Boot code tutorial - Housekeeping

- The objectives of this session is to gain clarity and appreciation of the role of the boot process described during the lecture

- It is anticipated that at the end of the session you will understand how/why the boot code needs to copy itself and move itself to another area of memory.
  - how commenting (or dry running code) can aid understanding
  - and also why Linus wanted to get into 32 bit mode asap!

- Work on the following problem (in the next slide) - asking questions if required for 20 minutes
- The lecturer will then show your their worked comments for 10 minutes, after that you can correct your work and continue for the remainder of the tutorial

Boot code Tutorial Question

- read the following code (`boot.S`) on the next slide and write a simple explanation of what it is perfoming
  - start commenting at the line "start commenting at this point" in red
- the assembler moves data from the right operand to the left operand
  - so for example:
-
```
    mov  ax, #BOTOFMEMSEG
    mov  ss,ax
```
- could be commented as
-
```
    mov  ax, #BOTOFMEMSEG   !   ax = BOTOFMEMSEG
    mov  ss,ax              !   ss = ax
```

Boot code Tutorial Question

- another example of assembly code a comments are shown below:

```
    mov ax, #'1             !  ax = '1'
    push ax                 !  push ax to the stack
    call funct              !  call function func
```

- in the tiny model 8088 the microprocessor has
  - registers: `ax`, `bx`, `cx`, `dx`, `ss`, `cs`, `ds`, `es`
  - the `ax`, `bx`, `cx`, `dx` are all 16 bit and the programmer can access the high byte of the `ax` register via `ah` and the low byte via `al` (the same is true for `bx`, `cx` and `dx` registers.

Boot code Tutorial Question

- the `ss`, `cs`, `ds`, `es` registers are segment registers. `ss` is the stack segment, `cs` is the code segment `ds` is the data segment and `es` is the extra segment

- a segment register contains the address base / 16 of the section of memory
  - so if the stack segment `ss` contains the value `A000`, it means that the bottom physical address of the stack can be `A0000` and the top physical address of the stack is `AFFFF`
  - another way of viewing the segment register is that its smallest addressable unit is 16 bytes
  - the last 16 bytes are referenced by the stack pointer register, `sp`
  - sometimes 8088 documentation will refer to segment addressing
  - physical address = segment address * 16 + offset
  - so in the case of the stack pointer: physical = `ss * 16 + sp`

Boot code Tutorial Question

- in the assembly code below `0xABCD` means hexadecimal `ABCD` and `10` is `10` decimal

- you should log into GNU/Linux and open up the gnome calculator, set it to programming mode and choose the hexadecimal mode. You will find it useful when working out addresses.

Boot code Tutorial Question

- the code is the first boot code (the role of the first boot code was described during the lecture)

- **boot.S**

```
!
!   Bootstrap loader which must be <= 512 bytes of code
!                          and have no data segment!
!
entry _start
!
!
! Note that the operating system must be <= 9*64k as we place boot
!      and secondary boot code at 640k-64k..640k
!
```

Boot code Tutorial Question

boot.S

```
!
! The task of this piece of code is to load the secondary bootstrap
! code written in Modula-2 for the 8088/8086. We also set up the
! ss, cs, ds registers. We set them to the same value  640k-64k
! and we know that the secondary boot code is small and will fit into
! 64k including stack, data and code. This makes life alot easier!
!
!
```

Boot code Tutorial Question

- `boot.S`

```
TOPOFMEM     =    0xA0000        !  We assume every machine has at least 640k
TOPOFMEMSEG  =    TOPOFMEM / 16
SIXTYFOURKSEG=    0x10000 / 16
BOTOFMEMSEG  =    TOPOFMEMSEG - SIXTYFOURKSEG
STACKSIZE    =    0x1000         !  4k of stack space

BOOTSEG      =    0x7C00 / 16 ! This is where the BIOS puts the
                 ! first sector (boot sector).

MAXSECONDSEG =    0x8000 / 16   ! Max no of clicks of code for secondary boot

STACKCLKS    =    STACKSIZE / 16
STACKSEG     =    TOPOFMEMSEG - STACKCLKS        ! Assign the stack here

SECONDSEG    =    0x90200 / 16 ! BOTOFMEMSEG
                               ! Secondary boot code
SECTORSIZE   =    256  ! number of 2 byte words in a sector
SECONDSIZE   =    14   ! max number of sectors which may contain the
                       ! secondary boot.

!BOOTDRIVE   =    0x00 ! floppy (/dev/fd0 or a:)
BOOTDRIVE    =    0x80 ! harddisk (USB-HDD in the BIOS)
```

Boot code Tutorial Question

boot.S

```
!
! we choose 14 since:
!
! (i)     a 5 and 1/4 inch floppy drive has 15 sectors / track
! (ii)    a 3 and 1/2 inch floppy drive has 18 sectors / track
! (iii)   ROM BIOS insists that the bootsector is 1 sector.
! (iv)    therefore the minimum number of sectors available on track 1
!         is 15-1 = 14
!
! (v)     we could make boot more complicated (so it could load in second
!         from a range of tracks) but I really wanted to
!         keep it as simple as possible and jump into Modula-2 as soon
!         as possible. The whole intention of using a secondary
!         bootstage was to keep any complexity in a HLL
!
```

Boot code Tutorial Question

- start commenting at this point
  - ignore the data declarations, ie the `DW` and `DB` lines
  - `DW` means a declaration of a 16 bit value
  - `DB` means a declaration of an 8 bit value
  - don't comment a line which already contains a comment (a line with a `!` symbol)
  - only comment lines with an instruction on them
    - these lines start with at least one space

Boot code Tutorial Question

- `boot.S`

```
extern minbios_WriteChar  ! prototype for the debugging routine in C
                          ! void minbios_WriteChar (char ch);

_start:
    jmp after_sig     ! your first comment goes here!
    nop
OEM_ID:  .ascii "luk-boot"
BytesPerSector:  DW 0x0200
SectorsPerCluster:   DB 0x01
ReservedSectors: DW 0x0001
TotalFATs:   DB 0x02
MaxRootEntries:  DW 0x00E0
TotalSectorsSmall:   DW 0x0B40
MediaDescriptor: DB 0xF0
SectorsPerFAT:   DW 0x0009
SectorsPerTrack: DW 0x0012
NumHeads:    DW 0x0002
HiddenSectors:   DD 0x00000000
TotalSectorsLarge:   DD 0x00000000
DriveNumber: DB BOOTDRIVE
Flags:   DB 0x00
Signature:   DB 0x29
VolumeID:    DD 0xFFFFFFFF
VolumeLabel: .ascii "luk-bootusb"
SystemID:    .ascii "FAT12   "
```

Boot code Tutorial Question

boot.S

```
after_sig:
    mov     DriveNumber, dl  ! save the bios given bootdrive
    !
    ! set up stack
    !
    mov  ax, #BOTOFMEMSEG  ! and your second comment goes here
    mov  ss,ax
    mov  ax, #0xffe0
    mov  sp,ax

    mov  ax, #'1
    push ax
    call minbios_WriteChar
    pop  ax
```

Boot code Tutorial Question

boot.S

```
!
! now jump to _load at a new code segment BOOTSEG:
!
! we need to do this so that the cs is initialised to _start.
! The bios doesn't do this for us.
jmpi _load,BOOTSEG                ! jmp  far   _load:BOOTSEG
```

Boot code Tutorial Question

boot.S

```
_load:
     ! excellent now we set our Data segment = Code segment
     ! we need to do this because there are some OS parameters
     ! right at the end of this 512 disk sector.
     ! We pick these up in Util.S just before we go into Modula-2 in second

     mov  ax,cs
     mov  ds,ax          ! set up Data Segment

     mov  ax, #'2
     push ax
     call minbios_WriteChar
     pop  ax

     call    _SecondLoad
```

Boot code Tutorial Question

boot.S

```
    !
    ! now jump to _SecondLoad at a new code segment SECONDSEG:
    !
    mov  ax, #'4
    push ax
    call minbios_WriteChar
    pop  ax

    xor     dx, dx
    mov  dl, DriveNumber          ! take DriveNumber value with us
    jmpi 0,SECONDSEG              ! jmp  far 0:SECONDSEG
```

Boot code Tutorial Question

- `boot.S`

```
_SecondLoad:
    mov  ax, #'3
    push ax
    call minbios_WriteChar
    pop  ax

    mov  ax, #SECONDSEG
    mov  es, ax             ! ES = SECONDSEG
    mov  bx, #0x0     ! address = SECONDSEG:0
    xor      dx, dx                   ! dh (head no) = 0
    mov      dl, DriveNumber          ! drive no
    mov  cx, #0x02          ! sector 2, track 0
    mov  ax, #0x0200+SECONDSIZE    ! service 2, nr of sectors
                    ! (assume all on head 0, track 0)
    int  0x13           ! read it
    jnc  ok_found       ! ok - continue

    mov  ah, #'e
    push ax             ! display error message
    call minbios_WriteChar
    pop  ax
```

Boot code Tutorial Question

boot.S

```
        ! these three instrucions call a bios function which resets the boot device
    mov  dl, DriveNumber          ! drive number
    xor  ah, ah
    int  0x13
    jmp  _SecondLoad      ! try to load Secondary boot again

ok_found:
    ret
```