Proces

- program = vykonatelný soubor
- proces = jedna instance vykonávaného programu

UNIX

- souběžně (simultaneously) se může vykonávat mnoho procesů (šachový velmistr)
- může se vykonávat mnoho instancí jednoho programu (např. pogramu **kp** pro kopírování souborů)

Proces v UNIXu

- proces je jednotka (entita), která vykonává programy a poskytuje prostředí pro jejich vykonávání
- adresový prostor + počítadlo instrukcí
- proces je základní jednotkou plánování (*scheduling*)
- procesor vykonává v jednom okamžiku nejvíc jeden proces
- soutěží a vlastní prostředky
- požadují vykonání služeb jádra

Systémová volání pro procesy

- vytvoření procesu

```
pid = fork();
```

vytvoří se (téměř) identická kopie volajícího procesu

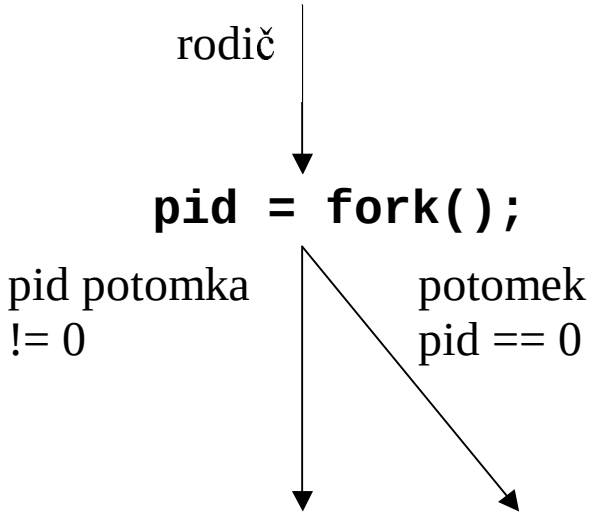
- adresový prostor je kopie adresového prostoru volajícího programu a vykonává se stejný program
- vytvořený proces má svou kopii deskriptorů souborů, které odkazují na stejné soubory
- volající proces rodič
- vytvořený proces potomek
- každý proces (kromě prvního má svého rodiče)
- rodič může mít více potomků
- návrat ze systémového volání (fork) na stejné místo

- jak je rozeznáme ?

- jádro identifikuje procesy číslem procesu, které se nazývá identifikátor procesu (*process identifier* PID)
- návratová hodnota **`pid`** bude ve volajícím procesu PID vytvořeného potomka a v potomkovi bude nula
- program může obsahovat kód rodiče i potomka

```
main()
{
   /*kód rodiče*/
   pid=fork();
   if (!pid)
   {
      /*kód potomka*/
      ...
```

```
    }
    if (pid)
    {
    /*kód rodiče*/
    ...
    }
}
```



- častěji, v nově vytvořeném procesu se vykoná nový program voláním některého tvaru služby **exec**

  kp – název souboru, který obsahuje vykonatelný program pro kopírovaní souborů

```
main(int argc, char *argv[])
{
   int stav;

   if (fork == 0)
          execl("kp", "kp", argv[1],
             argv[2], 0);
   wait(&stav);
   printf("kopirovani skonceno");
}
```

původní program je v paměti přepsán a potomek nepokračuje vykonáváním starého programu, ale
potomek se vrátí z volání s počítadlem instrukcí nastaveným na první vykonatelnou instrukci nového
programu

- čekání na skončení potomka

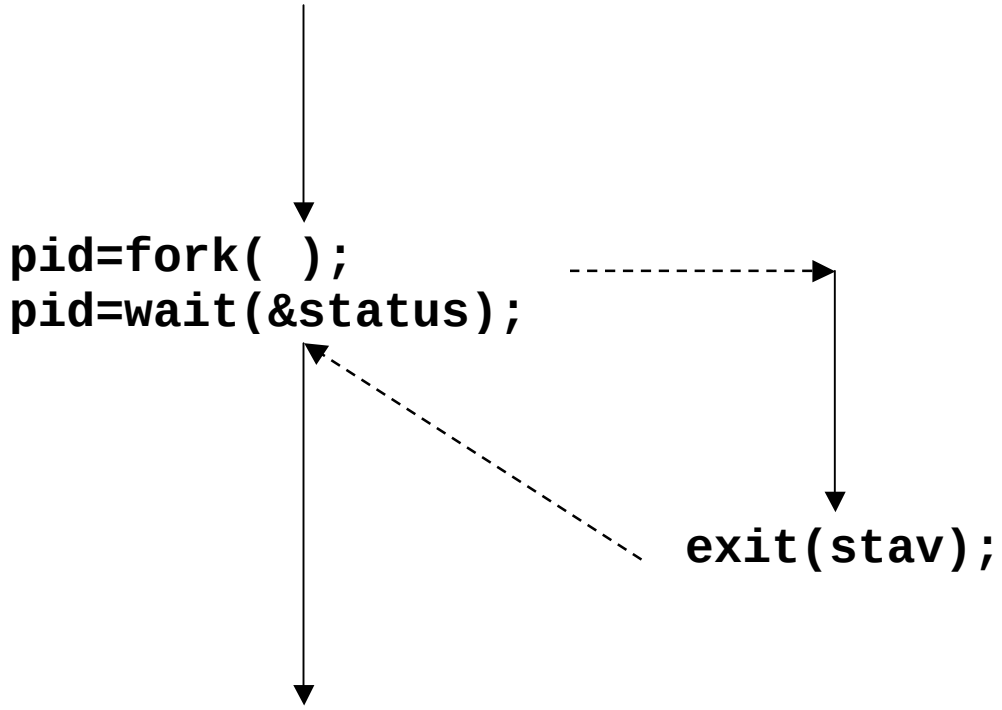
  pid = wait (stav_adresa);

  stav_adresa je adresa celočíselné proměnné, která bude obsahovat koncový stav procesu

- ukončení procesu

  exit(stav);

  C programy volají **exit** při návratu z funkce **main**



proč jsou na vykonání nového procesu nutná dvě systémová volání a tedy dvojité náklady?

- v klient-server aplikacích program server může vytvořit voláním **`fork`** více procesů pro obsluhu klientů (v moderních systémech více vláken)
- možno v procesu vyvolat vykonání programu bez vytvoření nového procesu
- mezi **`fork`** a **`exec`** může potomek vykonat vhodné akce ještě dřív než je vyvolán nový program

Meziprocesová komunikace

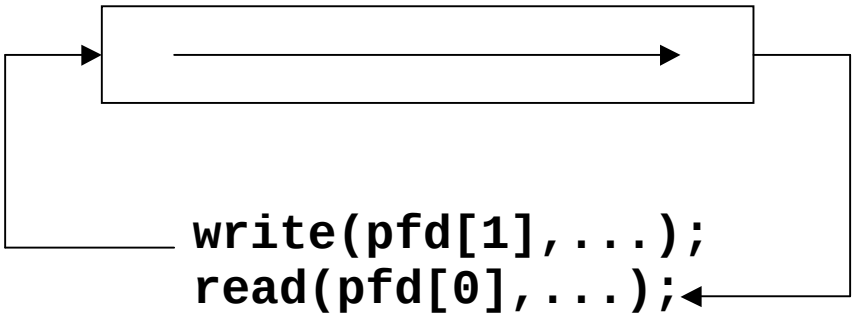
- přenos dat mezi procesy umožňují roury

- vytvoření roury

```
pipe(fdptr);
```

fdptr pole dvou deskriptorů pro zápis do a čtení z roury

```
int pfd[2];
...
pipe(pfd);
...
   write(pfd[1],...);
   read(pfd[0],...);
...
```

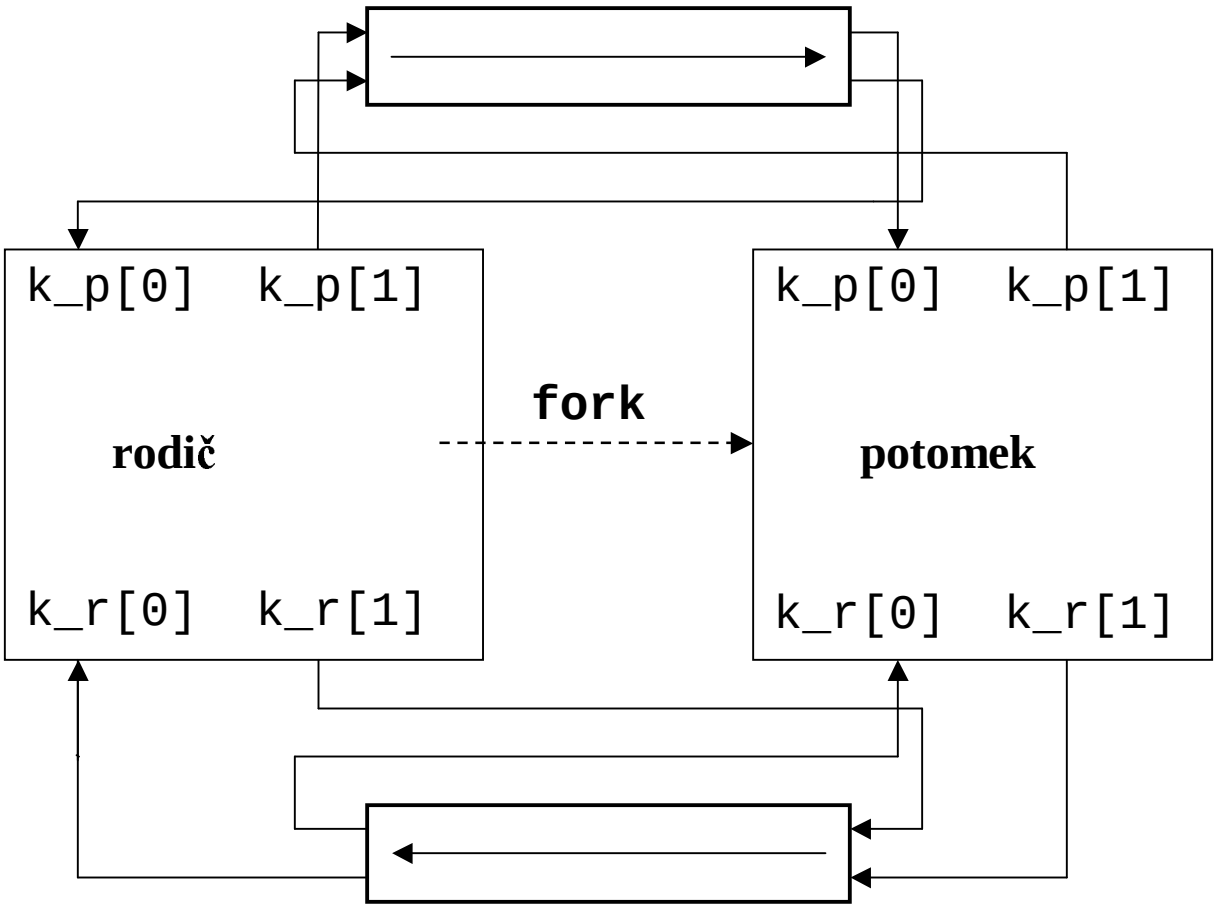


- komunikace mezi procesy

  - proces vytvoří rouru voláním **pipe**

- voláním **fork** vytvořené procesy získají deskriptory souborů roury
- procesy čtou z a zapisují do roury
- synchronizace

- příklad obousměrné komunikace rodiče a potomka



◄

- vytvoříme dvě roury pro tok dat k rodičovi **k_r** a tok

dat k potomkovi **k_p**

- potomek má vlastní kopie deskriptorů souborů pro obě roury
- standardní vstupy a výstupy přesměrujeme na roury

```
char string[] = “ahoj”;
main()
{
    int pocet, i;
    int k_r[2], k_p[2];
    char b[64];
    pipe(k_r);
    pipe(k_p);
    if(fork() == 0)
    {
        /*potomek*/
        close(0);
        dup(k_p[0]);
        close(1);
        dup(k_r[1]);
        close(k_r[1]);
        close(k_p[0]);
        close(k_r[0]);
        close(k_p[1]);
        for(;;)
        {
            if ((pocet=
              read(0,b,sizeof(b))) == 0)
               exit();
            write(1,buf,pocet);
        }
    }
    /*rodic*/
    close(1);
```

```
    dup(k_p[1]);
    close(0);
    dup(k_r[0]);
    close(k_p[1]);
    close(k_r[0]);
    close(k_p[0]);
    close(k_r[1]);
    for (i=0; i<3; i++)
    {
       write(1,string,strlen(string));
       read(0,buf, sizeof(buf));
    }
}
```

vykonání:

potomek buď najde v rouře **k_p** data anebo počká až je tam rodič vloží

když je přečte vloží je do roury **k_r**

rodič třikrát vloží data do roury **k_p** a potom přečte nebo čeká na data z roury **k_r**

ahoj ahoj ahoj

a po jejich třetím přečtení skončí

potomek, po třetím přečtení čeká na další data
protože žádný proces nemá otevřený deskriptor souboru pro zápis, nikdo už do roury data nezapíše
volání **read** vrátí konec souboru, tedy nula přečtených bytů a potomek skončí – **exit**

proč zavírat nadbytečné deskriptory souborů?

- šetříme
- vykonáváním **`fork`** a **`exec`** získávají nezavřené deskriptory souborů další procesy a v nich vykonávané programy
- **`read`** z roury vrátí konec souboru jenom tehdy není-li otevřená pro zápis

Vykonatelný (*executable*) program

obyčejný soubor určený na vykonání na HW v prostředí OS

více formátů

a.out Assembler OUTput Format
ELF Executable and Linking Format (Linux, System V)
COFF Common Object File Format (BSD)

Mají následující strukturu:

1. Primární hlavička identifikující typ vykonatelného programu, často formou magického čísla, počet sekcí, začáteční hodnotu počítadla instrukcí

2. Hlavičky sekcí s velikostí sekce, virtuální adresou, ...

3. Sekce obsahující „data“, text (instrukční segment), inicializovaná data, informace o neinicializovaných

datech (bss *block started by symbol*)

4. Jiné sekce obsahující tabulku symbolů užitečnou pro ladění

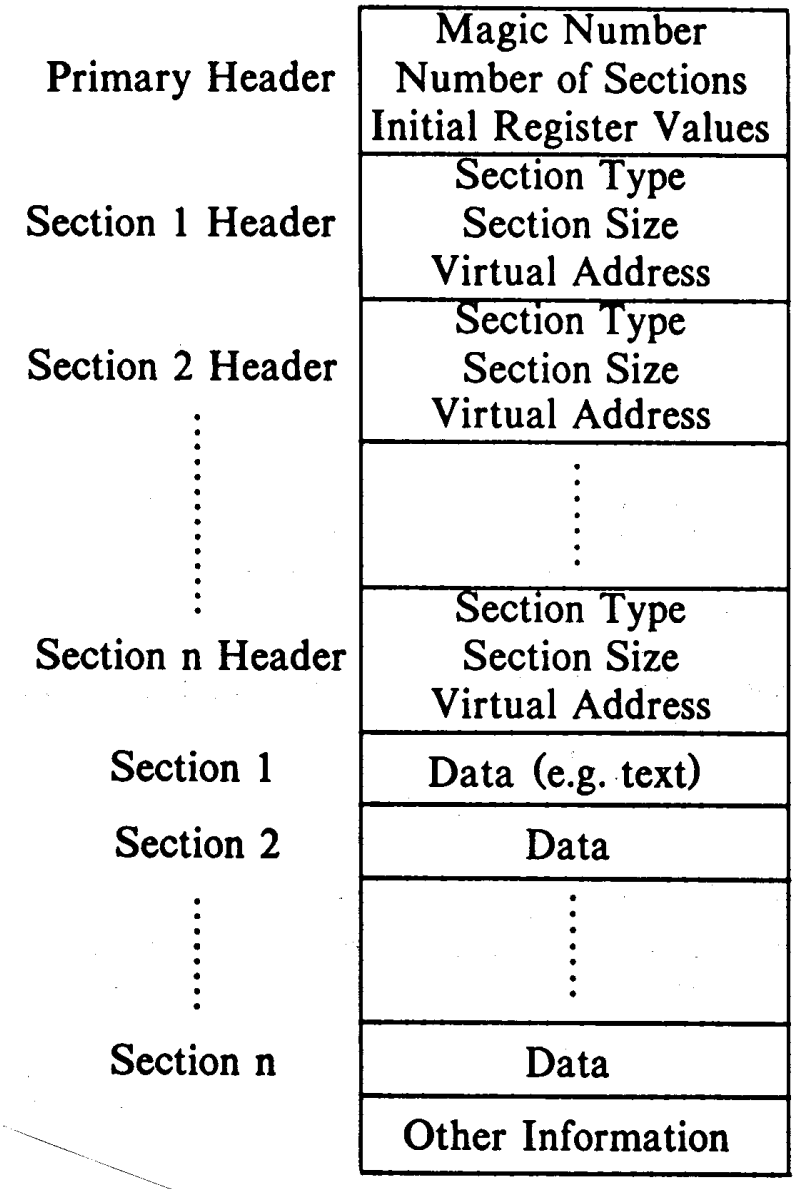


Image of an Executable File

Shell

interpret příkazů

- první slovo (symbol) na řádku je interpretován jako jméno příkazu
    - kód vykonatelného programu, např. po kompilaci programu v C jazyce
    - vykonatelný program jako posloupnost příkazů shellu
    - vnitřní (vestavěné) příkazy, vykoná shell
        - příkazy pro řízení vykonávání *if*, *for*, *while*
        - cd, who...

- příkazy mohou být vykonávány
    - synchronně, shell čeká na vykonání příkazu před čtením následujícího příkazu
    - asynchronně, v pozadí, za příkazem následuje &, shell začne vykonávat příkaz a je připraven přijmout další příkaz

- přesměrování
    - **< soubor** použij soubor jako standardní vstup
    - **> soubor** použij soubor jako standardní výstup
    - **2> soubor** použij soubor jako standardní chybový výstup

- kolona

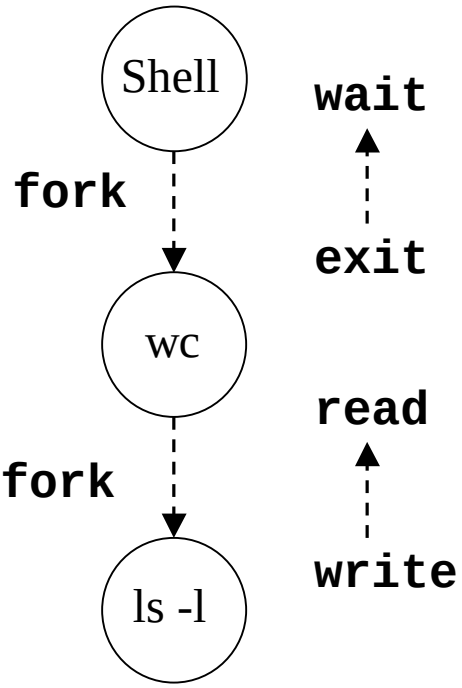
```
ls -l | wc
```

```
/*read command line until “end of line”*/
while(read(stdin, buffer, numchars))
{
    /*parse command line*/
    if(/*command line contains & */)
            amper = 1;
    else
            amper = 0;
    /*for commands not part of the shell
      command language*/
    if (fork() == 0)
    {
        /*redirection of IO?*/
        if (/*redirect output*/)
        {
            fd = creat(newfile, fmask);
            close(stdout);
            dup(fd);
            close(fd];
            /*stdout is now redirected*/
        }
        if(/*piping*/)
        {
            pipe(fildes);
            if (fork() == 0)
            {
                /*first component of command
                  line*/
                close(stdout);
                dup(fildes[1]);
                close(fildes[1]);
                close(fildes[0]);
```

```
            /*stdout now goes to pipe*/

            /*child process does
              command*/
            execlp(command1,command1,0);
          }
          /*2nd command component of
           command line*/
          close(stdin);
          dup(fildes[0]);
          close(fildes[0]);
          close(fildes[1]);
          /*standard input now comes from
           pipe*/
       }
       execve(command2,command2,0);
    }
    /*parent continues over here...
     *waits for child to exit if required
     */
    if(amper == 0)
       retid = wait(&status);
}
```

[Bach 86]

```
who
ls -l

nroff –mm velkydokument &
nroff –mm velkydokument > vystup

ls –l | wc
```



Zavedení operačního systému

- „nezávislost“ HW a OS
  - na jedné HW architektuře různé OS, Linux/Windows
  - na různých HW architekturách stejný OS (vyčlení se strojově závislá část OS)

- při zapnutí počítače v hlavní paměti není žádný program
- operační systém musí zavést sám sebe

- *bootstrap, to boot*

Main Entry: [1]**boot·strap** □
Pronunciation: 'büt-"strap
Function: *noun*
Date: 1913
1 *plural* **:** unaided efforts -- often used in the phrase *by one's own bootstraps*
2 : a looped strap sewed at the side or the rear top of a boot to help in pulling it on

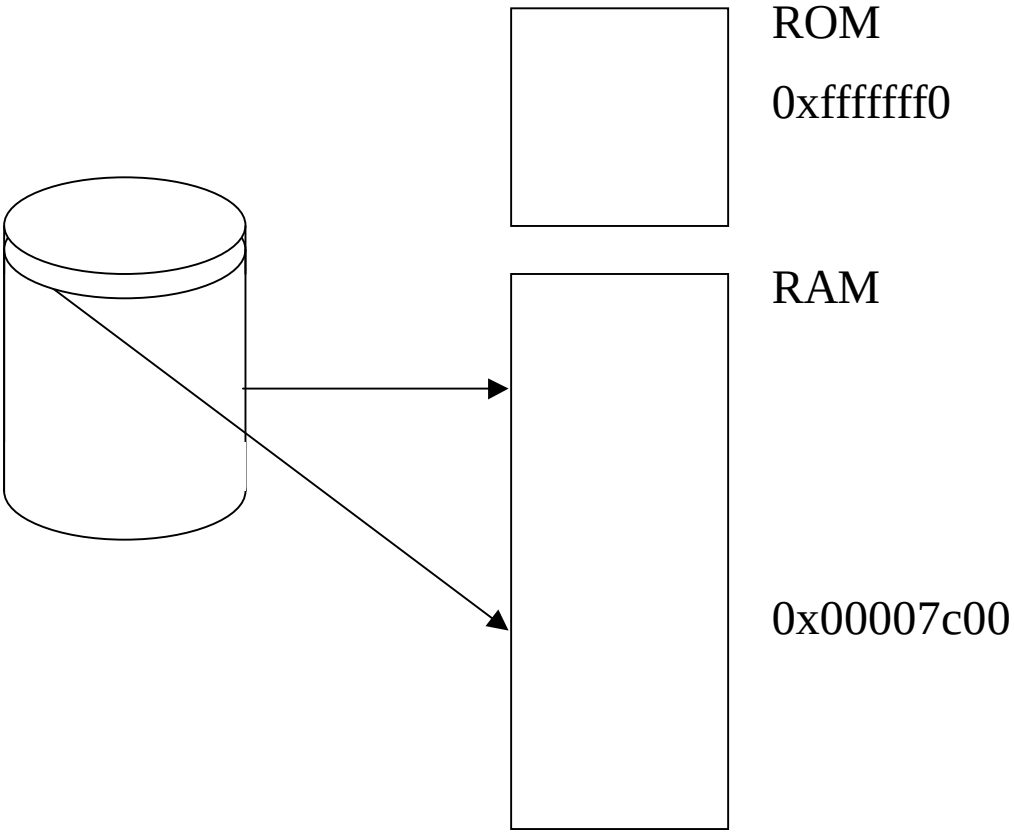
---

Main Entry: [3]bootstrap
Function: *transitive verb*
Date: 1951
: to promote or develop by initiative and effort with little or no assistance <*bootstrapped* herself to the top>

Merriam-Webster Online

- zavedení OS je posloupnost kroků:
    - po připojení k síti HW generuje RESET
    - začne se vykonávat program v trvalé paměti (ROM) – strojový zavaděč, který čte první sektor z disku do hlavní paměti
      (PC paměť ROM adresa 0xffffff0, BIOS
        1. test HW (přítomnost zařízení)
        2. inicializace HW (tabulka instalovaných zřízení)
        3. hledá disk s operačním systémem (pružné, pevné, CD-ROM)

4. přečte první sektor a zapíše ho do RAM, adresa 0x00007c00 a vykoná skok na tuto adresu



- začne se vykonávat zavaděč operačního systému (boot loader), který je (nebo jeho začátek) v prvním sektoru, který z disku do RAM přečte jádro OS

Linux

- zavedení z pružného disku
  - komprese při kompilaci
  - dekomprese při zavádění
  - zavaděč je v jazyce symbolických instrukcí (assembly language)

    - po přeložení jádra je zavaděč umístěn na začátek souboru s přeloženým jádrem
    - zapíše se na pružný disk od prvního sektoru
    - BIOS tedy přečte zavaděč a vykoná skok na jeho začátek
    - zavolá proceduru BIOSu na vypsání „Loading ...“
    - zavolá proceduru BIOSu na zavedení funkce **`setup()`**jádra na adresu 0x00090000 a zavede zbytek jádra
    - skok na **`setup()`**

- zavedení z pevného disku
- obecně
    - pevný disk je rozdělen na oblasti, které můžeme považovat za logické disky
    - první sektor disku, MBR *master boot record* obsahuje tabulku oblastí a krátký program, který zavádí první sektor oblasti, která je označena jako aktivní

- LILO (LInux LOader)

- dvoustupňové zavádění
    - instalován
        - v MBR namísto programu, který zavádí první sektor aktivní oblasti
        - v prvním sektoru aktivní oblasti
    - dvě části
    - první část zavede BIOS na adresu 0x000007c0 a tato zavede druhou část do RAM na adresu 0x0009b000
    - druhá část zjistí operační systémy na disku a nabídne uživateli, aby si vybral

- po výběru (anebo po uplynutí čekací doby předdefinovaný *default*) přečte první sektor vybrané oblasti
- jestli je zaváděn Linux, zavaděč vypíše „Loading ...“
- zavede funkci **`setup()`** jádra na adresu 0x00090000 a zavede zbytek jádra
- skok na **`setup()`**

- **`setup()`**
  - zjistí velkost RAM
  - inicializuje anebo reinicializuje přídavná zařízení, ...
  - skok na funkci **`startup_32()`**

- **`startup_32()`**
  - vykonává dekompresi
  - vytvoří proces 0
  - skok na **`start_kernel()`**

- **`start_kernel()`**
  - inicializuje téměř všechny součásti jádra
  - vytvoří proces 1 s programem **`init`**

UNIX obecně

- při zavádění vytvoří proces 0 běžící v módě jádro
- proces 0 vytvoří službou **`fork`** proces 1, který sám sebe přepíše do uživatelského adresového prostoru
- proces 1 vykoná **`exec( “ /.../init”, ...)`**

- **`init`**

čte řádky souboru **inittab** a vytváří procesy, ve kterých vykoná **exec** programu specifikovaného v řádku, pro terminály **getty**

- **getty**

  otevření zařízení jako otevření souboru, **open** vrátí deskriptor souboru, vykoná se však specificky pro jednotlivé druhy zařízení

  pro terminál, **open** čeká na vstup

```
{
  ...
  open terminál;
  if(otevření úspěšné)
  {
     exec login;
     if(úspěšné přihlášení)
     {
        ...
        exec shell;
     }
     else
        počítej pokusy;
        opakuj pro povolený počet;
 }
}
```

- úspěšné přihlášení
  začal přihlašovací (*login*) shell, **init** čeká na jeho skončení (login shell je potomek) a vytvoří nový **getty**

- neúspěšné přihlášení
  `login` vykoná **`exit,`** zavře se terminál, **`init`** vytvoří nový **`getty`**