Zavedení a struktura operačního systému.

Wednesday, May 29, 2013 4:49 PM

Zavedení OS

BIOS

- program přítomný ve vestavěné paměti HW (většinou na základní desce)
- provádí testy a nastavení HW
- vybere zaváděcí jednotku
- načte první sektor (MBR), kde je umístěn program zavaděče a provede skok na adresu jeho programu, čímž mu předá řízení

Zavaděč (bootstrap program)

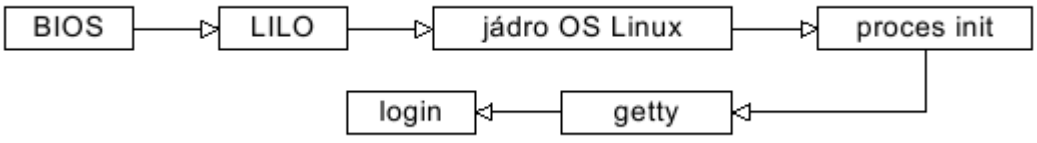
- pro Linux LILO (LInux LOader) je všeobecně použitelný zavaděč (boot loader) pro Linux) nebo GRUB
- dává možnost zvolit startující OS
- načte jádro operačního systému do paměti a spustí ho

Jádro OS

- detekuje hardware a odpovídajícím způsobem nastaví ovladače zařízení
- připojí kořenový svazek pro čtení a provede kontrolu souborového systému
- spustí na pozadí proces init

Proces init

- proces init se konfiguruje pomocí souboru /etc/inittab
- inicializuje operační systém
- spuštěn po celou dobu běhu operačního systému a ošetřuje některé události (úklid v adresáři /tmp)
- spustí služby - Démony
- nakonec spustí program getty pro terminály a virtuální konzoli a v ní program *login*
- nastavením parametru jádra tzn. runlevel lze upravit chování systému



Úrovně běhu systému – runlevel

- runlevel 0 – zastavení systému - halt
- runlevel 1 – jednouživatelský režim Single user mode
- runlevel 2 – víceuživatelský režim bez podpory sítě Multiuser, without NFS
- runlevel 3 – víceuživatelský režim s podporou sítě Full multiuser mode
- runlevel 4 – není použit unused
- runlevel 5 – víceuživatelský režim s podporou sítě a XFree X11
- runlevel 6 – restart systému reboot

Zavádění systému

- nejprve proběhne úspěšný test zavádění systému – POST
  - kontroluje HW v zařízení
  - série testů ke zjištění, zda HW pracuje správně
  - zjištěné chyby jsou uloženy nebo oznámeny – blikáním LED/série pípnutí
  - Po dokončení je řízení předáno bootovací sekvence volající ovládací SW nebo zavaděč OS

- **Bootování**
  - Najde a zavede (vygeneruje se přerušení 19h) se tzv. Bootovací sector (boot sector)
  - **Boot sector -** oblast 512 bajtů na záznamovém médiu, které je jako první nastavené v paměti BIOSu
    - Bootsektor se nachází na prvním sektoru záznamového média (v případě pevných disků je to válec 0 hlava 0 stopa 0 sektor 1)
  - BIOS se snaží najít na tomto sektoru Master Boot Record (MBR) – hlavní spouštěcí záznam.
    - Ten nahraje do paměti na adresu 0000:7C00 a v případě úspěchu mu předá řízení.
    - V případě chybného MBR se bootovací proces přeruší pomocí softwarového přerušení 18h – vygeneruje chybové hlášení (např. NO ROM BASIC – SYSTEM HALTED)
    - Správnost MBR BIOS zjišťuje pomocí kontrolní hodnoty umístěné na posledních dvou bajtech sektoru - **AA55h** (zápis je uložen ve formátu little endian)
    - MBR – ze dvou částí – **partition loader** a **partition tabulky**
    - MBR uchovává záznamy o rozdělení disku (oddílech) a určuje, ze kterého z nich se má bootovat.
    - Pokud MBR OK – řízení se předá partition loaderu
    - **Partition loader** v Partition tabulce vyhledá oddíl, který je označen jako aktivní a přejde na první sektor tohoto oddílu. MBR sám sebe překopíruje na jiné místo v paměti a na své původní místo zkopíruje tento první sektor a předá mu řízení

-

Spuštění počítače

- Po zapnutí PC jsou všechny procesory v reálném režimu
- - Náhodně se vybere jedno jádro -> bootstrap processor (BSP)
- - ostatní CPU jsou nyní application processors AP - pozastavené, dokud je nezapne kernel
- - nyní je BSP v tzv. real mode, vypnuté stránkování (BSP simuluje staré 8086 ze let okolo '78)
- - v tomto stavu je adresováno pouze 1MB paměti bez ochrany (lze v ní spustit cokoliv)
- - u Intel CPU je hack, kdy se nastaví bázová adresa (jako offset) na tzv. reset vector – 0xFFFFFFF0 (konec 4GB paměti - 16B)
- - na adrese reset vectoru je jump na adresu, kde je namapovaný BIOS entry point (zajišťuje základní deska)
- - tento skok vymaže Intelí hack bázovou adresu
- - oblasti v paměti jsou zaplněna správnými daty díky memory map v chipsetu
- - nyní BSP spustí BIOS -> Power-On self test (POST) -> error = pípání PC speakeru nebo zombie PC
- - po POST bootování systému, umístění volitelné (disketa, DVD ROM, HDD, ...)
- - BIOS nyní přečte první sektor umístění (HDD) o velikosti 512B (zero sector) = Master Boot Record (MBR)
- - MBR obsahuje:
- 1) malý zavaděč na začátku MBR specifický pro OS
- 2) tabulka partition
- - obsah MBR je načten do adresy 0x7c00 a skočí na začátek kódu v MBR (zavaděč)

- - bootujeme

Princip
o Po zapnutí jsou všechny procesory v reálném režimu
o BIOS vybere BSP a ostatní procesory zastaví
o Kód SMP jádra běžící na BSP prohledá paměť na _MP_
o Pokud nenašel, zavede se jednoprocesorové jádro
o Pokud našel, inicializuje APIC BSP
• K tomu je nutné se přepnout do chráněného režimu
o Kód vykonávaný BSP postupně vzbudí AP pomocí Init-IPI (Inter-Processor Interrupt)
o AP se přepne do chráněného režimu a začne svoji další činnost synchronizovat s kódem, který ho spustil a běží na BSP
o Jakmile jsou inicializovány všechny AP, BSP přepne I/O APIC do symetrického IO režimu
• Routovací tabulka, která přesměruje přerušení od sběrnic periferií na některý lokální APIC AP
o SMP jádro pokračuje dál s vlastní inicializací
Viz přednáška PPR d_multithreading.pdf.

http://www.cheesecake.org/sac/smp.html

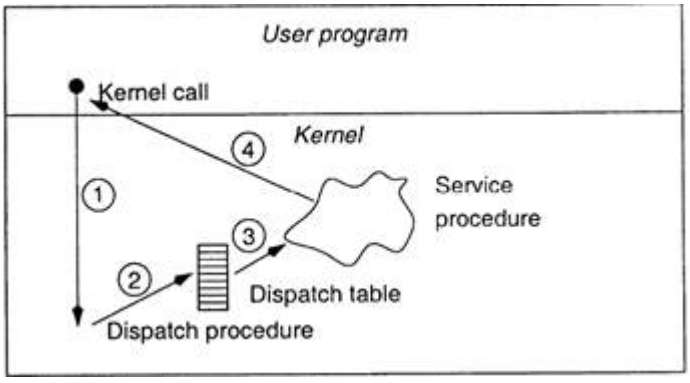
Struktura OS

Monolitické systémy

- pro jednotlivé funkce jsou definovány moduly
- modul může volat jakýkoli jiný modul
- všechny moduly jsou spojeny do vykonatelného souboru s operačním systémem

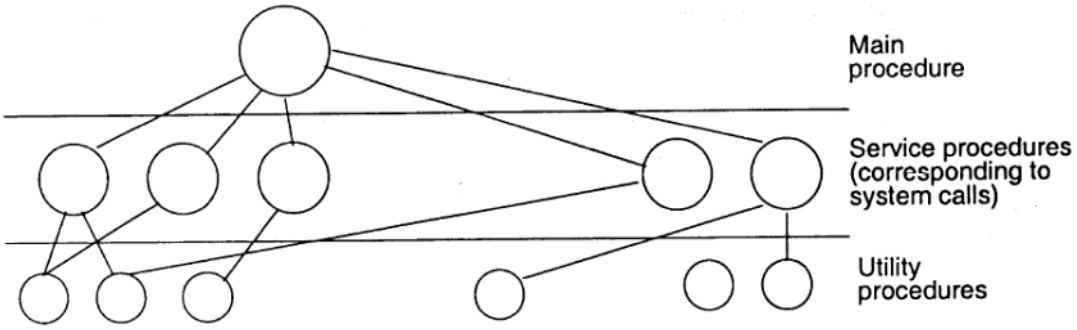
Systémové volání (služba jádra)

- volání vstupního bodu jádra OS s přepnutím do privilegovaného režimu
- zjištění čísla požadované služby
- volání obslužné procedury
- návrat s přepnutím do neprivilegovaného režimu



Model struktury monolitického systému

- hlavní program, který spouští obslužnou proceduru
- množina obslužných procedur pro systémová volání
- podpůrné procedury pro vykonání obslužných procedur
-



-
- mají tendenci extrémně narůstat
- Monolit akumuluje moduly, které by potenciálně mohli být potřebné
- Těžce se ladí

Např. Linux, Windows

Systémy založené na mikrojádře

- **Vrstvené systémy**
  - Hierarchie vrstev poskytujících služby
  - Programy vyšší vrstvy využívají služeb nižších vrstev
  - Holý počítač je nejnižší vrstva
  - Aplikační program je nejvyšší vrstva
  - Princip vrstev umožňuje systematickou tvorbu programů a jejich testování
  - Princip vrstev je možné použít pro monolitický model i pro systémy

- **Mikrojádro**
  - Vrstva nad holým strojem, která obsahuje minimální množinu abstrakcí, tak aby ostatní funkce OS mohly být implementovány nad ním
  - Tyto funkce OS nemusí být vykonávány v privilegovaném režimu
  - Jenom mikrojádro musí být vykonáváno v privilegovaném režimu
  - Typická množina abstrakcí implementována mikrojádrem:
    - Přerušení
    - Vlákna
    - Správa paměti
    - Meziprocesová komunikace
    - Procesy
  - Ostatní funkce – soubory, adresáře, síťové služby – jsou programy vykonávané v uživatelském režimu

základní rozdělení

- monolitické systémy - hlavní program, obslužné procedury, podpůrné procedury
- vrstvené systémy - hierarchie vrstev, nejnižší je holý počítač, nejvyšší je aplikační program

funkční hierarchie - někdy je problém rozdělit do vrstev podle úrovně abstrakce, proto dělení do vrstev podle funkčnosti

- klient-server - obsahuje mikrojádro, které poskytuje pouze základní funkce, většinu práce dělají servery, které jsou oddělené od jádra
- objektově orientovaná struktura - jádro spravuje řadu objektů (zastupují soubory, HW zařízení, ...), mezi objekty jsou tzv. capability = odkaz na objekt + množina práv definujících operace

http://wiki.zvesela.cz/index.php/Funkce_opera%C4%8Dn%C3%ADho_syst%C3%A9mu%2C_struktura_a_rozhran%C3%AD_opera%C4%8Dn%C3%ADho_syst%C3%A9mu%2C_mikroj%C3%A1dro.

Preface

This tutorial is intended as a supplement to the SigOps OS Tutorial to teach the fundamentals of symmetric multprocessing using Intel MP compliant hardware. Knowledge of the concepts and implementations of basic operating system parts such as managing virtual memory and multitasking are assumed and will not be discussed except as they relate to multiprocessing. Knowledge equivalent to an intermediate or advanced computer architecture college course will be helpful in understanding scheduling issues, but is not required.
This tutorial is not intended to be a complete explanation of how to implement an SMP-capable operating system, nor as a replacement for Intel's documentation. Rather it is designed to give an overview of the things I learned in writing SMP support for OpenBLT, a freely redistributable microkernel-based operating system under the BSD licence. Particularly, some tedious hardware aspects will not be discussed in detail when the reader could just as easily read official Intel documentation. The interested reader should refer to the references for more detailed information. For code examples, the reader should refer to the source code of OpenBLT or FreeBSD. The Linux kernel source code might be helpful, although it is under the GPL.
This tutorial is a work in progress. If you see an error or something that needs clarification, please e-mail me.

Terminology

AP
: application processor. A processor that is not the BSP. All APs are in a halted state when the BIOS first gives control to the operating system.

APIC
: Advanced Programmable Interrupt Controller. Either a local APIC or an I/O APIC. It is attached to the APIC bus.

APIC bus
: A special non-architectural bus on which the APICs in the system send messages.

BSP
: bootstrap processor. The processor which is given control after the BIOS finishes its POST.

I/O APIC
: A special APIC for receiving and distributing interrupts from external devices which is backward compatible with the PIC. There is generally only one per computer.

IPI
: interprocessor interrupt. A special interrupt sent to a processor by the originating processor programming its APIC with a target or logical target ID, and an interrupt vector.

Local APIC
: an APIC built in to the processor. It is responsible for dispatching interrupts sent over the APIC bus to its processor core and sending interrupts to other processors over the APIC bus.

MP
: Intel's MultiProcessor Specification, a standard which defines how SMP hardware should be presented to the operating system and how the operating system should interact with this hardware.

serialisation
: The act of executing a certain instruction which causes the processor to pause to retire all instructions currently being executed before proceeding to the next instruction in the stream. For example, before switching to protected mode, the processor must retire all instructions that began executing in real mode before beginning any in protected mode.

SMP
: symmetric multiprocessing. Using multiple processors which share the same physical memory in the same computer at the time. You are probably reading this tutorial with the hope that your operating system will become SMP-capable.

UP
: uniprocessor. Your operating system to date is a UP operating system.

MP Detection and Configuration

When the system first starts, the BIOS detects the hardware installed in the system using electric means and then creates structures to describe this hardware to the operating system. There are two such tables. The first is the MP Floating Pointer Structure, which is required. The second is the MP Configuration Table, which is optional. If the configuration table does not exist, the operating system should set up the default configuration indicated in the floating pointer structure. Some data in the tables is in ASCII. Strings are padded with spaces and are not null-terminated.

First, you need to find the floating pointer structure. According to the spec, it can be in one of four places: (1) in the first kilobyte of the extended BIOS data area, (2) the last kilobyte of base memory, (3) the top of physical memory, or (4) the BIOS read-only memory space between 0xe0000 and 0xfffff. You need to search these areas for the four-byte signature "_MP_" which denotes the start of the floating pointer structure. Absence of this structure indicates that the system is not MP compliant. At this point your operating system can either halt, or it can fall back into a UP setup.

You should checksum the structure to make sure it has not been corrupted. There is not much of interest in the floating pointer structure, unless your system does not have a configuration table. In this case, you will need to get the number of the default configuration your system adheres to and set up the system for SMP using those parameters. Otherwise, you will need to get the address of the configuration table and begin parsing that.

The configuration table is divided into three parts: a header, a base section, and an extended section. The header begins with the four-byte signature "PCMP", although you do not have to search for it. Once you find it, checksum it. At this point, you can print the OEM and product ID strings in the configuration table if you want. You should get the address of the local APIC from this and store it. Then, proceed to parse the base section.

The base section consists of a set of entries that describe either processors, system busses, I/O APICs, I/O interrupt assignments, or local interrupt assignments. All entries are eight bytes in length,

save processor entries which are twenty bytes. The first byte of each entry denotes the type of the entry. Look through each entry. You will probably want to generate quite a few OS-specific data structures here. In particular, you will want to note the APIC ID of each processor in the system, its version, and its type as well as the address of the system's I/O APIC.

Using Local APICs

MP systems have a special bus to which all APICs in the system are connected. This bus is one of the ways the processors can communicate with one another (the other, of course, is shared memory). APICs (both local and I/O) are memory mapped devices. The default location for the local APIC is at 0xfee00000 in physical memory. The local APIC will appear in the same place for each processor, but each processor will reference its own APIC; the APIC intercepts memory references to its registers, and those references will not generate bus cycles on some systems. Since APICs are mapped in high memory, the APs will have to switch to protected mode before they can intialise their local APICs. If you like, you can map the APIC to a different address using the paging unit, but be sure to disable caching in the page table entry since some registers can change between accesses. For this reason, pointers to APIC registers should be volatile. To initialise the BSP's local APIC, set the enable bit in the spurious interrupt vector register and set the error interrupt vector in the local vector table.

Booting Application Processors

Once you have detected the processors in the system, set up your local APIC, and verified that you can communicate with it (hint: read the APIC version register), it's time to boot the APs. N.B. that it is good practise to not try to boot the BSP here. That would be bad.

Since the APs will wake up in real mode, everything they need to get started should be in low memory (below 0x100000 or one megabyte). First, set the shutdown code by setting address 0:f to 0xa. Then, grab a page of memory for the AP's stack. You will also need space to store the `trampoline' code, i.e. the code the processor executes after waking up to switch to protected mode and jump to the kernel. You can either use the same page of code for each processor or store the code at the bottom of the processor's stack. Note that the start of the code must be at a page-aligned address. Copy the code there, then set the warm reset vector at address 40:67 to the start of this code. Next, you should reset a bit in the kernel which the processor will use to signal that it has booted and finished initialisation and clear any APIC error by writing a zero to the error status register. If you need to pass any parameters or data to the AP, now would be a good time to set that up. For example, since OpenBLT's kernel runs in high memory, I have to pass the address of the page directory in memory so that the AP can load it and enable paging before calling the kernel.

Now you can actually boot the processor. The procedure consists of sending a sequence of interrupts to the processor. The incremental effect of each is undefined, but at the end of the sequence, the processor will be booted. First send an INIT IPI. Assert the INIT signal by writing the target processor's APIC ID to the high word of the interrupt command register. Then write to the low word with the bits set to enable the INIT delivery mode, level triggered, and assert the interrupt. Deassert INIT by repeating the procedure with the assert bit reset. Now, wait 10 ms. Use of the APIC timer is suggested.

If the local APIC is not an 82489dx, you need to send two STARTUP IPIs. Clear APIC errors, set the target APIC ID in the ICR, then send the interrupt by writing to the low word of the ICR with bits set for STARTUP delivery mode and with the code vector in the low byte. The code vector is the physical page number at which the processor should start executing, i.e. the start of your trampoline code. Wait 200 ms, then check the low word of the ICR to make sure bit 12 is reset to indicate the interrupt was dispatched before sending the second STARTUP. After sending it, spin and wait for the AP to set its ready bit in memory. You may want to set a timeout of 5 seconds, after which you assume the processor did not wake up.

Switching from Real Mode to Protected Mode

Provided you did everything right above, the processor at some point woke up in real mode and started executing the code you told it to. First, execute a "cli" instruction to turn off interrupts, just in case. Now, begin the switch to protected mode. Load an appropriate value into GDTR. This can either point to the actual GDT or in my case, a temporary GDT. If you need to activate paging, load the address of a page directory into cr3. Then set bit zero in cr0 to enable protected mode as well as bit 31 if you need to enable paging to get into the kernel. Then do an ljmp to the kernel text segment with an offset that points to the next instruction to serialise the processor. Now that you're in protected mode, load appropriate descriptors into the segment registers, then execute a "cld",

which is reportedly what gcc expects. Then, jump to the starting address of your kernel.
Don't reference any symbols in this code since it will be running at an address for which it was not linked; all memory references must be absolute. Since your kernel is above one megabyte in memory, you can't access any global variables in real mode. Also be careful in specifying your offset address for the ljmp instruction, and do specify the address of the start of your kernel, not a symbol in the instruction that goes into the kernel. Jumping to a symbol doesn't seem to work. For details, see OpenBLT's kernel/trampoline.S.
Debugging this part is really not too bad. What you have to do is establish some communication space in low memory, then have the AP write bytes to that memory to explain what it is doing and print these out on the BSP.

Interprocessor Interrupts

IPIs are used to maintain synchronisation between the processors. For example, if a kernel page table entry changes, both processors must either flush their TLBs or invalidate that particular page table entry. Whichever processor changed the mapping knows to do this automatically, but the other processor does not; therefore, the processor which changed the mapping must send an IPI to the other processor to tell it to flush its TLB or invalidate the page table entry.
Using the local APIC, you can send interrupts to all processors, all processors but the one sending the interrupt, or a specific processor or logical address as well as self-interrupts. To send an IPI, write the destination APIC ID, if needed, into the high word of the ICR, then write the low word of ICR with the destination shorthand and interrupt vector set to send the IPI. Be sure to wrap these functions in spinlocks. You might want to turn off interrupts as well while sending IPIs.

Other Considerations

One thing to note is that semaphores (a.k.a. spinlocks) may need to be done differently under SMP. Consider a scenario where semaphores are procured with a ``bts'' instruction. If both processors hit that instruction at the same time while the semaphore is reset, they might both think they have acquired it. For this reason, you would need to use a ``lock'' prefix on that instruction to lock the system bus and maintain synchronisation.

From <http://www.cheesecake.org/sac/smp.html>

From <https://d.docs.live.net/e3534876709763a3/Dokumenty/ZCU/Statnice/Statnice.docx>