From Power-on to [/usr/root/]#

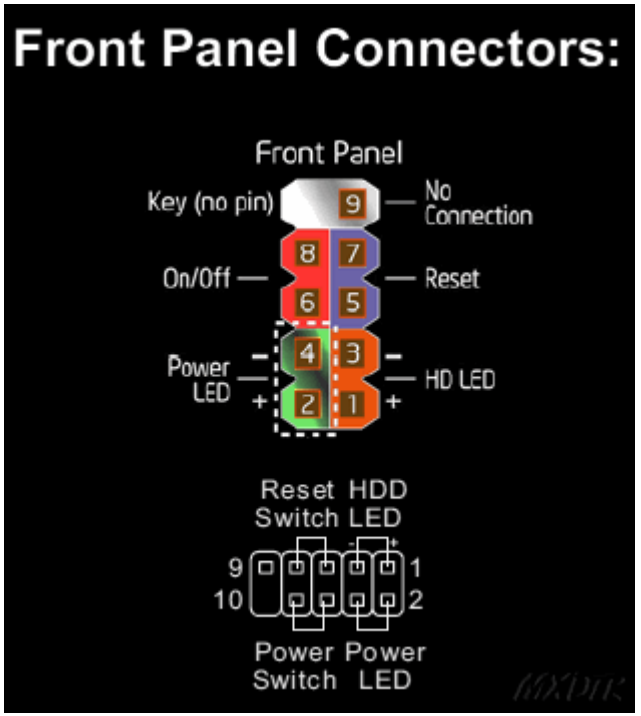
A brief guide to Linux 0.11 booting

System startup and booting process

- Hardware
  - Power supply
  - Motherboard
  - CPU
  - Storage device (hard drive or floppy)
- Software
  - BIOS and POST
  - Boot sector
  - Bootstrap (OS loader)
  - OS kernel initialization (Linux 0.11)

When you pressed the power button…

- It applies a ground to the green wire (ATX connector, pin 16) through motherboard.
- It kickstarts the PSU circuit (your AC power supply), then active the main power circuits.
- When motherboard is powered up, it initializes it own firmware like the chipsets and other.



When you pressed the power button…

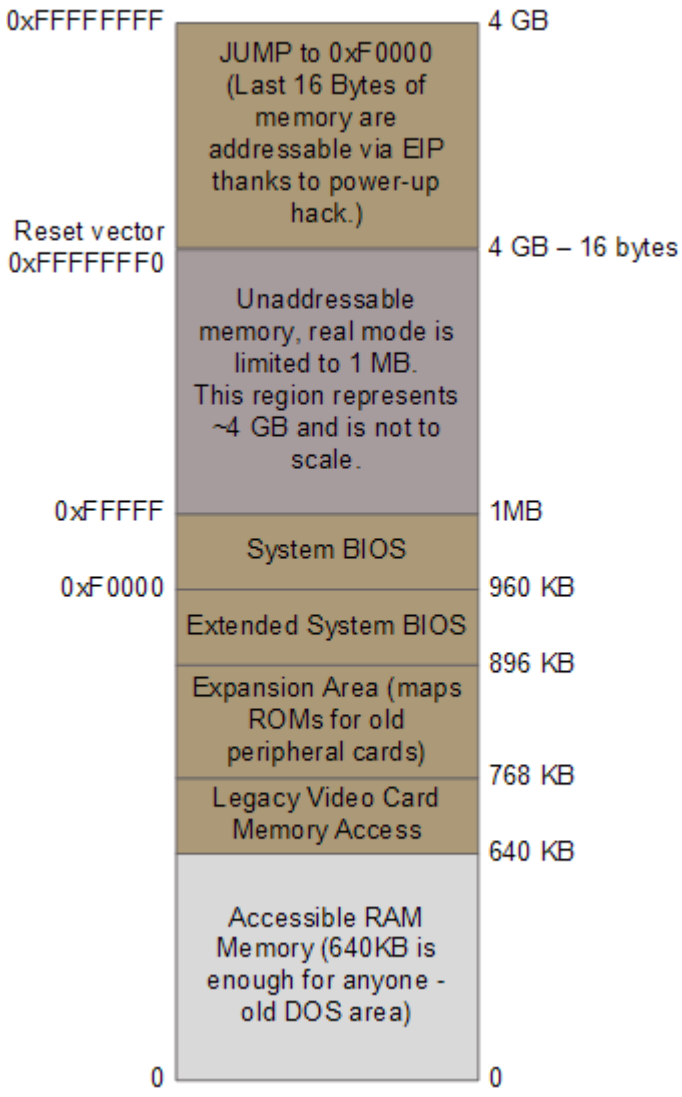
- The chipset generates a reset signal to CPU to prevent harm from unreliable power.
- As soon as all the power supply reach operating voltage, a “Power OK” (a.k.a “Power Good”) signal will raise. (Actually it’s a +5V signal on gray wire, pin 8 on ATX connector)
- This signal also removes the hardware reset signal from the CPU, so the CPU could initialize itself.

CPU Initialization

- When the processor starts up, there's nothing to execute in memory.
- CPU IP (Instruction Pointer, EIP in 32bit CPUs) is pre-programmed to always point to address 0xFFFFFFF0 (the end of the memory – 16bytes)
- This particular address is called "*reset vector*", this address only contains a "*jmp*" instruction. Motherboard ensures this instruction redirects to the memory location that holds BIOS entry point.

CPU Initialization

- The memory map which is held by the chipset knows address is pointing to the BIOS, so CPU would run instructions from BIOS flash memory instead of RAM.
- As you can see in this state the processor works under *real mode*, which means there's only 1MB of memory could be used.



CPU Initialization (a time travel)

- This initialization sequence is designed all the way back to Intel 8086 in 1978.
- All modern x86 CPUs behave the same way as 8086 due to the back compatibility.
- Except the *reset vector*, 8086/80286/80386 and it's processors have different physical memory address due to different addressing length.
  (8086/286 is 20bit while 386 and so on is more)

BIOS and POST

- While CPU is starting executing BIOS code, first it initialize some basic hardware in the system.
- Afterwards BIOS starts Power-on Self Test (POST) which tests various components in the computer.
- It tries to find video card first. In particular, POST tries to find the video card BIOS then run it to initialize video card. That's why a lot of computers shows video card information before system BIOS.

BIOS and POST

- If there's no video cards, POST beeps for alarm.
- Then POST checks other hardware like RAM\Hard drives\Floppy drives\CD-ROM… and show their information on screen.
- If there's any fatal errors (like keyboard is missing, there's no memory…) the system halts immediately.
- If there's something wrong but not too serious, POST might pause for warning, but the system still can boot by ignoring them manually.

Boot sector

- If POST runs successfully, the BIOS calls a special system interrupt *INT 19h* to start booting.
- BIOS selects a boot device according to the boot priority sequence, then copies the first sector from the first bootable device into physical memory address 0x7C00.
- This very first boot sector called MBR (Master Boot Record) or VBR (Volume Boot Record), depends on the storage device is partitioned or not. MBR invokes VBR from an active partition eventually.

Boot sector

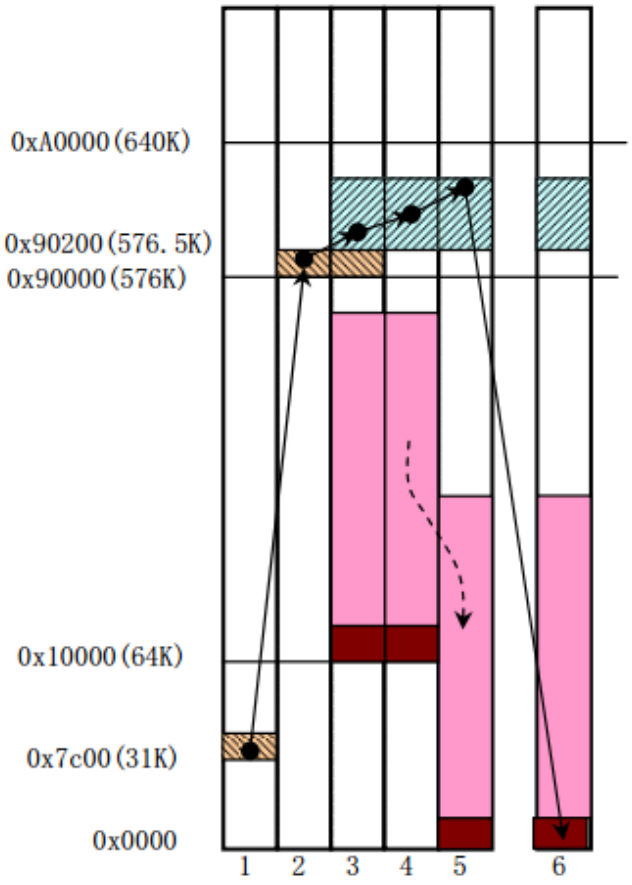
- On IBM-compatible PC (which means almost all modern PCs), there's also a signature for a boot sector. It always ends with last two bytes are *0x55 0xAA*. This helps BIOS recognizing whether this device is bootable or not.
- Once the boot sector is loaded in the RAM, BIOS tells CPU to run the code at address 0x7C00, hands the boot process to boot sector.

Linux 0.11 boot sector (bootsect.s)

- So the source code of boot sector in Linux 0.11 is placed under */boot/* in *bootsect.s*
- This piece of assembly code is written in as86 format (*it's different from GNU assembly!)*
- It mainly does these following things:
  - 1. Move itself from 0x7C00 to 0x90000
  - 2. Use BIOS interrupt INT 13h to load system initialization routine (*setup.s*) at 0x90200 and compiled system module (the kernel) at 0x10000
  - 3. Check which root-device to use & invoke *setup.s*.

Linux 0.11 boot sector (bootsect.s)

- From above we could know that actually *bootsect.s* works nothing more like a loader.
- It's worth mentioning that as Linus writing 0.11 kernel, he thoughts the size of kernel will never be over 512KB. That's why he decided to move *bootsect.s* at 0x90000 and put kernel at 0x10000.

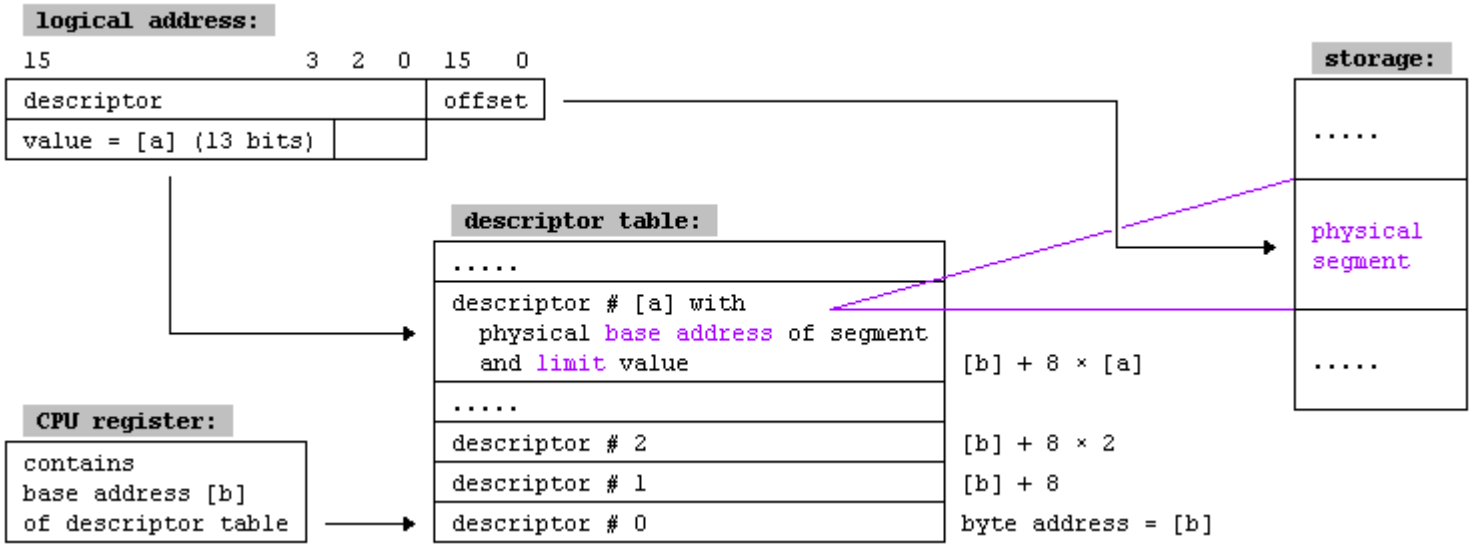


Linux 0.11 OS loader (setup.s)

- This routine is also placed under */boot/*
- It's loaded by *bootsect.s* at 0x90200 (exactly 512bytes after 0x90000, which is the length of boot sector)
- This code is responsible for getting system data from BIOS (like memory/disk/others) and putting them to a "safe" place (0x90000 actually, where *bootsect.s* used to stay).
- It also tries to get CPU into protected mode and invoke the kernel.

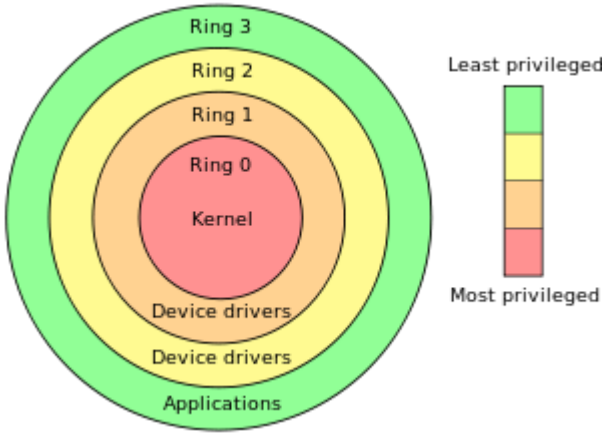
Protected mode

- While in *real mode*, CPU could only use 1MB of memory, in *protected mode* there's no limit thanks to the segment addressing. Additionally it also provides *paging* feature so that OS could manage memory more efficiently.



Protected mode

- *Protected mode* also introduces the *privilege levels* (from Ring 0 to Ring 3), allows for OS to restrict tasks from accessing data, call gates or executing privileged instructions.
- In Linux 0.11 and so on, the kernel and some of the device drivers run in Ring 0,
  other applications run in Ring 3.



Into protected mode

- Setup.s does the following things in order to get CPU into protected mode:
  - 1. Move the whole kernel from 0x10000 to 0x00000
  - 2. Load segment descriptors
  - 3. Enable A20
  - 4. Reprogramming the interrupt vector table
  - 5. Enable protected mode, reload machine status word (CR0 register)

Move kernel from 0x10000 to 0x0

- This part of code does almost exactly the same thing as the bootsect.s, or even simpler since the source is already in RAM, not on a storage device.
- It's the first step of enabling the protected mode. From now on, no interrupt is allowed while executing until the OS is loaded.
- *(line 108~127)*

Preparing segment descriptors

- 80X86 protected mode provides *segment* mechanism for memory management.
- In order to do such a thing, it introduces "*descriptor tables*" to hold basic information for different segments like max length, base address, privilege level, read/write access control, etc.
- It also brings in some new registers, *GDTR\LDTR\IDTR\TR*, respectively corresponds to Global\Local\Interrupt\Task, which holds the address to the actual *descriptor tables*.

Preparing segment descriptors

- As we all know there's almost nothing in our memory when we started up the computer, so we have to set descriptor tables (GDT and IDT in particular) and their corresponding registers manually in order to get into protected mode.
- In setup.s, Linus decided to set up GDT\IDT temporarily, fill GDTR\IDTR with correct address and hand them to the actual OS initialization process to deal with.

Preparing segment descriptors

- More precisely, the GDT is set to hold code and data descriptor for the current kernel (at 0x0).
- IDT is simply set to an empty placeholder, since no interrupt is allowed while setup.s is executing.
- That's enough for now to get CPU into 32bit protected mode, the OS initialization routine (*head.s*) later will set up GDT and IDT again for kernel.

  (line 205~244, loads at line 128~135)

A20 line and its history

- Back to the 8086/80186 era, there's only 20 address lines (A0~A19) in CPU, so the processor can access 1MB memory at total. As for the address above 1MB (like 0x100000), the bits higher than the 20th bit is been cut, so actually the memory address is "wrapped around" to 1MB.
- When IBM designed IBM-PC, they decided to use more powered 80286 as the CPU, it has the protected mode that could addressing more than 1MB, it also has the real mode for back compatibility.

A20 line and its history

- Sadly 80286 has a bug where it failed to force A20 line to zero in real mode. This causes this CPU cannot do the “wrap around” trick like the original 8086. Some old programs might no longer work.
- For compatibility’s sake, IBM decided to fix the problem on the motherboard. This was done by inserting a logic gate on the A20 line between the processor and system bus. This gate can be enabled or disabled by software to allow or prevent the address bus from receiving a signal from A20.

A20 line and its history

- The original Gate-A20 is connected to the Intel 8042 keyboard controller (simply because there are spare pins in it). But it operates slow, so there are several other methods to do it. Finally the industry settled on the PS/2 method of using a bit in *PORT 92h* control the A20 line.
- 80286's processors use different techniques to simulate Gate-A20's behavior, until recently in Haswell microarchitecture, Intel finally decides to remove A20 support since it's "not used by modern operation systems".

A20 line controlling in setup.s

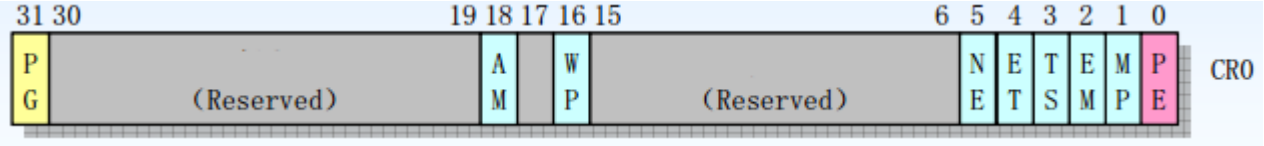
- From above we know that it's an important step to set A20 enabled in order to get into protected mode.
  (Actually in POST routine, BIOS would enable A20 for testing all system memory, but disabled it again after testing.)
- In setup.s, Linus uses the classic keyboard controller method to operate Gate-A20.
  (line 137~144)

Reprogramming Interrupts

- In setup.s, Linus complains about how IBM original PC BIOS messed up the interrupts by putting them at *INT 0x08-0x0f*, which is still in Intel-reserved hardware interrupts range.
- So he has to reprogram the 8259 PICs and put those interrupts at *INT 0x20-0x2f*, right after the hardware interrupts range.
- This part of code is been labeled “not fun” by Linus and he also pokes at BIOS routine being “wants lots of unnecessary data, and less ‘interesting’ ”. *(line 154~180)*

Enable protected mode

- With all above works done, finally we're about to enable the protected mode.
- 80x86's has several *control register* (CR0~CR4), among these, CR0 register is the one that controls the operation type of CPU. The lowest bit (*PE bit*) defines whether the CPU is working under real mode or protected mode.
- The highest bit (*PG bit*) of CR0 is paging bit, defines the paging mechanism is enabled or not.



Enable protected mode

- Now, all we have to do is to set the PG bit in CR0 as 1 (enable), then refresh the instruction queue.
- CPU reads instruction before executing them, so the address in old instruction queue is still under real mode (which is wrong by now). Refreshing is accomplished by executing “jmp” instruction.
- Since we’ve moved kernel to 0x0, the “jmp” instruction jumps to 0x0 directly to invoke the OS kernel initialization process. *(line 191~193)*

Linux 0.11 OS Initialization (head.s/main.c)

- Although *head.s* is placed under */boot/*, but it's compiled with *main.c* (under */init/*) and all other system modules, forming a complete executable kernel.
- Apparently *head.s* is placed at the start of the compiled kernel, it's also responsible for invoking *void main(void)* from *main.c*.
- head.s is written in normal GNU assembly language, different from *bootsect.s* and *setup.s*. It runs under 32bit protected mode.

head.s

- This initialization routine is rather simple, basically it's just a bunch of tests and settings.
- It does these following things:
  - 1. Resets all the data segment registers (*DS, ES ,FS, GS* and stack register *SS, SP*)
  - 2. Setup the new GDT and IDT
  - 3. Check whether A20 is truly enabled
  - 4. Test if there's any co-processors available
  - 5. Enable paging (CR0 31st bit)
  - 6. Invoke *void main(void)* from *main.c*

Reset data segment registers

- Since we're in protected mode, we need "*segment selectors*" to specify a segment.
- CPU provides us 6 registers, *CS, SS, DS, ES, FS, GS*, used for different selection. CS is used for code segment, SS is used for stack segment, DS-GS is all used for data segment.
- So the first thing that *head.s* do is initializing these registers, DS-GS is set with the data descriptor that defined in *setup.s*. SS is set with *_stack_start*, which is defined in */kernel/sched.c*, used as the start of user stack. CS is reloaded with GDT.

Set up the new GDT

- There's no big difference between the new GDT and old, expect the segment length is adjusted to 16MB instead of 8MB.
- But the thing that worth mentioning is after loading new GDT, usually all the respecting segment registers should be reloaded too.
  In head.s though, Linus doesn't reloaded CS register because 8MB segment is enough for kernel initialization, and there'll be another *jmp* instruction later for reloading CS register later.
  (This problem has been solved in later kernel versions, by inserting a *ljmp* instruction)

Set up the new IDT

- At this stage, all 256 entries in IDT points to *ignore_int*, which is a simple placeholder that does nothing. It's safe to do such a thing because we're still not allowing any interrupts.
  The real interrupt gates will be loaded and enabled later.
- The new GDT/IDT is located at the end of head.s.

(line 18~31)

Check the status of A20 line

- This is accomplished simply by writing any number to address 0x0, then check 0x100000 whether holds the same number.
- If they're same, that means the A20 line is not enabled, kernel can't use memory above 1MB. System will be trapped in a forever loop.

(line 32~36)

Testing co-processors

- The "*co-processor*" here are mainly describing Intel 80287/387 floating point coprocessors. But all x87 floating point instruction are implemented in main CPU since 80486, it's rather unnecessary to check these co-processors nowadays.

(line 37~66)

Setup paging in Linux 0.11

- Setting up paging scheme is the last thing that head.s do before invoking *main(void)* from *main.c*.
- So firstly we have to push the address of main function into system stack that we could jump to it directly when returning from "*setup_paging*" subroutine.

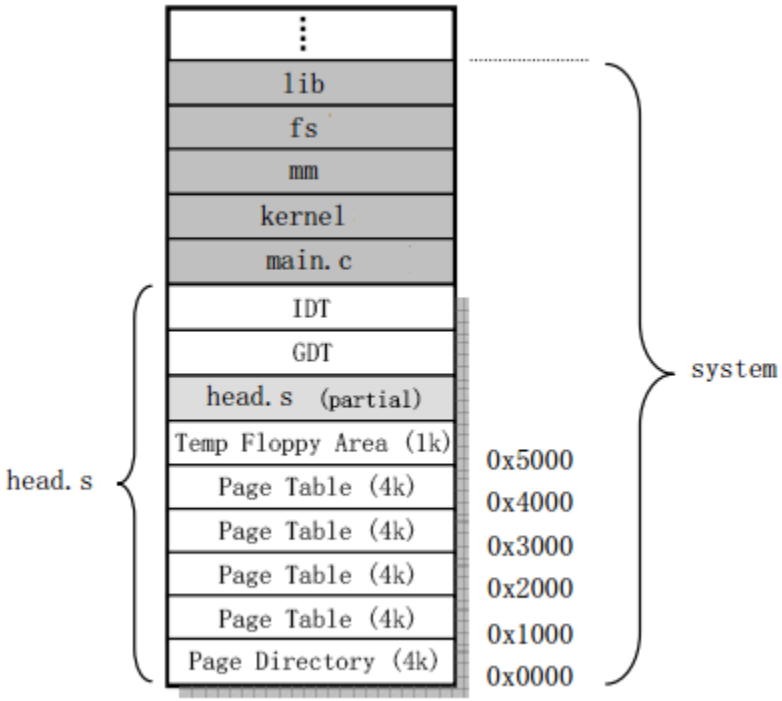
Setup paging in Linux 0.11

- The paging scheme in Linux 0.11 kernel is simple, 1 page directory at the top, 4 page tables, 1024 entries per table, 4KB per page, handles 16MB memory in total (could be larger if you hack the this part of code)
- The page directory and page tables are located at 0x0, the start of *head.s*, since the old routines that used to be there is useless by now.
- Once the page directory and page tables are set, we set the value of CR3 register which holds the base address to page directory. Finally, set the 31st bit (PG bit) of CR0 register to 1 to enable paging scheme.

Setup paging in Linux 0.11

- Like other changes that we did to GDT before, jump instruction is needed to refresh the instruction queue.
- Here in *head.s*, we simply use a "*ret*" instruction, it'll directly jump at the start of *main(void)* according to the stack we've pushed in before.
- This is a also a basic trick to invoke C function in assembly language.
  (line 135~218)

The memory looks like this while *main(void)* is being invoked…



main.c

- Here we are finally! The good old C code!
- *main.c* is the final part of the whole OS initialization process.
- First, it uses information that *setup.s* gathered to initialize the memory cache, RAMDISK (if defined) and main memory.
- Second, the kernel initialize all the basic hardware including interrupt vectors, block device, character device, tty… It also reads time from CMOS, initialize the scheduling program, buffer management, HDD and floppy, etc. When all hardware is initialized, the kernel enables interrupt. *(line 55~136)*

Task 0 Idle

- Now the system is all initialized, kernel move itself to user-mode and start creating processes (tasks).
- The very first two tasks that OS launches are special. Task 0 is the “idle” task.

for (;;) pause();

- All it does is keep invoking *pause()*, make itself interruptible by any other tasks.
  But the scheduling module always switches back to Task 0 while there’s no other tasks to do, regardless of its status.

(line 137~149)

Task 1 init

- Task 1, created by Task 0 by invoking system call *fork()*, is responsible for mounting the root file system, open the terminal device (*/dev/tty0*) and *stdout/stderr*, and creates Task 2 to make *sh* non-interactively run shell script from */etc/rc* (Similar to *AUTOEXEC.BAT* under MS-DOS)
- If all these going well, finally Task 1 creates another task to run "*-/bin/sh*", the hyphen tells *sh* that this is a interactive shell environment.
  At last, *sh* takes over the computer, shows the prompt, and waits input from user. *(line 168~210)*

Voilà!

Your computer has booted successfully!

But wait, we're not done yet. ☹

Why are they special?

- In Linux, creating new process is done by copying from the parent process first, and all other processes (including Task 1) are created by Task 0.
- As we know, every process has a two stacks, kernel stack and user stack. Since kernel stack is independent between processes, it's important to keep "clean" state of user stack, especially for Task 0.
- In fact, Task 0 and Task 1 is shared the same code and data segment from kernel, which means they share the same user stack.

Why are they special?

- So in *main.c*, all system call like *fork()/pause()* is inlined into the code, we could use them without polluting the user stack of Task 0.
- As for Task 1, it shares user stack with Task 0, but its page table entry is been set “read-only”. Once Task 1 executes a stack operation, it triggers a “*write-protect fault*”, then the memory management module allocates a new page for Task 1’s own user stack.
  (This technique is called “copy-on-write”)

Now we're done!
Thanks!

☺

Compiled by Light Catcher

Dec, 23, 2015

References

- http://duartes.org/gustavo/blog/post/how-computers-boot-up/
- http://www.pcguide.com/ref/mbsys/bios/bootSequence-c.html
- http://www.tomshardware.com/forum/273879-28-what-sequence-pushing-power-button-startup
- Wikipedia
- A Heavily Commented Linux Kernel Source Code –Linux version 0.11, Rev 3.0, Chapter 6\7