, %%eax; \
    sti;                 \
    jmp *%%ecx           "
              : : "r"(eip), "r"(esp), "r"(ebp), "r"(current_directory->physicalAddr));
```

Everything works fine when using gcc-4.2.4. However, the gcc-4.8.4 optimizer produces the following assembly (produced with **objdump -d src/kernel**):

```
 10387c:   fa                      cli
 10387d:   89 c1                   mov    %eax,%ecx
 10387f:   89 d4                   mov    %edx,%esp
 103881:   89 cd                   mov    %ecx,%ebp
 103883:   0f 22 db                mov    %ebx,%cr3
 103886:   b8 45 23 01 00          mov    $0x12345,%eax
```

```
 10388b:   fb                          sti
 10388c:   ff e1                       jmp    *%ecx
```

Note how the EAX register is assigned to the ECX register. However, later on the ECX register is assigned to EBP register. The reason for this is that the optimizer was using the EAX register to store the EIP variable and the ECX register to store the EBP variable. This results in the EIP variable being assigned to the ECX register *as well as* the EBP register. This leads to a subsequent **ret** statement sending the CPU to some invalid memory location.

A way to fix this is to remove the Inline Assembly by, for example adding this to **process.s**:

```
; Here we:
; * Stop interrupts so we don't get interrupted.
; * Temporarily put the new EIP location in ECX.
; * Temporarily put the new page directory's physical address in EAX.
; * Set the base and stack pointers
; * Set the page directory
; * Put a dummy value (0x12345) in EAX so that above we can recognize that we've just
;   switched task.
; * Restart interrupts. The STI instruction has a delay - it doesn't take effect until after
;   the next instruction.
; * Jump to the location in ECX (remember we put the new EIP in there).

[GLOBAL perform_task_switch]
perform_task_switch:
     cli;
     mov ecx, [esp+4]   ; EIP
     mov eax, [esp+8]   ; physical address of current directory
     mov ebp, [esp+12]  ; EBP
     mov esp, [esp+16]  ; ESP
     mov cr3, eax       ; set the page directory
     mov eax, 0x12345   ; magic number to detect a task switch
     sti;
     jmp ecx
```

Then edit **task.c** and add this to the top of the file:

```
extern void perform_task_switch(u32int, u32int, u32int, u32int);
```

and replace the Inline Assembly with:

```
perform_task_switch(eip, current_directory->physicalAddr, ebp, esp);
```

Problem: User mode

There are several problems. The downloadable code has everything fixed except a `find_smallest_hole()` page-aligned heap allocation bug.

Problem 1: nasm byte keyword causing `0x80` to become `0xffffff80`

This code is at fault:

```
 %macro ISR_NOERRCODE 1  ; define a macro, taking one parameter
  [GLOBAL isr%1]        ; %1 accesses the first parameter.
  isr%1:
    cli
    push byte 0
    push byte %1
    jmp isr_common_stub
%endmacro
```

It should be:

```
%macro ISR_NOERRCODE 1  ; define a macro, taking one parameter
  [GLOBAL isr%1]        ; %1 accesses the first parameter.
  isr%1:
    cli
    push byte 0
    push %1
    jmp isr_common_stub
%endmacro
```

Problem 2: Don't forget to allow interrupts in user mode in idt_set_gate

Find this comment in chapter 4 and uncomment the code:

```
   // We must uncomment the OR below when we get to using user-mode.
   // It sets the interrupt gate's privilege level to 3.
   idt_entries[num].flags   = flags /* | 0x60 */;
```

Problem 3: regs var must called by reference instead of by value in the irq and isr handlers

Various changes are needed.

Change this in `page_fault()`:

```
void page_fault(registers_t regs)
// to
void page_fault(registers_t *regs)
```

And in `isr_handler()` and `irq_handler()`, change

```
handler(regs);
```

to

```
handler(&regs);
```

This fixes a problem where the syscall hander won't get called.

Problem 4: Missing documentation around set_kernel_stack

`KERNEL_STACK_SIZE` needs defining in `task.h`:

```
#define KERNEL_STACK_SIZE 2048       // Use a 2kb kernel stack.
```

Also, some code needs to be added to four sections in `task.c`

In `initialise_tasking()`

```
 current_task->kernel_stack = kmalloc_a(KERNEL_STACK_SIZE);
```

In `fork()`

```
current_task->kernel_stack = kmalloc_a(KERNEL_STACK_SIZE);
```

In `switch_task()`

```
set_kernel_stack(current_task->kernel_stack+KERNEL_STACK_SIZE);
```

In `switch_to_user_mode()`

```
set_kernel_stack(current_task->kernel_stack+KERNEL_STACK_SIZE);
```

Problem 5: find_smallest_hole() bug in heap code causing fork() to page fault

This bug from the heap chapter may not hit you until now. The bug results in the newly allocated `kernel_stack` messing up the page directory, causing `clone_directory()` to fail in `fork()`. See the heap section above for the details.

Conclusion

The tutorial isn't bad as an example, but its design is not optimal and some parts of it are just plain bad (see multitasking). Indeed, you should just use it to get started and diverge from it as fast as possible, only using the tutorial when you need an example or can't implement it yourself. You should prefer consulting information on this wiki if possible. I haven't yet located all the problems in the tutorial, and some are quite minor and not technically problems but just small subjective design flaws. It is worth anticipating whether your future self will be removing tutorial code from your operating system and thus saving effort by never putting it there in the first place.

Categories: OS Development | Troubleshooting | FAQ
