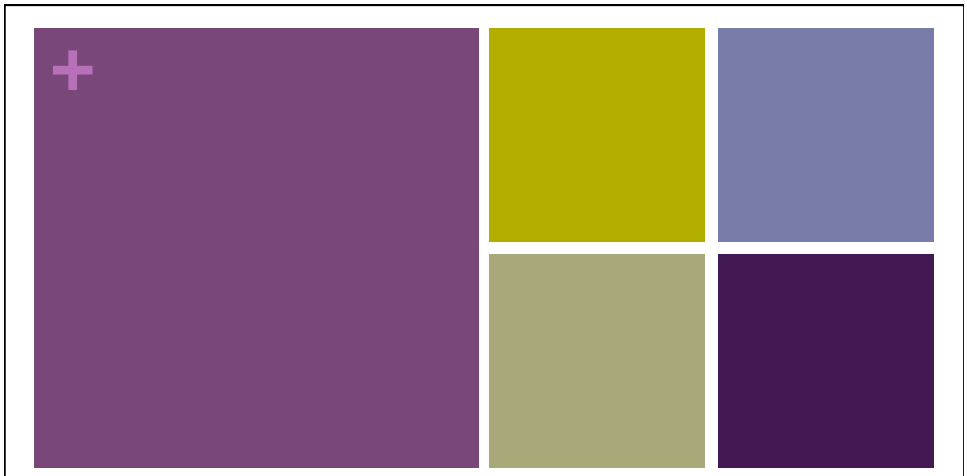

Projects: Developing an OS Kernel for x86

Introduction

The boot process

- After the power button has been pressed…
  - Power supply certifies that can supply the correct amount of power to all devices.
  - Sends BIOS "power_good" signal.
- Motherboard Control, POST (Power On Self Test)
  - Confirms power level, tests memory for corruptions
  - Addresses proprietary chips.
  - The BIOS (Basic Input Output System) takes over
  - Finds and loads boot sector (512). Executes image in the boot sector.
- Bootsector image loads bootloader
  - No size constraint
- … and off you go

The Kernel Entry Point (start.asm)

```
[BITS 32]
global start
start:
    mov esp, _sys_stack     ; This points the stack to our new stack area
    jmp stublet

; This part MUST be 4byte aligned, so we solve that issue using 'ALIGN 4'
ALIGN 4
mboot:
    ; Multiboot macros to make a few lines later more readable
    MBOOT_PAGE_ALIGN    equ 1<<0
    MBOOT_MEMORY_INFO   equ 1<<1
    MBOOT_AOUT_KLUDGE   equ 1<<16
    MBOOT_HEADER_MAGIC  equ 0x1BADB002
    MBOOT_HEADER_FLAGS  equ MBOOT_PAGE_ALIGN | MBOOT_MEMORY_INFO | MBOOT_AOUT_KLUDGE
    MULTIBOOT_CHECKSUM  equ -(MBOOT_HEADER_MAGIC + MBOOT_HEADER_FLAGS)
    EXTERN code, bss, end

    ; This is the GRUB Multiboot header. A boot signature
    dd MBOOT_HEADER_MAGIC ; declare 4-byte variables
    dd MBOOT_HEADER_FLAGS
    dd MBOOT_CHECKSUM

    ; AOUT kludge - must be physical addresses. Make a note of these:
    ; The linker script fills in the data for these ones!
    dd mboot
    dd code
    dd bss
    dd end
    dd start

stublet:
    ; Add code here to call allocators of global objects.
    EXTERN _main ; defined in another file
    call   _main
    ; Add code here to call destructors for global objects.
    jmp $  ; infinite loop after we return from main. (don't do this in a real system)

; Here is the definition of our BSS section. We'll use it just to store the stack.
; Remember that a stack grows downwards, so we declare the size of the data before declaring
; the identifier '_sys_stack'
SECTION .bss
    resb 8192               ; This reserves 8KBytes of memory here
_sys_stack:
```

The Linker Script (linker.ld)

```
OUTPUT_FORMAT("binary")
ENTRY(start)
phys = 0x00100000;
SECTIONS
{

  .text phys : AT(phys) {
    code = .;
    *(.text)
    *(.rodata)
    . = ALIGN(4096);
  }
  .data : AT(phys + (data - code))
  {
    data = .;
    *(.data)
    . = ALIGN(4096);
  }
  .bss : AT(phys + (bss - code))
  {
    bss = .;
    *(.bss)
    . = ALIGN(4096);
  }
  end = .;
}
```

Kernel Development in C++

(see "Writing a Kernel in C++" by David Stout)

- Beware of Run-Time Support!

"Except for the new, delete, typeid, dynamic_cast, and throw operators and the try-block, individual C++ expressions and statements need no run-time support."

Bjarne Stroustrup, "The C++ Programming Language, 3rd ed."

- Features that require run-time support:
  - Built-in functions (new, delete)
  - Run-Time type information (typeid, dynamic_cast)
  - Exception handling (throw, try-block)

Q: What else do we need?

A: If our code refers to it, we need a C++ standard library.

Compiling C++ Code …

- **gxx –ffreestanding –fnostdlib –fno-builtin –fno-rtti –fno-exceptions –c *.cpp**

-ffreestanding: assume that standard libraries & main may not exist.

-fnostdlib:

-fno-builtin: Do not recognize any built in functions (e.g. new, delete).

-fno-rtti: Do not generate run-time type descriptors information.

-fno-exceptions: Do not generate code to support run-time exceptions.

We are disabling a lot of functions…

- What happens before and after the main() function?s
- _main() is typically called before main() function
  - handles constructors of global and static objects.
- _atexit() is called after main() function exits.
  - handles destructors of global and static objects.

➔ Global and static objects are off-limits until we add support for them!(*)

(*) Even then, the implementation of _main() and _atexit() will be compiler specific.

The Kernel Entry Point (start.asm)

```
[BITS 32]
global start
start:
    mov esp, _
    jmp stubl

; This part M
ALIGN 4
mboot:
    ; Multibo
    MBOOT_PAG
    MBOOT_MEM
    MBOOT_AOU
    MBOOT_HEA
    MBOOT_HEA
    MULTIBOOT
    EXTERN co

    ; This is
    dd MBOOT_
    dd MBOOT_
    dd MBOOT_

    ; AOUT kl
    ; The lin
    dd mboot
    dd code
    dd bss
    dd end
    dd start

stublet:
    ; Add cod
    EXTERN _m
    call   _m
    ; Add cod
    jmp $  ;

; Here is the
; Remember th
; the identifier _sys_stack
SECTION .bss
    resb 8192               ; This reserves 8KBytes of memory here
_sys_stack:
```

```
stublet:
; Initilization of static global objects. This goes through each object
; in the ctors section of the object file, where the global constructors
; created by C++ are put, and calls it. Normally C++ compilers add some code
; to do this, but that code is in the standard library - which we do not include.
; See linker.ld to see where we tell the linker to put them.
   extern start_ctors, end_ctors, start_dtors, end_dtors

   static_ctors_loop:
      mov ebx, start_ctors
      jmp .test
   .body:
      call [ebx]
      add ebx,4
   .test:
      cmp ebx, end_ctors
      jb .body

; Entering the kernel proper.
   extern _main
   call _main

; Deinitialization of static global objects. This goes through each object
; in the dtors section of the object file, where the global destructors
; created by C++ are put, and calls it. Normally C++ compilers add some code
; to do this, but that code is in the standard library - which we do not include.
; See linker.ld to see where we tell the linker to put them.
   static_dtors_loop:
       mov ebx, start_dtors
       jmp .test
   .body:
       call [ebx]
       add ebx,4
   .test:
       cmp ebx, end_dtors
       jb .body

   jmp $
```

The Linker Script (linker.ld)

```
OUTPUT_FORMAT("binary")
ENTRY(start)
phys = 0x00100000;
SECTIONS
{
  .text phys : AT(phys) {
    code = .;
    *(.text)
    *(.gnu.linkonce.t.*)
    *(.gnu.linkonce.r.*)
    *(.rodata)
    . = ALIGN(4096);
  }
  .data : AT(phys + (data - code))
  {
    data = .;
    *(.data)
    start_ctors = .;
    *(.ctor*)
    end_ctors = .;
    start_dtors = .;
    *(.dtor*)
    end_dtors = .;
    *(.gnu.linkonce.d.*)
    . = ALIGN(4096);
  }
  .bss : AT(phys + (bss - code))
  {
    bss = .;
    *(.bss)
    *(.gnu.linkonce.b.*)
    . = ALIGN(4096);
  }
  end = .;
}
```

Generating the kernel.bin File

```
/* file 'kernel.C' */

void main() {
  /* This is where the kernel code would come
*/

  /* for now we just idle … */
  for(;;);
}
```

- Assume:
  - File start.asm contains code with multiboot header and entry point.
  - File kernel.C contains the "kernel" code.
  - We compile and link everything using the following simple makefile:

```
GCCOPT  -nostartfiles -nostdlib -fno-rtti -fno-exceptions
start.o: start.asm
        nasm –f aout –o start.o start.asm

kernel.o: kernel.c
        gcc $(GCCOPT) –c kernel.c

kernel.bin: loader.o kernel.o
        ld –T linker.ld –o kernel.bin start.o kernel.o
```

Loading the Kernel onto a Floppy Image

- Download the grub disk image **`dev_kernel_grub.img`** from the course web page.
  - Bootable floppy image with **`grub`** bootloader and demo kernel.
- Call it **`my_disk.img`**
- Mount the disk image:
  - **`filedisk /mount 0 my_disk.img g:`**
- Now you can copy your **`kernel.bin`** file to the disk.
- Unmount the disk image!
  - **`filedisk /umount g:`**
- You now have a bootable floppy disk with your kernel.
- For details on how to run your kernel, see handout for MP1!