

ZÁKLADY OPERAČNÍCH SYSTÉMŮ

I.

KIV/ZOS 2024

L. Pešička

Kontaktní informace

- Ing. Ladislav Pešička
- UN 358
- *pesicka@kiv.zcu.cz*
 - *Předmět zprávy začít: ZOS*
- Úřední hodiny
 - Út 10:00 až 11:00
 - St 10:00 až 11:00
- Všechny informace najdete v coursewaru

Požadavky na zápočet

- **Zápočtový test**
- **Semestrální práce**
 - téma: práce se souborovým systémem zjednodušený oproti reálnému systému
- **Linuxový online kurz (NDG Linux Essential)**
 - testy s alespoň 60 procent úspěšností
 - není povinný, ale velmi doporučený, bonus
- **Mezní termín získání zápočtu : 10. února 2025**

Zápočtový test

- časově cca koncem listopadu (CW – významná data)
- znalosti Linuxu ze cvičení
 - Jaký je rozdíl v `>`, `>>`, `2>`, `2>>`
 - Nastavení přístupových práv na soubor `s1`
 - Jak vypíšete právě druhou řádku ze souboru `s1`
- řešení příkladů podobných příkladům na cvičení
- otázky z přednášek

NDG Linux Essential - kurz

NDG Linux Essentials - Chapter 9 - Basic Scripting

Content Objectives Key Terms

```
#!/bin/bash
CURRENT_DIRECTORY=`pwd`
echo "You are in $CURRENT_DIRECTORY"
```

This pattern is often used to process text. You might take text from one variable or an input file and pass it through another command like `sed` or `awk` to extract certain parts and keep the result in a variable.

It is possible to get input from the user of your script and assign it to a variable through the `read` command:

```
#!/bin/bash
echo -n "What is your name? "
read NAME
echo "Hello $NAME!"
```

The `read` command can accept a string right from the keyboard or as part of command redirection like you learned in the last chapter.

There are some special variables in addition to the ones you set. You can pass arguments to your script:

```
#!/bin/bash
echo "Hello $1"
```

A dollar sign followed by a number *N* corresponds to the *N*th argument passed to the script. If you call the example above with `./test.sh` the output will be `Hello Linux`. The `$0` variable contains the name of the script itself.

After a program runs, be it a binary or a script, it returns an *exit code* which is an integer between 0

Ubuntu PC

```
total 72
drwxr-xr-x 1 root root 4096 Mar 14 2016 bin
drwxr-xr-x 2 root root 4096 Apr 19 2012 boot
drwxr-xr-x 5 root root 400 Sep 18 20:41 dev
drwxr-xr-x 1 root root 4096 Sep 18 20:40 etc
drwxr-xr-x 1 root root 4096 Mar 14 2016 home
-rwxr-xr-x 1 root root 226 Mar 14 2016 init
drwxr-xr-x 1 root root 4096 Mar 14 2016 lib
drwxr-xr-x 2 root root 4096 Mar 3 2016 lib64
drwxr-xr-x 2 root root 4096 Mar 3 2016 media
drwxr-xr-x 2 root root 4096 Apr 19 2012 mnt
drwxr-xr-x 2 root root 4096 Mar 3 2016 opt
dr-xr-xr-x 1449 root root 0 Sep 18 20:41 proc
drwx----- 1 root root 4096 Mar 14 2016 root
drwxr-xr-x 1 root root 4096 Sep 18 20:41 run
drwxr-xr-x 1 root root 4096 Mar 14 2016 sbin
lrwxrwxrwx 1 root root 9 Mar 14 2016 sbin??? -> /bin/bash
drwxr-xr-x 2 root root 4096 Mar 5 2012 selinux
drwxr-xr-x 2 root root 4096 Mar 3 2016 srv
dr-xr-xr-x 13 root root 0 Aug 11 09:45 sys
drwxrwxrwt 1 root root 4096 Mar 14 2016 tmp
drwxr-xr-x 1 root root 4096 Mar 14 2016 usr
drwxr-xr-x 1 root root 4096 Mar 14 2016 var
sysadmin@localhost:~$
```

V pravé části obrazovky je k dispozici konzole, kde si můžete rovnou zkoušet probírané příkazy a další věci

Zkouška

- Písemka
 - test na 90 min. bez pomůcek
 - zvolit správnou odpověď, odpovědět na otázku, doplnit či nakreslit diagram atd..
 - vyhodnocení písemky (body, známka) mailem
 - přijít si prohlédnout písemku, konzultace

OS v běžném životě - mobily

- **Android**

- Poslední verze Android 15
- Na Linuxovém jádru
- Roztříštěnost verzí mezi uživateli

- **iOS**

- Poslední verze iOS 18
- Apple iPhone, iPad (iPadOS)
- Většina zařízení - poslední verze OS



OS ve vesmíru



Photograph courtesy of NASA/JPL CALTECH

NASA's Twin Mars Rovers.

CPU:
radiálně odolný
20MHz PowerPC

Commercial **Real-time OS**

Software and OS was
developed during multi-year
flight to Mars
and downloaded using a
radio link

OS – příklady použití

- **Servery, pracovní stanice, notebooky**
 - MS Windows, GNU/Linux, macOS
- **Mobilní zařízení, tablety**
 - Android, iOS, Windows CE, Symbian, Linux, ...
- **Router, AP, SOHO síťová zařízení**
 - Cisco IOS, Linux, VxWorks
- **Embedded zařízení**
 - Bankomaty, stravovací systémy, lékařské přístroje
 - MS Windows, Linux
 - Windows CE, Windows XP embedded

OS

Dva základní pohledy na OS:

- **Rozšířený stroj** (shora dolů)
 - Holý počítač -> rozšířený stroj
 - Místo jednoduchých strojových instrukcí komplexní akce
 - Ulož číslo do registru vs. zobraz řetězec „ahoj“
- **Správce zdrojů** (zdola nahoru)
 - OS jako manager, přiřazuje a spravuje zdroje
 - Zdroje: čas CPU, alokace v RAM, přístup k disku

OS jako rozšířený stroj

- Holý počítač
 - Obtížně programovatelný
 - Strojové instrukce – jednoduché činnosti
 - Např. přístup k rotačnímu disku
 - Práce s hlavičkou disku
 - Alokace a dealokace bloků dat na disku
 - Více programů chce sdílet stejné médium
- Jako programátor chceme
 - Vysokoúrovňové služby
 - pojmenované soubory a operace nad nimi

OS jako rozšířený stroj

- Strojové instrukce (holý stroj)
- **Vysokoúrovňové služby** (rozšířený stroj)
 - „Rozšířené instrukce“
 - Systémová volání – vyvolání služeb systému
- Z pohledu programátora
 - Pojmenované soubory
 - Neomezená paměť
 - Transparentní I/O operace
 - OS skrývá před aplikacemi podrobnosti o HW (přerušení, správu paměti..)
- Příklad: chceme vypsát ahoj: `printf("Ahoj\n");`
 - Holý počítač by uměl jen operaci „rozsvit’ pixel“
 - Potřebujeme fonty atp.

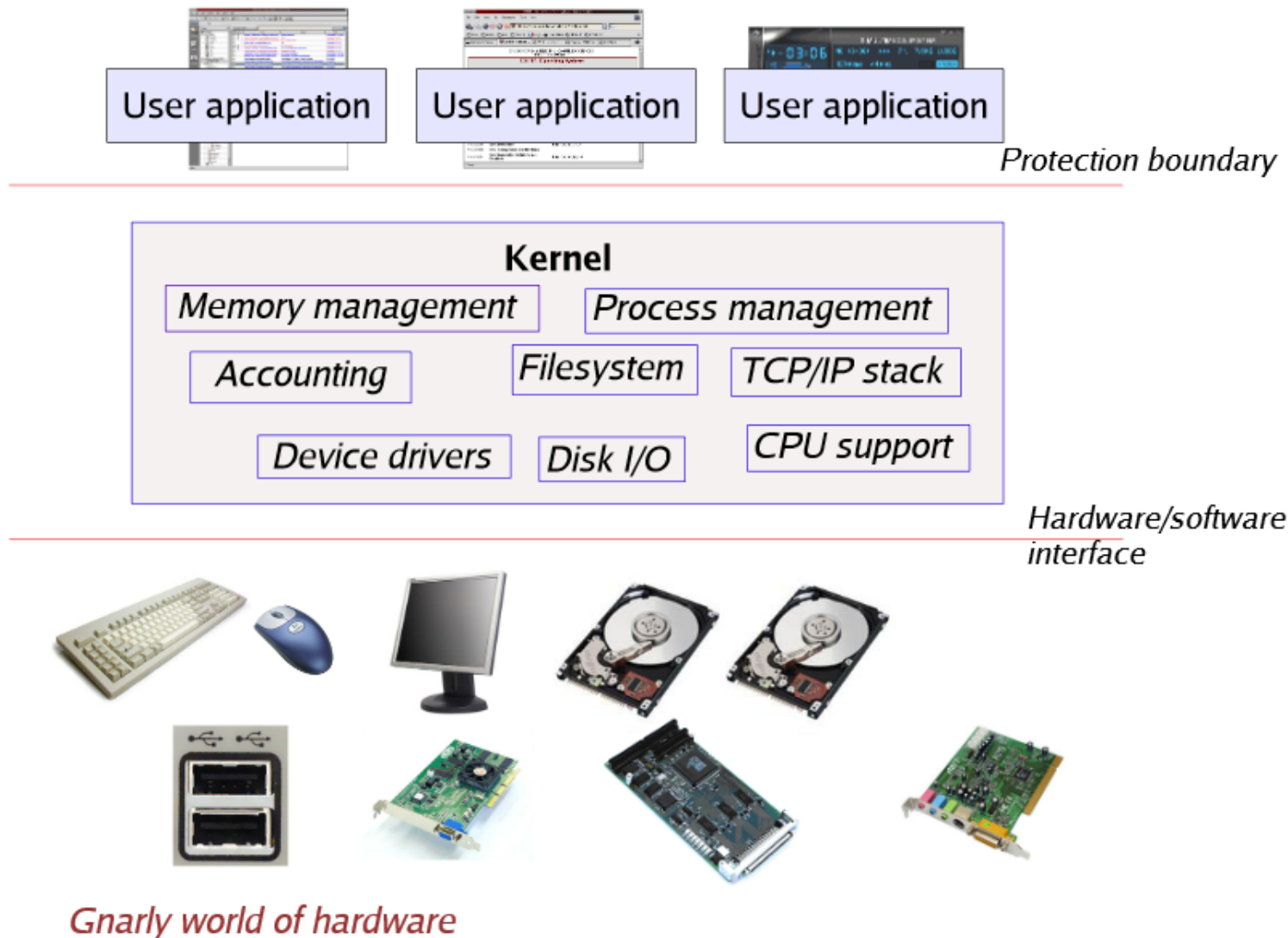
OS jako správce zdrojů

- OS jako poskytovatel / správce zdrojů (resource manager)
- Různé zdroje (čas CPU, paměť, I/O zařízení)
- OS – správná a řízená **alokace** zdrojů procesům, které je požadují (přístupová práva)
- **Konfliktní požadavky** na zdroje
 - V jakém pořadí vyřízeny
 - Efektivnost, spravedlivost

Struktura OS (!!)

- modul pro správu **procesů**
 - program, proces, vlákno, plánování procesů a vláken
 - kritická sekce, souběh, synchronizace (semaforey, ...)
 - deadlock, vyhladovění
- modul pro správu **paměti**
 - virtuální paměť: stránkování, segmentace
- modul pro správu **I/O**
- modul pro správu **souborů**
- síťování
- ochrana a bezpečnost
- uživatelské rozhraní

Operating System Overview



Operační systém - definice

OS je softwarová vrstva (základní programové vybavení), jejíž úlohou je spravovat hardware a poskytovat k němu programům jednotné rozhraní

- OS zprostředkovává aplikacím přístup k hardwaru
- OS koordinuje zdroje a poskytuje služby aplikacím
 - Zdroje – čas na procesoru , přidělená paměť, disk, síťová karta
- OS je program, který slouží jako prostředník mezi aplikacemi a hardwarem počítače.

Program, proces

- **Program**

- Posloupnost instrukcí, která popisuje realizaci dané úlohy počítačem
 - Spustitelný kód, v binární podobě ve formě strojových instrukcí pro dané CPU
 - Příkazy programovacího jazyka (např. Python, Bash skript)
- Nejčastěji soubor uložený na disku
- Např. C:\windows\system32**calc.exe**

- **proces** – instance běžícího programu

- **PID** (process id) – číslo přidělené procesu systémem
- Přidělen **čas CPU**
- Potřebuje **paměťový prostor**
- **vstupy a výstupy**
- *Dle jednoho programu můžeme spustit více procesů*

PROGRAM

PROCES

PROCES

Windows

The image shows a Windows Explorer window displaying the contents of the 'System32' folder. The file 'calc.exe' is highlighted. Below it, the 'Task Manager' window is open, showing the 'Processes' tab. The 'Processes' list includes 'calc.exe' with two entries, each with a unique PID (5596 and 6944). Arrows indicate that 'calc.exe' is the program, and the two entries in the Task Manager are separate processes of that program.

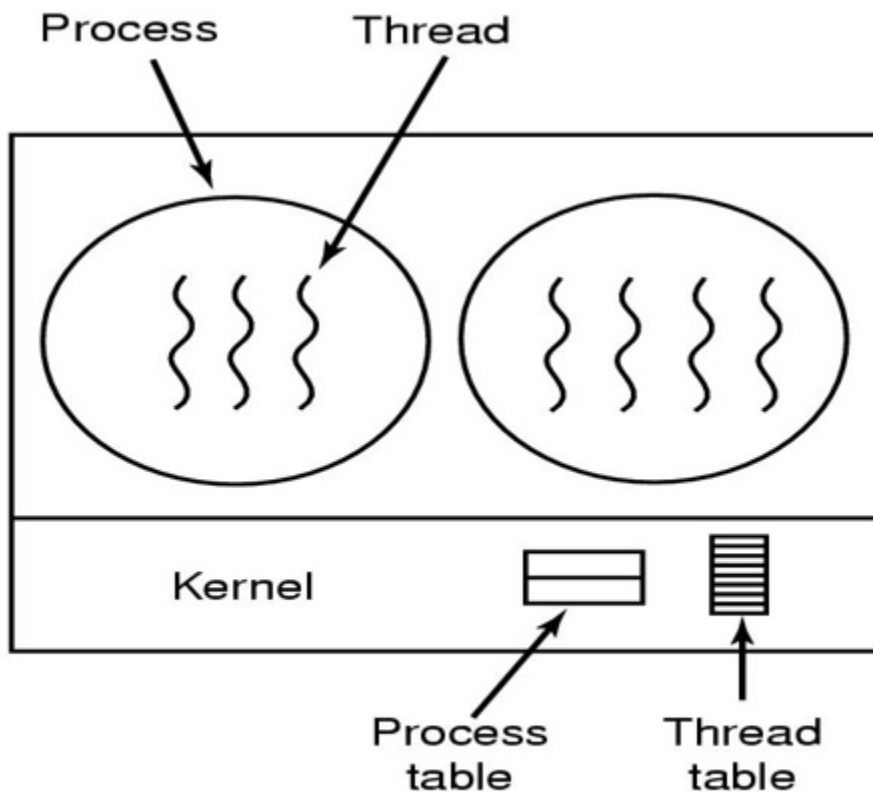
| Název položky | Datum změny | Typ | Velikost |
|------------------|----------------|---------------------|----------|
| calc.exe | 14.7.2009 3:38 | Aplikace | 897 kB |
| capiprovider.dll | 14.7.2009 3:40 | Roziřování aplikace | 53 kB |
| capis... | 7.2009 | | 25 kB |
| Card... | 7.2009 | | 59 kB |
| catsn... | 7.2009 | | 52 kB |
| catsn... | 7.2009 | | 55 kB |
| catsn... | 7.2009 | | 14 kB |
| cca.d... | 11.2010 | | 93 kB |
| cdd.d... | 11.2010 | | 41 kB |

| Název procesu | PID | Uživatel... | P... | Paměť (soukromá pracovní sada) | Popis |
|------------------|------|-------------|------|--------------------------------|--------------------|
| acrotray.exe *32 | 1260 | pesicka | 00 | 1 568 kB | AcroTray |
| avgui.exe *32 | 5624 | pesicka | 00 | 6 204 kB | AVG User Interface |
| calc.exe | 5596 | pesicka | 00 | 5 940 kB | Windows Calculator |
| calc.exe | 6944 | pesicka | 00 | 5 948 kB | Windows Calculator |
| csrss.exe | 904 | | 00 | 2 768 kB | |

PID (ID procesu) – základní identifikátor procesu !!

Vlákno

- Proces se může skládat z více vláken

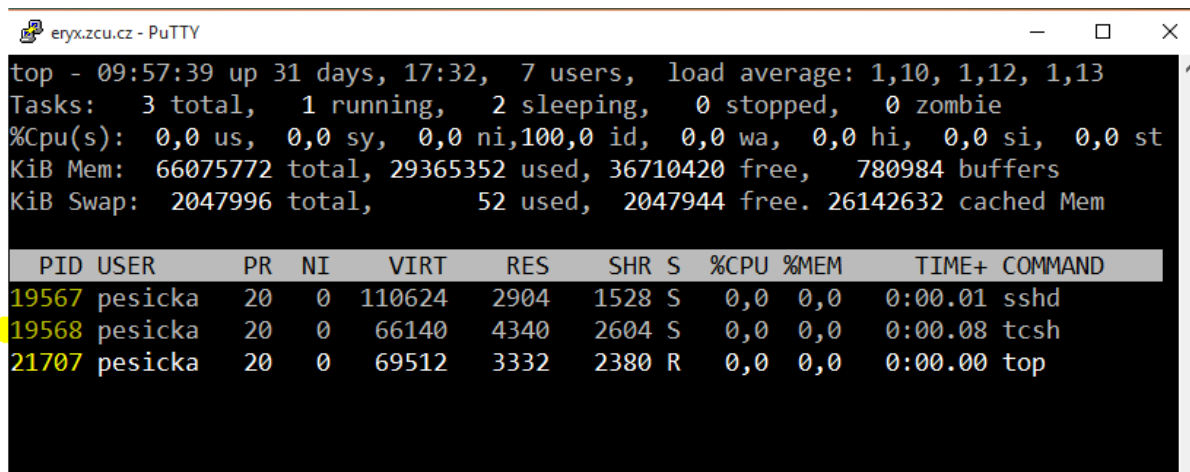


PID - Linux

- Příkaz **ps**

```
eryx3> ps x
  PID TTY          STAT TIME  COMMAND
19567 ?            S      0:00  sshd: pesicka@pts/6
19568 pts/6        Ss      0:00  -tcsh
19583 pts/6        R+      0:00  ps x
eryx3>
```

- Příkaz **top**



```
top - 09:57:39 up 31 days, 17:32, 7 users, load average: 1,10, 1,12, 1,13
Tasks: 3 total, 1 running, 2 sleeping, 0 stopped, 0 zombie
%Cpu(s): 0,0 us, 0,0 sy, 0,0 ni,100,0 id, 0,0 wa, 0,0 hi, 0,0 si, 0,0 st
KiB Mem: 66075772 total, 29365352 used, 36710420 free, 780984 buffers
KiB Swap: 2047996 total, 52 used, 2047944 free. 26142632 cached Mem
```

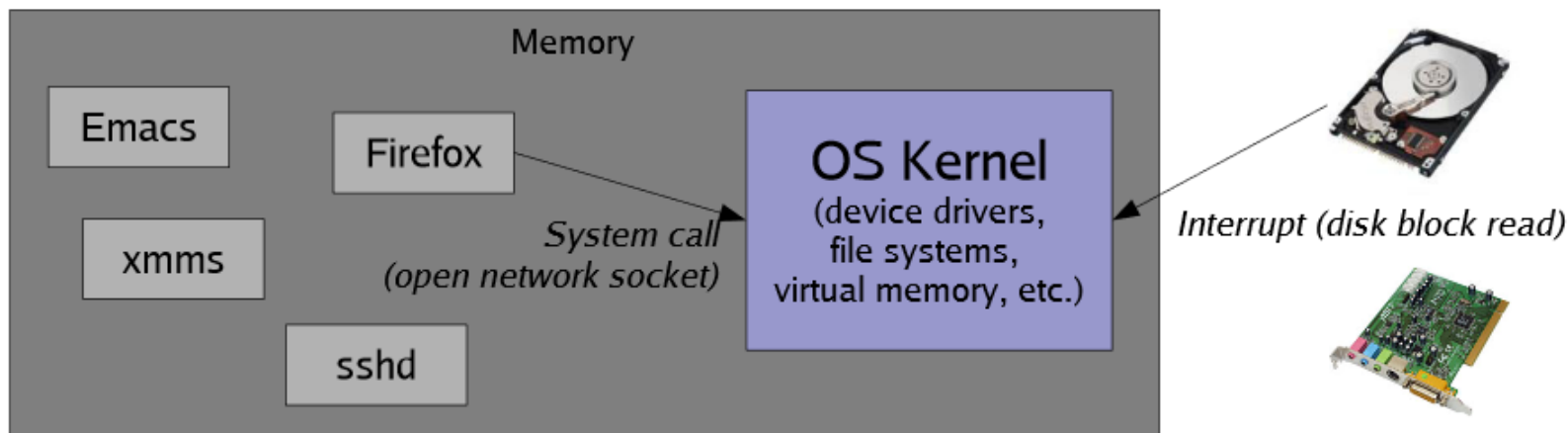
| PID | USER | PR | NI | VIRT | RES | SHR | S | %CPU | %MEM | TIME+ | COMMAND |
|-------|---------|----|----|--------|------|------|---|------|------|---------|---------|
| 19567 | pesicka | 20 | 0 | 110624 | 2904 | 1528 | S | 0,0 | 0,0 | 0:00.01 | sshd |
| 19568 | pesicka | 20 | 0 | 66140 | 4340 | 2604 | S | 0,0 | 0,0 | 0:00.08 | tcsh |
| 21707 | pesicka | 20 | 0 | 69512 | 3332 | 2380 | R | 0,0 | 0,0 | 0:00.00 | top |

Základní pojmy z oblasti OS

- Jak OS **pomáhá**?
 - Chceme otevřít soubor
 - Nemusíme zkoumat zda je umístěn na SSD či rotačním disku, jak vystavovat hlavičky disku, ...
 - Použijeme **systemové volání** OS (open) nebo knihovní funkci (fopen), která systém zavolá
- Jak OS **chrání**?
 - Kontrola, zda danou činnost smíme provést
 - Aplikace běží v **uživatelském režimu** CPU
 - Některé instrukce jdou vykonávat pouze v **privilegovaném režimu** CPU, v kterém běží jádro (např. instrukce pro přímý přístup k HW jako je IN, OUT)

Operating System basics

The OS kernel is just a bunch of code that sits around in memory, waiting to be executed



OS is triggered in two ways: *system calls* and *hardware interrupts*

System call: Direct "call" from a user program

- For example, `open()` to open a file, or `exec()` to run a new program

Hardware interrupt: Trigger from some hardware device

- For example, when a disk block has been read or written

How else might the kernel get control ???

Privilegovaný a uživatelský režim procesoru

- **Jádro** OS běží v tzv. **privilegovaném** režimu CPU
 - Všechny instrukce CPU jsou zde povoleny

CPU ví v jakém režimu se nachází

aplikace nemá
přímý přístup k
HW

- **Aplikace** – v **uživatelském** režimu CPU
 - Některé instrukce **zakázány** (tzv. **privilegované** instrukce)
např. není přímý přístup k disku, narušitel -> formátování
 - Při pokusu o vykonání privilegované instrukce => chyba, výjimka
 - Aplikace musí požádat OS o přístup k souboru,
ten rozhodne zda jej povolí
- OS může zasahovat do běhu aplikací (např. ukončit je)
- Aplikace může požádat OS o službu

2 základní režimy CPU

- **Uživatelský režim**

- V tomto režimu běží aplikace (word, kalkulačka,..)
- **Nemůžou** vykonávat všechny instrukce, např. přímý přístup k zařízení (tj. zapiš datový blok x na disk y)
 - Proč? Jinak by škodlivá aplikace mohla např. smazat disk
 - Jak se tomu zabrání? Aplikace musí požádat jádro o službu, jádro ověří, zda aplikace má na podobnou činnost oprávnění a jádro činnost provede

- **Privilegovaný režim (režim jádra)**

- Zde jsou **povoleny všechny** instrukce procesoru
- Běží v něm jádro OS, které mj. vykonává služby (systémová volání), o které je aplikace požádá

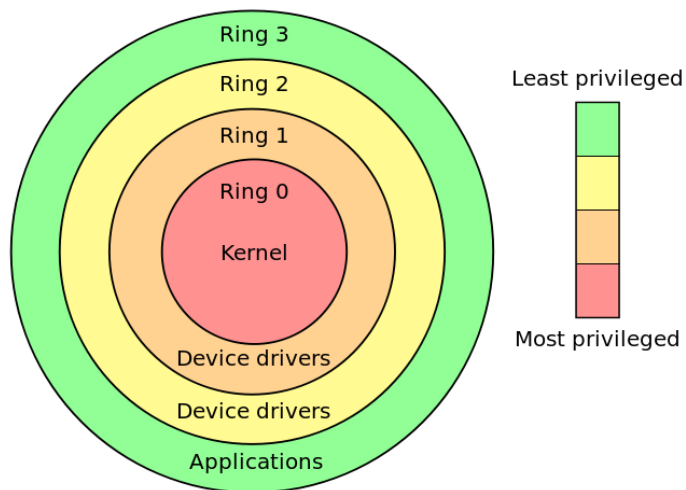
Jak CPU ví, v jakém je režimu?

- obecně
podle bitu ve stavové registru CPU
mode bit 0/1: privilegovaný/uživatelský
„vykonávám kód a vím, jestli je daný kód uživatelský
nebo privilegovaný“
reálně může být bitů více
- Konkrétněji (x86) – zde jen pro zajímavost
na kódový segment odkazuje CS registr, ten má v
deskriptoru privilege level (2bity, ring 0..3)

<http://stackoverflow.com/questions/5223813/how-does-the-kernel-know-if-the-cpu-is-in-user-mode-or-kenel-mode>

Intel x86 CPU – Protection ring

- CPU může poskytovat i více stupňů ochrany, než jen 2 (uživatelský a privilegovaný)
- Ring 0 – jádro (a ovladače v jádře), Ring 3 – aplikace
- Další ringy buď nejsou využity, nebo slouží pro ovladače, nebo pro virtualizační technologie, jako je VirtualBox aj.



Jak se dostat z uživatelského režimu do režimu jádra?

Jde o přepnutí „mezi dvěma světy“, v každém z nich platí jiná pravidla

- Softwarové přerušení – instrukce **INT**
 - začne se vykonávat kód přerušení
- Speciální instrukce
(**sysenter/sysexit**, **syscall/sysret**)
 - Speciální instrukce mikroprocesoru

Systémové volání

Definice:

Mechanismus používaný aplikacemi k volání služeb operačního systému.

Důvod:

- V **uživatelském režimu CPU** není možné celou řadu věcí vykonat – **není přímý přístup k HW**, nelze tedy přímo přečíst blok z disku, tedy otevřít soubor, číst z něj a zapisovat do něj.
- Pokud aplikace takovou činnost požaduje, nezbývá jí, než **požádat o danou službu operační systém**.
- Operační systém **zkontroluje**, zda má aplikace pro danou činnost oprávnění a pokud ano, požadovanou činnost vykoná. (Kontrola může být např. podle ACL, zda má proces daného uživatele právo zapisovat do souboru).

Systémové volání

- Pojem **systémové volání** znamená vyvolání služby operačního systému, kterou poskytuje jádro
- Aplikace, která chce volat nějakou službu:
 - přímo **systémové volání** (*open, creat*),
 - prostřednictvím **knihovní funkce** (v C např. *fopen*), která následně požádá o systémové volání sama.
- Výhodou knihovní funkce je, že je na různých platformách stejná, ať už se vyvolání systémové služby děje různým způsobem na různých platformách.

Systémové volání – příklad (!)

1. Do vybraného registru (EAX) uložím číslo služby, kterou chci vyvolat
 - Je to podobné klasickému číselníku, menu v restauraci apod.
 - Např. služba 1- ukončení procesu, 2- vytvoření dalšího procesu, 3 - čtení ze souboru, 4 – zápis do souboru, 20 – zjištění PIDu našeho procesu
2. Do dalších registrů uložím další potřebné parametry
 - Např. kde začíná řetězec, který chci vypsát
 - Nebo délku řetězce
3. Provedu instrukci, která mě **přepne do režimu jádra**
 - tedy INT 0x80 nebo sysenter
4. V režimu jádra se zpracovává požadovaná služba
 - Může se stát, že se aplikace zablokuje, např. čekání na klávesu
5. Návrat, uživatelský proces pokračuje dále

Př. Hello world v assembleru

```
.data
    s:
        .ascii "hello world\n"
        len = . - s
.text
.global _start
_start:

    movl $4, %eax    /* write system call number */
    movl $1, %ebx    /* stdout */
    movl $s, %ecx    /* the data to print */
    movl $len, %edx  /* length of the buffer */
    int $0x80

    movl $1, %eax    /* exit system call number */
    movl $0, %ebx    /* exit status */
    int $0x80
```

Compile and run with:

```
gcc main.S as -o main.o main.s
ld -o main.out -s main.o
./main.out
```

Do EAX dáme, že chceme **službu číslo 4** (zápis do souboru)

DO EBX dáme, že budeme zapisovat na stdout (obrazovku)

ECX – co chceme vypsát

EDX – velikost dat

Zavoláme SW přerušení INT 0x80 (vstupní bod jádra)

Příklad – Linux - getpid.c

```
int pid;
```

```
int main() {
```

- do registru EAX dáme číslo služby 20
- systémové volání přes int 0x80
- v registru EAX máme návratovou hodnotu pro naši službu (getpid)

```
    __asm__(
```

```
        "movl $20, %eax \n"    /* getpid system call – 20 (0x14) */
```

```
        "int $0x80 \n"        /* syscall */
```

```
        "movl %eax, pid \n"    /* get result */
```

```
    );
```

```
    printf("Test volani systemove sluzby...\nPID: %d\n", pid);
```

```
    return 0;
```

```
}
```


Přehled systémových volání - Linux

<http://syscalls.kernelgrok.com/> (původní zdroj)

jednotlivá volání – co znamenají

je vidět, co se dává do registrů (do EAX – číslo služby, ...)

Linux Syscall Reference

| Show <div>All</div> entries | | | | | | | | Search: <input type="text"/> |
|-----------------------------|---------------------|-----------|-----------------------------|------------------------|--------------|-----|-----|-------------------------------|
| # | Name | Registers | | | | | | Definition |
| | | eax | ebx | ecx | edx | esi | edi | |
| 0 | sys_restart_syscall | 0x00 | - | - | - | - | - | kernel/signal.c:2058 |
| 1 | sys_exit | 0x01 | int error_code | - | - | - | - | kernel/exit.c:1046 |
| 2 | sys_fork | 0x02 | struct pt_regs * | - | - | - | - | arch/alpha/kernel/entry.S:716 |
| 3 | sys_read | 0x03 | unsigned int fd | char __user *buf | size_t count | - | - | fs/read_write.c:391 |
| 4 | sys_write | 0x04 | unsigned int fd | const char __user *buf | size_t count | - | - | fs/read_write.c:408 |
| 5 | sys_open | 0x05 | const char __user *filename | int flags | int mode | - | - | fs/open.c:900 |
| 6 | sys_close | 0x06 | unsigned int fd | - | - | - | - | fs/open.c:969 |
| 7 | sys_waitpid | 0x07 | pid_t pid | int __user *stat_addr | int options | - | - | kernel/exit.c:1771 |
| 8 | sys_creat | 0x08 | const char __user *pathname | int mode | - | - | - | fs/open.c:933 |

Poznámka

- Podrobný popis pro zájemce:
- <https://blog.packagecloud.io/eng/2016/04/05/the-definitive-guide-to-linux-system-calls/>
- Přehled systémových volání - Linux:
- <https://man7.org/linux/man-pages/man2/syscalls.2.html>
- Zkuste zadat do vyhledávače:
[Linux system call cheat sheet](#)

Možnosti programátora v C, když chce službu od jádra

1. inline assembler a INT 0x80
viz předchozí ukázka (reálný příklad Linux)
2. použití instrukce **syscall()**
potřebujeme znát číslo funkce
interně použije INT 0x80
`id1 = syscall (SYS_getpid);`
3. přímo je funkce, např. `getpid()`, `fopen()`
existuje „wrapper“ v libc knihovně
nejpohodlnější a také nejpoužívanější
`id2 = getpid();`

Knihovní funkce

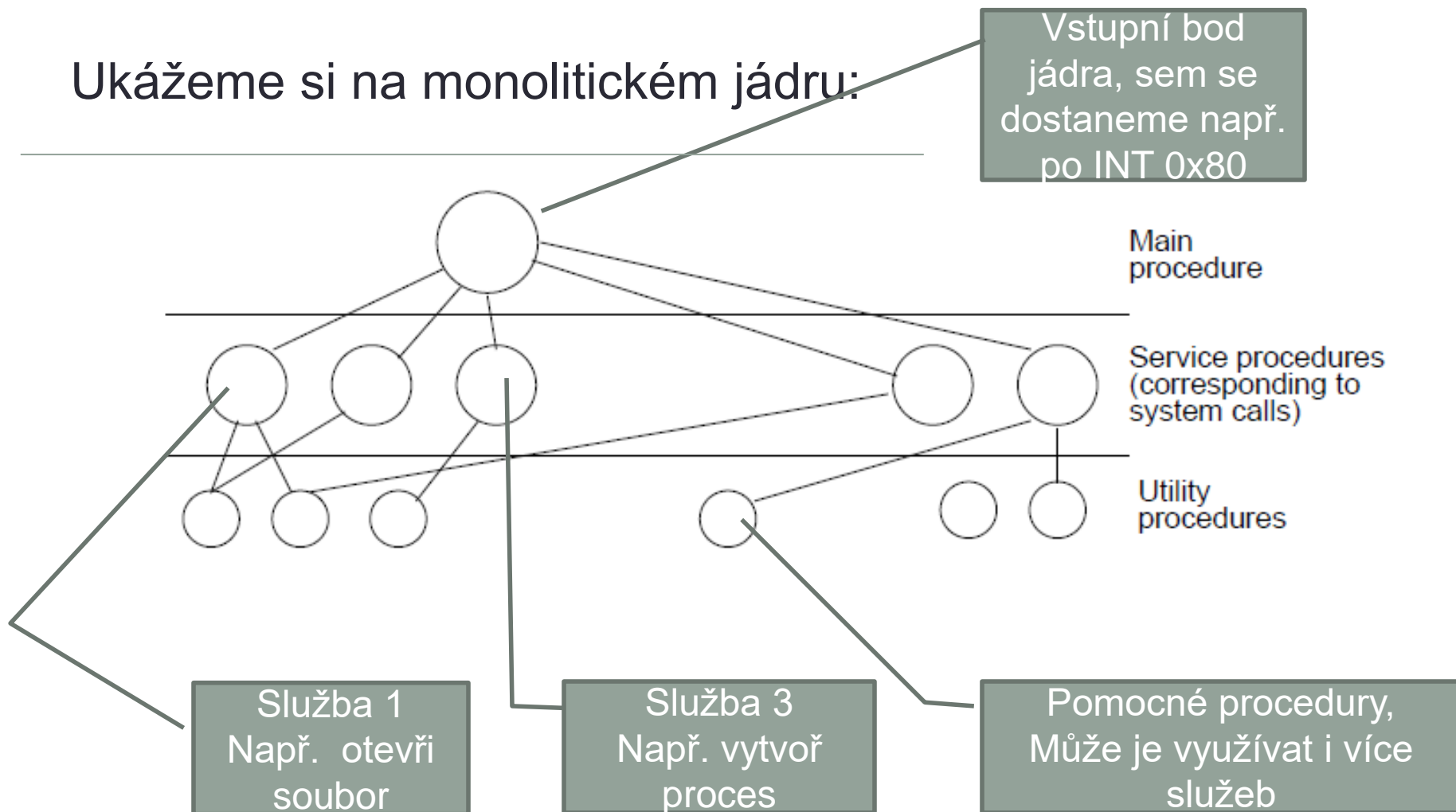
- mechanismy volání jádra se v různých OS liší
- knihovní funkce
 - programovací jazyky zakrývají služby systému, aby se jevily jako běžné knihovní funkce
 - Jazyk C: `printf("Retezec");`
 - Knihovní funkce se sama postará, aby vyvolala vhodnou službu OS na dané platformě pro výpis řetězce
- Ne vždy musí volání knihovní funkce znamenat systémové volání, např. matematické funkce spočítá v uživatelském režimu

Další možnosti uložení parametrů

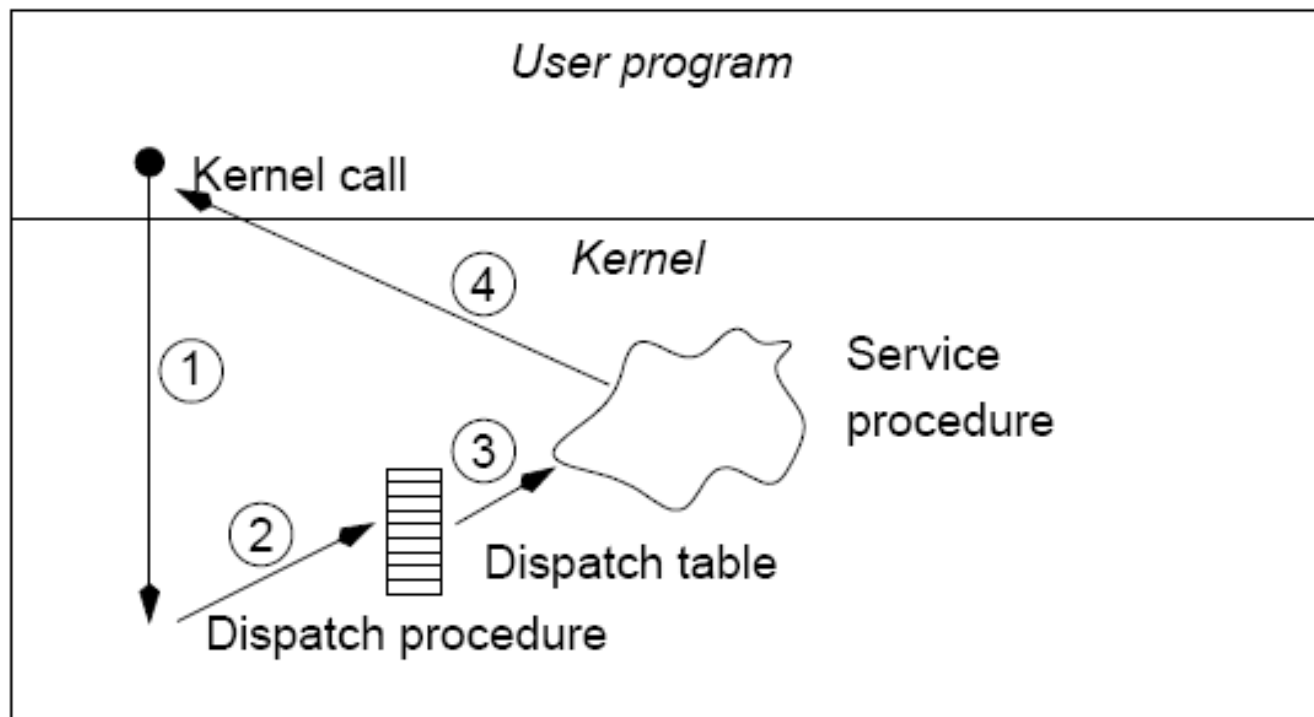
- Musím nějak jádru říci, kterou službu chci a další parametry
- Informaci můžeme uložit
 - Do registrů (nejčastěji – EAX, EBX, ECX)
 - Na zásobník
 - Na předem danou adresu v paměti
 - Kombinací uvedených principů
- Příklad hypotetického volání:
*chci po OS službu 2 (natočení piva) číslo služby uložím do **EAX**, do registru **EBX** uložím velikost piva (malé), do registru **ECX** značku (plzeňská dvanáctka)... vyvoláme INT 0x80*
- *Jádro pozná z EAX, že je zavolána služba 2 a ví, jak interpretovat hodnoty dalších registrů*

Jak jádro implementuje jednotlivé služby?

Ukážeme si na monolitickém jádru:



Vyvolání služby systému (opakování)



Vyvolání služby systému (opakování)

- Parametry uložíme na určené místo
 - **registry**, zásobník, ...
- Provedeme speciální instrukci, např. INT (1)
 - vyvolá obsluhu v jádře
 - **přepne do privilegovaného** režimu
- OS převezme parametry, zjistí, která služba je vyvolána a zavolá příslušnou obsluhu (2,3)
- Návrat zpět (4)
 - Přepnutí do uživatelského režimu (obecně do původního režimu)

Poznámka

- Systémové volání nevyžaduje přepnutí kontextu na jiný proces
- Je zpracováno v kontextu procesu, který jej vyvolal

Co znamená INT x?

- instrukce v assembleru pro x86 procesory, která generuje SW přerušení
- x je číslo v rozsahu 0 až 255
- Index do tabulky vektorů přerušení

INT 0x80

- v 16kové soustavě 80, dekadicky 128
- pro vykonání systémového volání v Linuxu
- do registru EAX se dá číslo systémového volání, které chceme vyvolat

Jak se aplikace dostane do režimu jádra? (opakování)

- **Softwarové přerušení**
 - Volající proces způsobí softwarové přerušení
 - Na platformě x86: instrukce **INT**
 - Přerušení se začne obsluhovat, procesor se přepne do režimu jádra a začne se provádět kód jádra
- **Speciální instrukce**
 - Novější, rychlejší
 - Platforma x86: instrukce **sysenter**, **sysexit**

Může se lišit na různých platformách

Přerušení

- **Přerušení = Událost, kterou je třeba obsloužit !!!**
- **Obsluha přerušení = obsluha události**
- asynchronní (přijde kdykoliv – HW – stisk klávesy)
- synchronní (instrukce SW přerušení v programu - INT), pak přijde očekávaně
- Analogie z reálného života
 - S někým si povídáte
 - Zazvoní telefon, vyřídíte telefon
 - Vráťte se k předchozímu povídání

Přerušení

Definice:

Metoda pro obsluhu událostí, kdy procesor přeruší vykonávání sledu instrukcí, vykoná obsluhu přerušení a pak pokračuje v předchozí činnosti.

Rozdělení:

- **HW přerušení** (vnější) – obsluha HW zařízení (klávesnice)
- **SW přerušení** – synchronní, instrukcí **INT číslo** v kódu procesu
- **Vnitřní přerušení (výjimky)** – procesor oznamuje chyby při vykonávání instrukcí (dělení nulou, neplatná instrukce)

Přerušení (interrupt)

- Přerušení patří k základním mechanismům používaným v OS
- **Asynchronní** obsluha události, procesor **přeruší** vykonávání sledu instrukcí (části kódu, které se právě věnuje), **vykoná** obsluhu přerušení (tj. instrukce v obslužné rutině přerušení) a **pokračuje** předchozí činnosti
- Analogie:
 - vařím oběd (vykonávám instrukce běžného procesu),
 - zazvoní telefon (přijde přerušení, je to asynchronní událost – kdykoliv)
 - Vyřídím telefon (obsluha přerušení)
 - Pokračuji ve vaření oběda (návrat k předchozí činnosti)
- Některé systémy mají **víceúrovňová** přerušení (vnoření)
 - (telefon přebije volání, že na někoho v sousedním pokoji spadla skříň)

Druhy přerušení (!!)

- **Hardwarové přerušení (vnější)**

- Přichází z **I/O zařízení**, např. stisknutí klávesy na klávesnici
- **Asynchronní** událost – uživatel stiskne klávesu, kdy se mu zachce
- Vyžádá si pozornost procesoru bez ohledu na právě zpracovávanou úlohu
- Doručovány prostřednictvím **řadiče přerušení** (umí stanovit **prioritu** přerušením, aj.)

- **Vnitřní přerušení**

- Vyvolá je sám processor
- Např. pokus o dělení nulou, neplatná instrukce, **výpadek stránky paměti** (!!)

- **Softwarové přerušení**

- Speciální strojová instrukce (např. zmiňovaný příklad **INT 0x80**)
- Je **synchronní**, vyvolané záměrně programem (chce službu OS)
volání služeb operačního systému z běžícího procesu (!!)
uživatelská úloha nemůže sama skočit do prostoru jádra OS, ale má právě k tomu softwarové přerušení

- Doporučuji přečíst:

<http://cs.wikipedia.org/wiki/P%C5%99eru%C5%A1en%C3%AD>

Kdy v OS použiji přerušení? (to samé z jiného úhlu pohledu)

- **Systémové volání** (volání služby OS)
 - Využiji softwarového přerušení a instrukce INT
- **Výpadek stránky paměti**
 - V logickém adresním prostoru procesu se odkazují na stránku, která není namapovaná do paměti RAM (rámec), ale je odložená na disku
 - Dojde k přerušení – výpadek stránky
 - Běžící proces se pozastaví
 - Ošetří se přerušení – z disku se stránka natáhne do paměti (když je operační paměť plná, tak nějaký rámec vyhodíme dle nám známých algoritmů 😊)
 - Pokračuje původní proces přístupem nyní už do paměti RAM
- **Obsluha HW zařízení**
 - Zařízení si **žádá pozornost**
 - Klávesnice: stisknuta klávesa
 - Disk : mám k dispozici data
 - Síťová karta: došel paket

Vyvolání HW přerušení

- I/O zařízení signalizuje přerušení (*něco potřebuji*)
- Přerušení přijde na nějaké lince přerušení
(**IRQ**, můžeme si představit jeden drát ke klávesnici, jiný drát k sériovému portu, další k časovači atd.)
- Víme číslo drátu (např. IRQ 1), řadič číslo drátu přemapuje na číslo přerušení, ale potřebujeme vědět, **na jaké adrese** začíná obslužný program přerušení
- Kdo to ví? ... tabulka vektorů přerušení
- Řadič přerušení dodá procesoru informaci o indexu do tabulky vektorů přerušení
- **Vektor přerušení** je vlastně **index do pole**, obsahující **adresu obslužné rutiny**, vykonané při daném typu přerušení

Poznámka

- IRQ 1 (tj. „drát 1“) neznamena, že hledáme přerušení v tabulce vektorů přerušení na indexu 1
- Řadič přerušení **přemapuje** „číslo drátu“ na index do tabulky vektorů přerušení a tuto informaci poskytne CPU
- Řadič přerušení zná mapování číslo IRQ – index do tabulky vektorů přerušení, pokud je více IRQ současně, rozhodne, které bude obslouženo první
- Viz 2. přednáška

Poznámka k přerušením

„signál“ operačnímu systému, že nastala nějaká událost, která vyžaduje ošetření (vykonání určitého kódu) - asynchronní

- **Hardwarové přerušení**

- původ v HW
- Stisk klávesy, pohnutí myši
- Časovač (timer)
- Disk, síťová karta, ztráta napájení,...

- **Softwarová přerušení**

- Pochází ze SW
- Obvykle z procesu v uživatelském režimu

Poznámka k přerušením

Příchod přerušení, z tabulky vektorů přerušení pozná, kde leží obslužný kód pro dané přerušení

Pozn. pro sw přerušení 0x80 ukazuje v tabulce přerušení (vektor přerušení) na vstupní bod OS v Linuxu

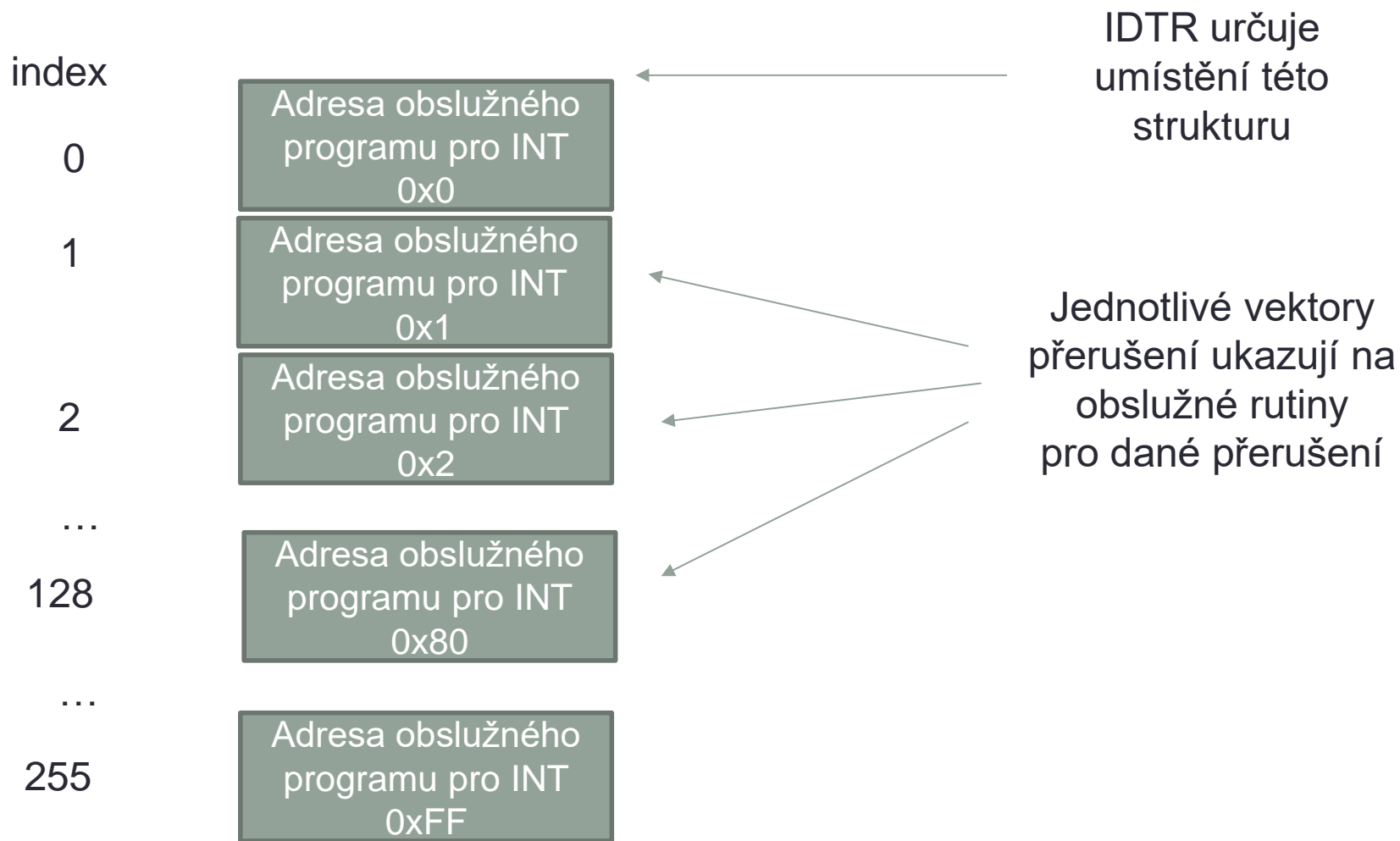
Maskování přerušení – v době obsluhy přerušení (musí být rychlá) lze zamaskovat méně důležitá přerušení

Sw přerušení jsou nemaskovatelná

Tabulka vektorů přerušení (!!!)

- Můžeme si ji představit jako pole,
index do pole - představuje číslo přerušení,
obsah daného prvku pole – adresa obsluhy
- Od adresy 0 do adresy 1KB v RAM
- 256 x 4bytový ukazatel
- Ukazatel – adresa obslužného programu pro dané přerušení
- Toto platí v tzv. **reálném režimu CPU** (MS DOS)
- V tzv. **protected modu CPU** (neplést s privilegovaným) – 8byte ukazatele (tedy celkem 2KB) a začíná od zvolené adresy v paměti – udává registr **IDTR**
- Tabulka IDT

Tabulka vektorů přerušení



4B nebo 8B velikost položky

Tabulka vektorů přerušení

Definice:

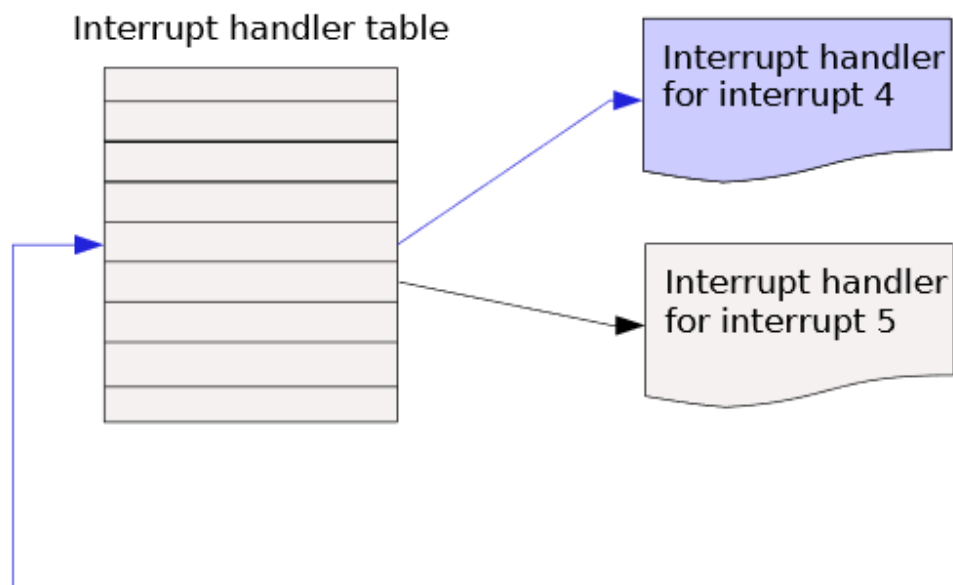
Tabulka vektorů přerušení je datová struktura, ve které se uschovávají vektory přerušení.

(můžeme si ji představit jako pole)

Vektor přerušení – adresa (první instrukce) podprogramu pro obsluhu daného přerušení.

Obsluha přerušení – může mít 2 části

- první část
ve vlastním režimu obsluhy přerušení
velmi rychlé (stabilita)
- odložená část
může naplánovat další část, která se vykoná
„až bude čas“



1) OS fills in interrupt handler table (usually at boot time)

2) Interrupt occurs – e.g., hardware signal



3) CPU state saved to stack

4) CPU consults interrupt table and invokes appropriate handler

Přijde-li přerušení... (!!)

1. Přijde signalizace přerušení
2. Dokončena rozpracovaná strojová instrukce
3. Na zásobník je **uloženo stavové slovo procesoru (registr FLAGS)**
4. Je nastaven **zákaz přerušení** (změna bitu stavového slova), aby naši obsluhu nic dalšího nepřerušovalo (příklad - 2x stisknu klávesu)
5. Na zásobník **uložena adresa následující** instrukce, kterou chceme v daném procesu dále pokračovat
6. Z tabulky vektorů přerušení zjistí adresu podprogramu pro obsluhu přerušení
7. Obsluha – rychlá
 - Nejčastěji uložíme registry na zásobník a před koncem obsluhy je zas vybereme, abychom je procesu, který jsme přerušili nezměnili
 - Na konci stejný stav procesoru (hodnoty registrů) jako na začátku (pokud neslouží k předání výsledku)
8. Instrukce návratu IRET
 - Vyzvedne ze zásobníku návratovou adresu a stavové slovo (a tím i povolí přerušení, protože ve stavovém slovu je původní hodnota bitu přerušení)
 - Běh pokračuje na návratové adrese
- Přerušená úloha (mimo zpoždění) nepozná, že proběhla obsluha přerušení

Přerušení a výjimky vznikají a obsluhují se v reálném režimu téměř shodně s procesorem 8086. Rozlišuje se 256 různých přerušení a výjimek. Pro každé přerušení nebo výjimku je v paměti uloženo 32 bitů adresy (přerušovací vektor) začátku obslužné programové rutiny. Adresy jsou zapsány v **tabulce přerušovacích vektorů** (viz obr. 4.2). Tabulka má velikost 1 KB a je implicitně uložena na začátku paměti od adresy 0000:0000.

| 31 | 0 | Adresa | Číslo přer. vektoru |
|----------------|---------------|--------|---------------------|
| <i>segment</i> | <i>offset</i> | 0:03FC | INT 0FFh |
| | ⋮ | ⋮ | |
| <i>segment</i> | <i>offset</i> | 0:000C | INT 3 |
| <i>segment</i> | <i>offset</i> | 0:0008 | INT 2 |
| <i>segment</i> | <i>offset</i> | 0:0004 | INT 1 |
| <i>segment</i> | <i>offset</i> | 0:0000 | INT 0 |

Obr. 4.2 Tabulka přerušovacích vektorů reálného režimu

Průběh obsluhy přerušení

Po přijetí žádosti o přerušení provádí procesor v reálném režimu tyto akce:

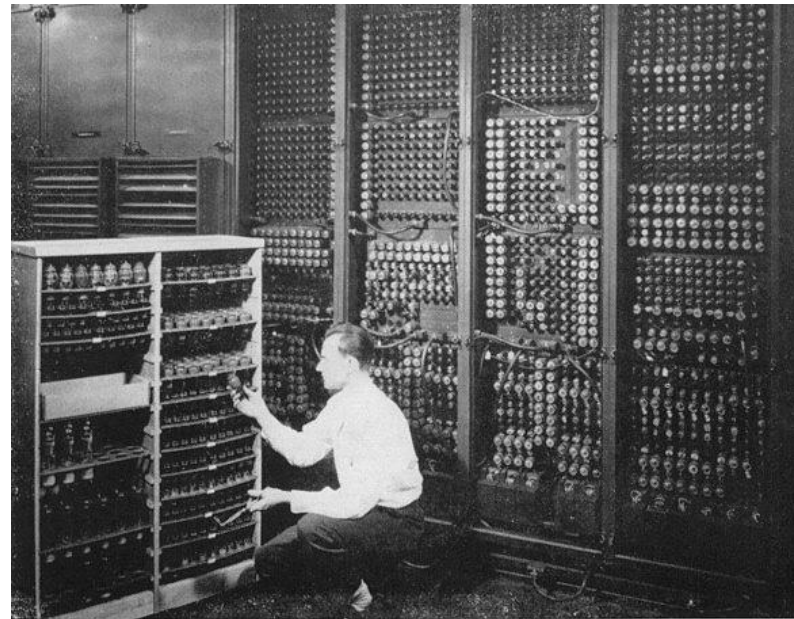
1. do zásobníku se uloží registr příznaků (FLAGS),
2. vynulují se příznaky IF a TF,
3. do zásobníku se uloží registr CS,
4. registr CS se naplní 16bitovým obsahem adresy $n \times 4 + 2$,
5. do zásobníku se uloží registr IP ukazující na neprovedenou instrukci,
6. registr IP se naplní 16bitovým obsahem adresy $n \times 4$.

Výjimky v reálném režimu nevracejí chybový kód. Návrat do přerušeného procesu a jeho pokračování zajistí instrukce IRET, která provede činnosti v tomto pořadí:

1. ze zásobníku obnoví registr IP,
2. ze zásobníku obnoví registr CS,
3. ze zásobníku obnoví příznakový registr (FLAGS).

Historický vývoj

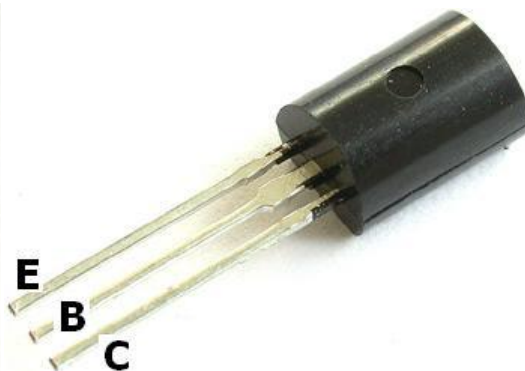
- Vývoj HW -> vývoj OS
- 1. počítač – **ENIAC**, 15.2.1946
 - Tělocvična
 - 18 000 elektronek
 - Regály, chlazení
 - 5000 operací/s



Replacing a bad tube meant checking among ENIAC's 19,000 possibilities.

Generace počítačů

1. Elektronky
2. Tranzistory
3. Integrované obvody
4. LSI, VLSI (mikroprocesory,..)



1. Generace (1945-55)

- Elektronky, propojovací desky
- Programování
 - V absolutním jazyce
 - Propojování zdířek na desce
 - Později děrné štítky, assembly, knihovny, FORTRAN
 - Numerické kalkulace
- Způsob práce
 - **Stejní lidé** – stroj navrhli, postavili, programovali !
 - Zatrhnout blok času na rozvrhu, doufat, že to vyjde
- OS ještě neexistují

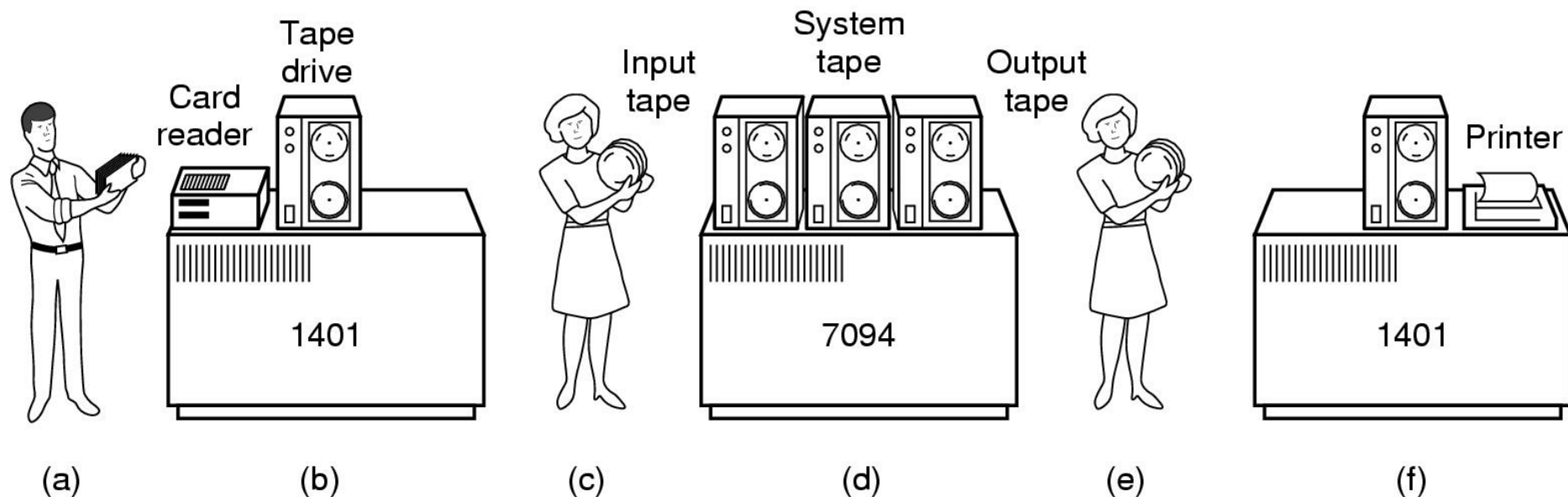
2. Generace (1955-65)

- Tranzistory, **dávkové OS**
- Vyšší spolehlivost; klimatizované sály
- **Oddělení** návrhářů, výroby, operátorů, programátorů, údržby
- Miliony \$ - velké firmy, vlády, univerzity
- Způsob práce
 - Vyděrovat štítky s programem
 - Krabici dát operátorovi
 - Výsledek vytisknut na tiskárně
- Optimalizace
 - Na **levném stroji** štítky přenést na magnetickou pásku

2. generace

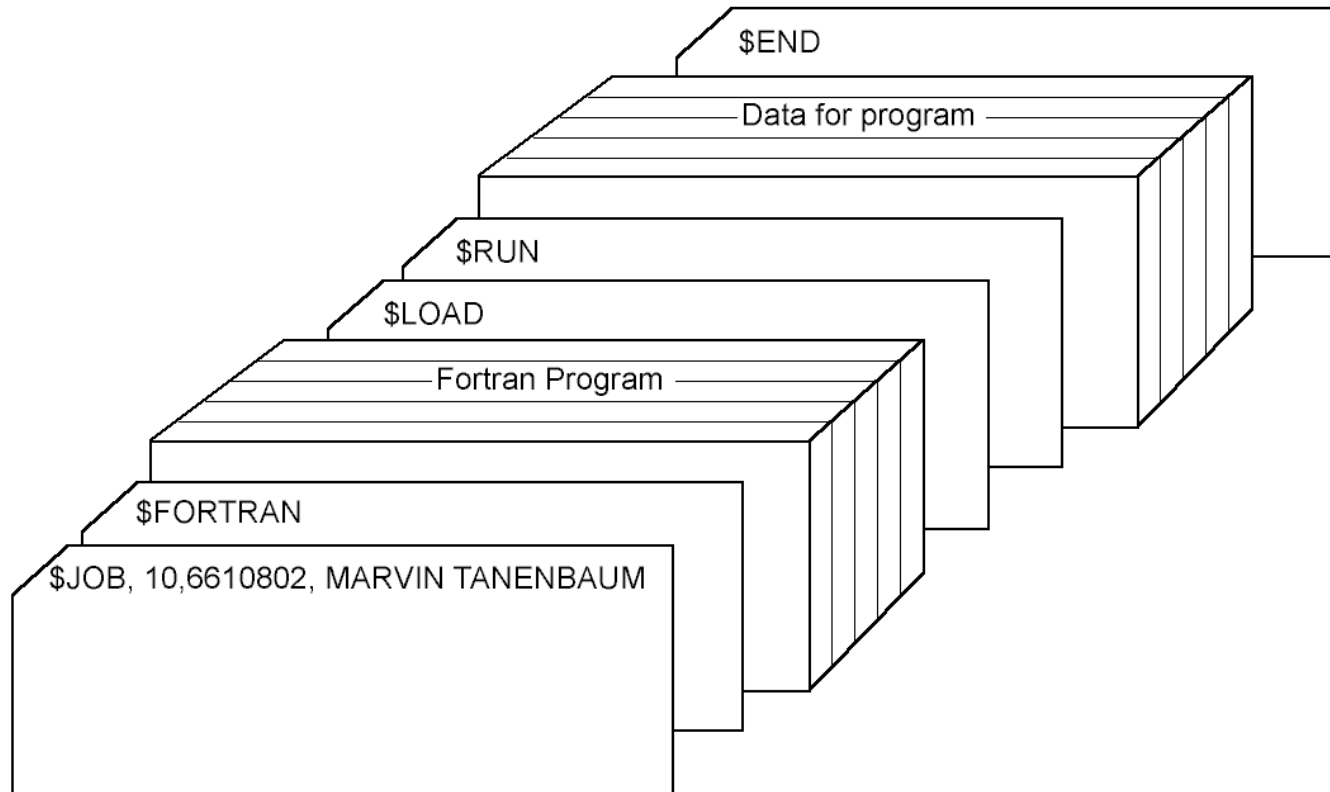
- Sekvenční vykonávání dávek
- Ochrana systému – kdokoliv dokázal shodit
- OS IBSYS = IBM SYSTEM FOR 7094
- Pokud úloha prováděla I/O, CPU čekal..
 - Čas CPU je drahý
- Viz slidy Tanenbaum

Dávkové systémy



- vezmi štítky k 1401
- štítky se zkopírují na pásek (tape)
- pásek na 7094 (drahý stroj), provádí výpočet
- output tape na 1401 vytiskne výstup

History of Operating Systems



- Struktura typické dávky – 2nd generation

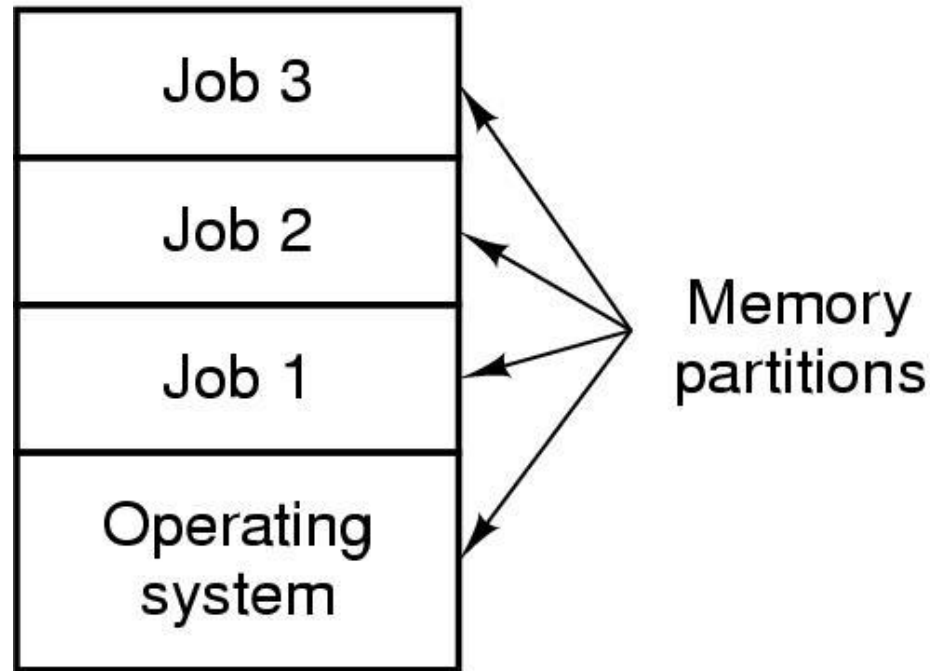
3. Generace (1965-80)

- Integrované obvody, **multiprogramování**
- Do té doby 2 řady počítačů
 - Vědecké výpočty
 - Komerční stroje – banky, pojišťovny
- IBM 360 – **sjednocení**
 - Malé i velké stroje
 - Komplexnost – spousta chyb

3. generace

- **Multiprogramování**
 - Doba čekání na I/O neefektivní (věda OK, banky 80-90% čekání)
 - Více úloh v paměti
 - Napřed konstantní počet
 - HW pro ochranu paměti
- Každá úloha ve vlastní oblasti paměti; zatímco jedna provádí I/O, druhá počítá ...

History of Operating Systems



- Multiprogramming system
 - three jobs in memory – 3rd generation

3. generace

- Spooling

- Na vstupu – ze štítků na disk, úloha se zavede z disku
- Na výstupu – výsledky na disk před výtiskem na tiskárně

spooling se dnes používá typicky pro sdílení tiskárny mezi uživateli
uživatel svůj požadavek vloží do tiskové fronty
až je tiskárna volná, speciální proces vezme požadavek z fronty a
vytiskne jej

- Stále dávkové systémy

- Dodání úlohy, výsledek – několik hodin

3. generace

- Systémy se sdílením času
(time shared system)
 - Varianta multiprogramování
 - CPU střídavě vykonává úlohy
 - Každý uživatel má on-line terminál
- CTSS (MIT 1962) *Compatible Time Sharing Sys.*
- MULTICS

Minipočítače

- DEC PDP (1961)
 - Cca 3.5 mil Kč , „jako housky“
 - Až PDP11 – nekompatibilní navzájem
- Výzkumník Bell Labs pracující na MULTICSu
Ken Thompson – našel nepoužívanou PDP-7,
napsal omezenou jednouživat. verzi MULTICSu
vznik UNIXu a jazyka C (1969)

4. Generace (1980)

- Mikroprocesory, PC
- GUI x CLI
- Síťové a distribuované systémy
- MS DOS, Unix, Windows NT
- UNIX – dominantní na nonIntel;
- Linux, BSD – rozšíření i na PC
 - Výzkum Xerox PARC – vznik GUI
 - Apple Macintosh
- film „Piráti ze Silicon Valley“

Dělení OS

Dle úrovně sdílení CPU:

- **Jednoprocesový**
 - MS DOS, v daném čase v paměti aktivní 1 program
- **Multiprocesový**
 - Efektivnost využití zdrojů
 - Práce více uživatelů

Dělení OS

Dle typu interakce:

- **Dávkový systém**
 - Sekvenční dávky, není interakce
 - Dodáme program a data, čekáme na výsledek
 - I dnes má smysl, viz. meta.cesnet.cz
- **Interaktivní**
 - Interakce uživatel – úloha
 - Víceprocesové – interakce max. do několika sekund (Winows, Linux, ..)

OS reálného času (!)

- Výsledek má smysl, pouze pokud je získán v nějakém omezeném čase
- Přísné požadavky aplikací na čas odpovědi
 - Řídící počítače, multimedia
- Časově ohraničené požadavky na odpověď
 - Řízení válcovny plechu, výtahu mrakodrapu ☺
- Nejlepší snaha systému
 - Multimedia, virtuální realita
- Př: RTLinux, RTX Windows, VxWorks

Hard realtime OS

- **Zaručena** odezva v **ohraničeném** čase
- Všechna zpoždění a režie systému ohraničeny

Omezení na OS:

- Často není systém souborů
- Není virtuální paměť
- Nelze zároveň sdílení času
- Řízení výroby, robotika, telekomunikace

Soft realtime OS

- Priorita RT úloh před ostatními
- Nezaručuje odezvu v daném čase
- Lze v systémech sdílení času
- RT Linux
- Multimédia, virtuální realita

Další dělení OS

- **Dle velikosti HW**
 - Superpočítač, telefon, čipová karta
- **Míra distribuovanosti**
 - Klasické - centralizované 1 a více CPU
 - Paralelní
 - Síťové
 - Na více uzlech sítě
 - Distribuované
 - Virtuální uniprocessor (tváří se jako jeden stroj)
 - Uživatel neví kde běží programy, kde jsou soubory

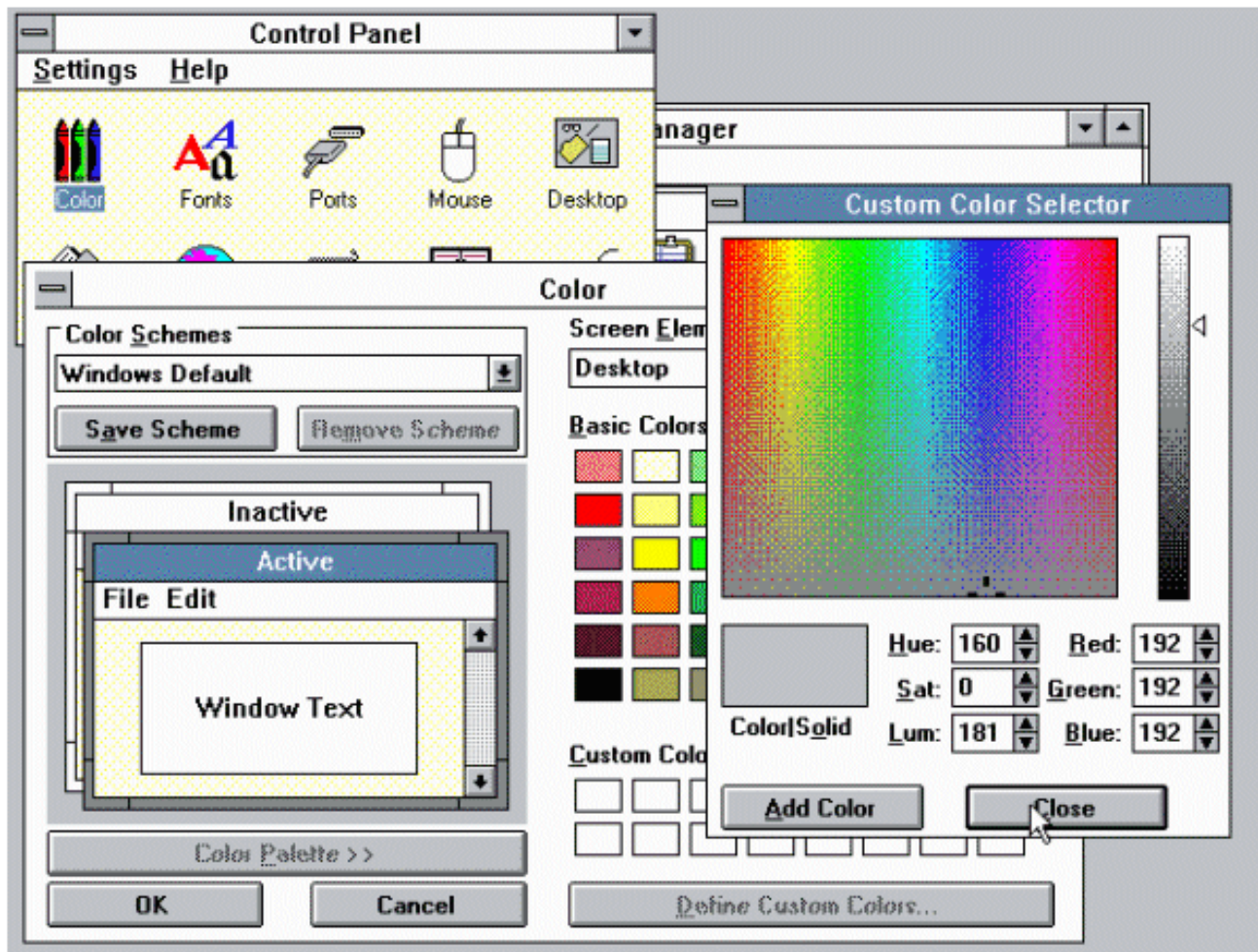
Další dělení OS

- Podle počtu uživatelů
 - Jedno a víceuživatelské
- Podle funkcí
 - Univerzální
 - Specializované (např. Cisco IOS)

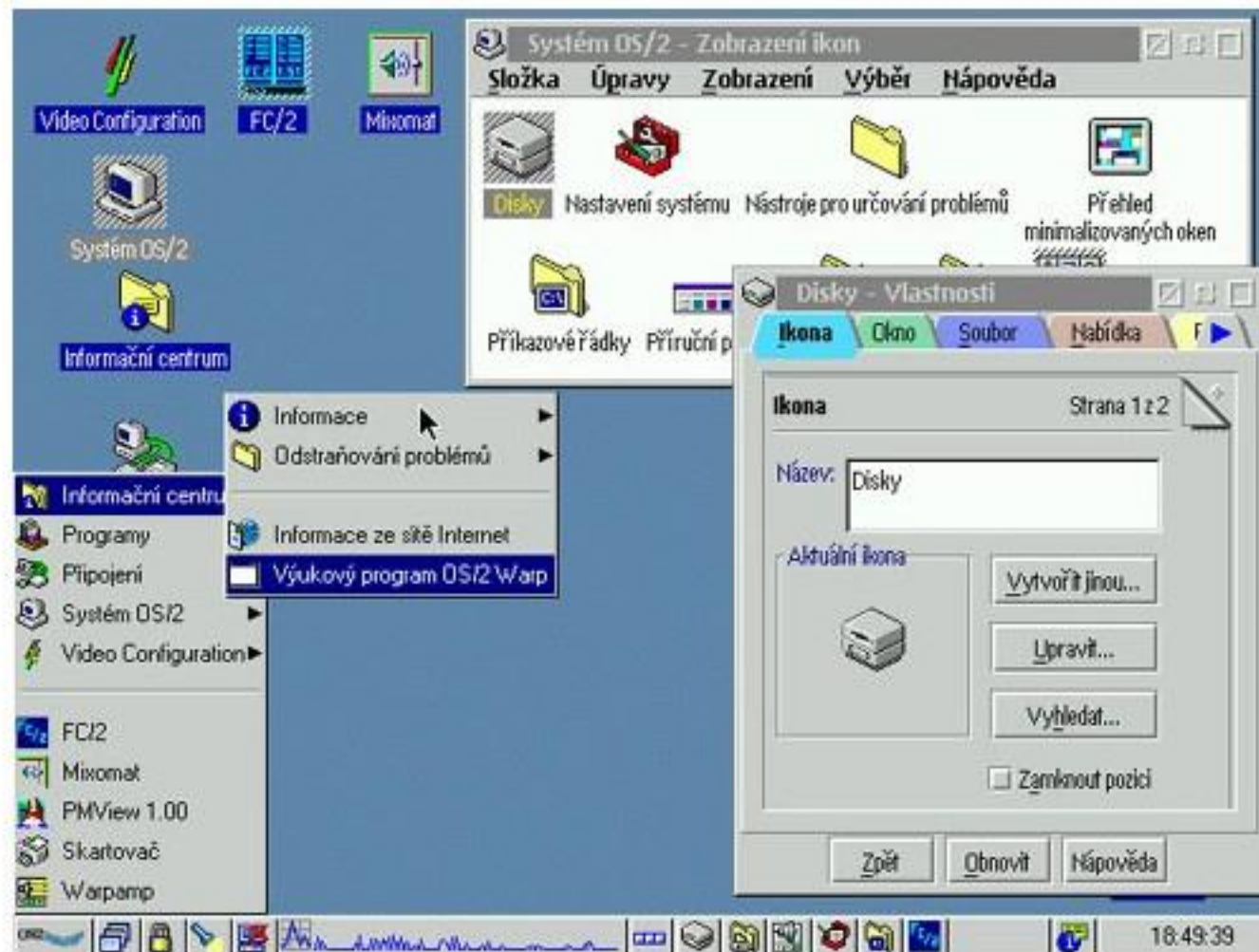
Uživatelské rozhraní

- uživatelské rozhraní
- CLI (command line interface)
- GUI (graphical user interface)
 - *ukázky z www.zive.cz*

Win 3.1



OS/2 Warp4



Architektury OS

OS = jádro + systémové nástroje

Jádro se zavádí do operační paměti při startu a zůstává v činnosti po celou dobu běhu systému

Základní rozdělení:

- **Monolitické jádro** – jádro je jeden funkční celek
- **Mikrojádro** – malé jádro, oddělitelné části pracují jako samostatné procesy v user space
- **Hybridní jádro** - kombinace

Architektura OS

Linux



| | |
|---|--|
| Rodina OS: | Unix-like |
| Aktuální verze: | 3.5 / 21. července 2012 |
| Podporované platformy: | IA-32, x86-64, PowerPC, ARM, m68k, DEC Alpha, SPARC, hppa, IA-64, MIPS, s390 a další |
| Typ kernelu: | Monolitické jádro |
| Implicitní uživatelské rozhraní: | GNOME, KDE, Xfce a jiné |
| Licence: | GNU GPL a jiné |
| Stav: | Aktuální |

Windows 10



| | |
|-------------------------------------|--|
| Web | http://www.microsoft.com/cs-cz/windows |
| Vyvíjí | Microsoft |
| Rodina OS | Windows NT |
| První oznámení | říjen 2014 |
| První vydání | 29. července 2015 ^[1] |
| Aktuální verze | 1903, build 18362.267 ^[2] / 26. července 2019 |
| Způsob aktualizace | Windows Update, Windows Store, Windows Server Update Services, System Center Configuration Manager |
| Správce balíčků | OneGet |
| Typ jádra | Hybridní jádro |
| Výchozí uživatelské rozhraní | GUI |
| Licence | Proprietární Microsoft EULA |
| Stav | V prodeji |
| Podpora do | všeobecná do 13. října 2020 rozšířená do 14. října 2025 ^[3] |

Mac OS X



| | |
|---|--|
| Web: | www.apple.com/macosx/ |
| Vyvíjí: | Apple Inc. |
| Rodina OS: | BSD |
| Druh: | Uzavřený vývoj (s využitím open source komponent) |
| Aktuální verze: | 10.8 / 19. července 2012 |
| Podporované platformy: | x86, x86-64, PowerPC (32bitový i 64bitový) |
| Typ kernelu: | Hybridní jádro |
| Implicitní uživatelské rozhraní: | Aqua |
| Licence: | Apple SLA (část pod APSL) |
| Stav: | Aktivní |

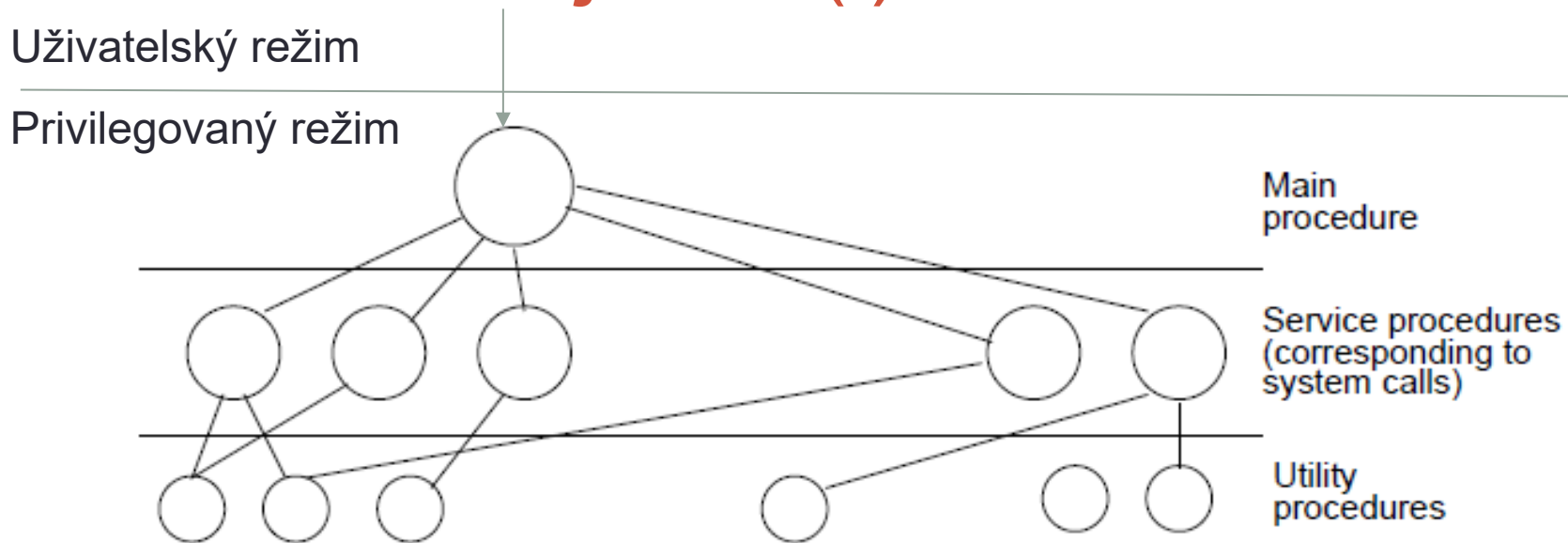
Monolitické jádro

- Jeden spustitelný soubor
- Uvnitř moduly pro jednotlivé funkce
- Jeden program, řízení se předává voláním podprogramů
- Příklady: UNIX, Linux, MS DOS

Typickou součástí monolitického jádra je
např. souborový systém

Linux je monolitické jádro OS, s podporou zavádění
modulů za běhu systému

Monolitické jádro (!)



Main procedure – vstupní bod jádra, na základě čísla služby (např. v EAX) zavolá servisní proceduru

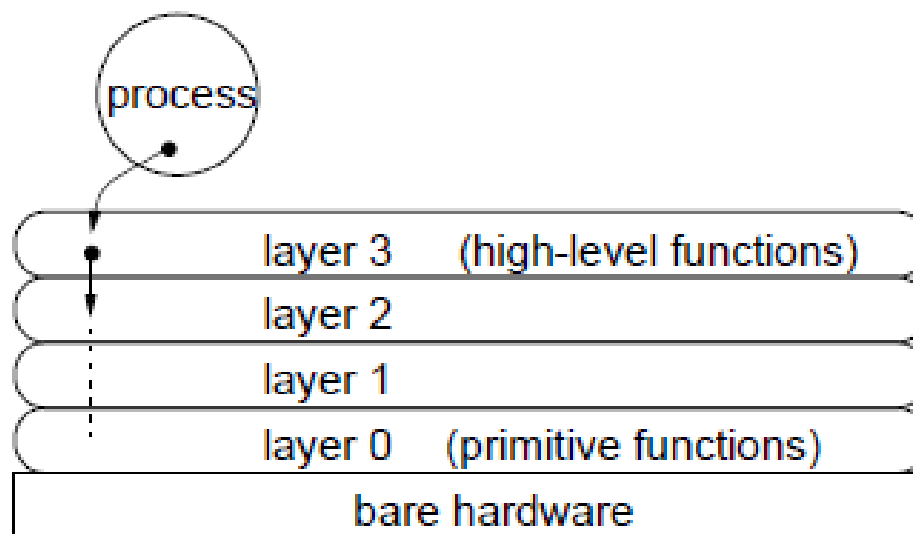
Service procedure – odpovídá jednotlivým systémovým voláním (zobrazení řetězce, čtení ze souboru, vytvoření procesu aj.)

Service procedure volá pro splnění svých cílů různé **pomocné utility procedures** (lze je opakovaně využít v různých voláních)

Vrstvené jádro

- Výstavba systému od nejnižších vrstev
- Vyšší vrstvy využívají primitiv poskytovaných nižšími vrstvami
- Hierarchie procesů
 - Nejnižše vrstvy komunikující s HW
 - Každá vyšší úroveň poskytuje abstraktnější virtuální stroj
 - Může být s HW podporou – pak nelze vrstvy obcházet (obdoba systémového volání)
- Příklady: THE, MULTICS

Vrstvené jádro



OS THE:

Úroveň 0 .. Virtualizace CPU (přepínání mezi procesy)

vyšší vrstva už předpokládá existenci procesů

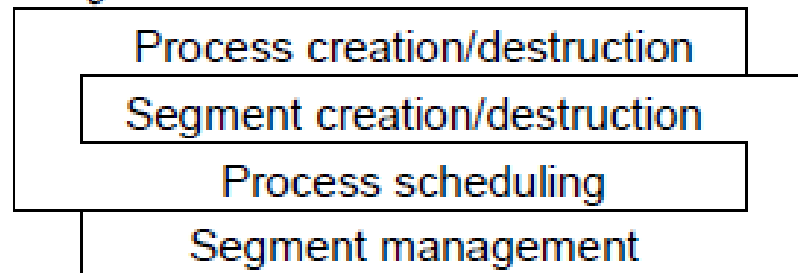
Úroveň 1 .. Virtualizace paměti

vyšší vrstvy už nemusí řešit umístění části procesů v paměti

Funkční hierarchie

- Problém jak rozčlenit do vrstev dřív slepice, než vejce
 - správa procesů vs. správa paměti
- Některé moduly vykonávají více funkcí, mohou být na více úrovních v hierarchii
- Př: Pilot

Process management



Memory management

Mikrojádro (!)

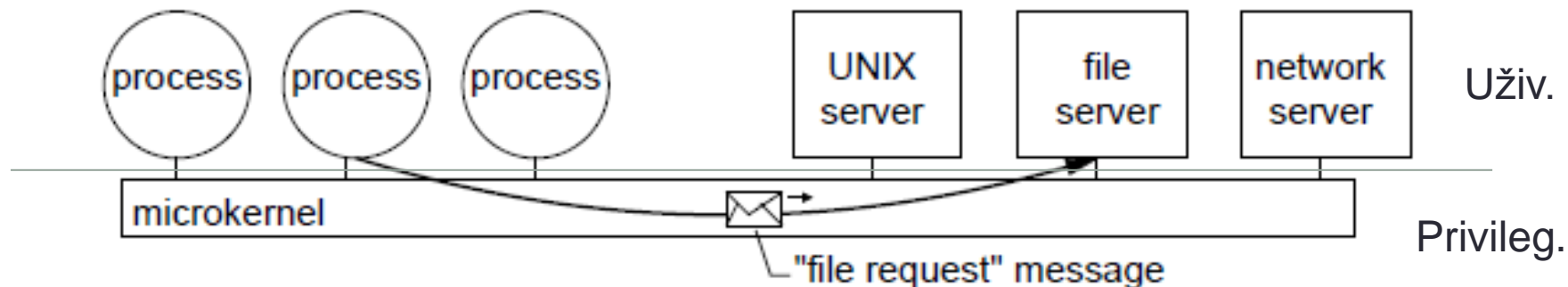
- Model **klient – server**
- Většinu činností OS vykonávají **samostatné procesy mimo jádro** (servery, např. systém souborů)
- Mikrojádro
 - Poskytuje pouze nejdůležitější nízkoúrovňové funkce
 - Nízkoúrovňová správa procesů (vytvoř proces, vlákno, ...)
 - Adresový prostor, komunikace mezi adresovými prostory
 - Obsluha přerušení, vstupy/výstupy
 - Pouze mikrojádro běží v privilegovaném režimu
 - Méně pádů systému

Činnosti vyžadující privilegovaný režim je stále nutné provést v mikrojádře – pouze to je v privilegovaném režimu CPU

Mikrojádro

- Výhody
 - Pád serveru nemusí znamenat pád jádra
 - Snadnější tvorba distribuovaných OS (komunikace přes síť), ale často běží i na 1 uzlu
- Nevýhody
 - Větší režie (4 x přepnutí uživatelský režim \leftrightarrow jádro)
 - Složitější návrh systému
- Potřebujeme nejen mikrojádro, ale i servery
 - Mikrojádro Mach + kolekce serverů Hurd = ekvivalent jádra
- Příklady: QNX, Hurd, OSF/1, MINIX, Amoeba

Mikrojádro



Mikrojádro – základní služby, běží v privilegovaném režimu

1. proces vyžaduje službu
2. mikrojádro předá požadavek příslušnému serveru
3. server vykoná požadavek

Snadná vyměnitelnost serveru za jiný

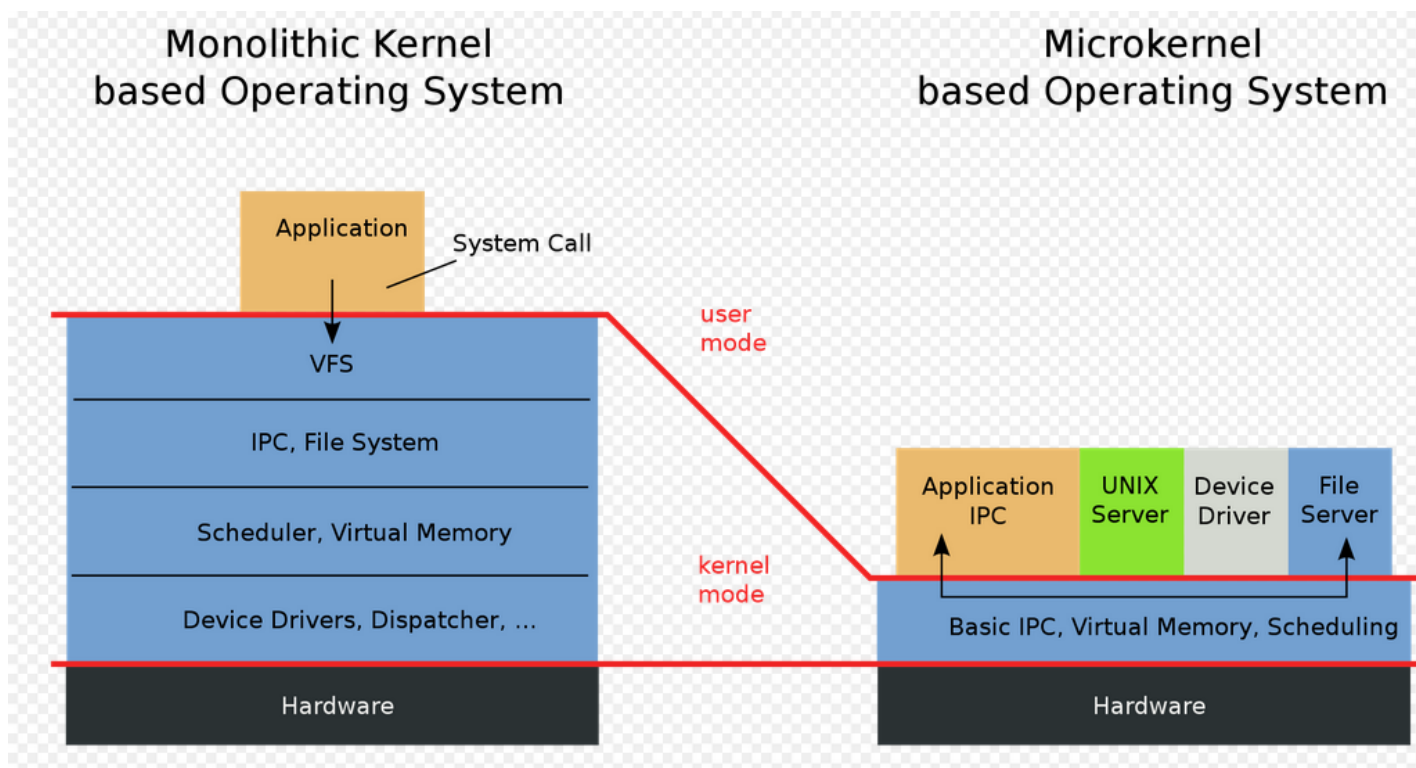
Chyba serveru nemusí být fatální pro celý operační systém
(není v jádře) – nepadne celý systém

Distribuované systémy - server může běžet i na jiném uzlu sítě

Mikrojádro

- Mikrojádro musí stejně nadále některé funkce vykonávat
 - Vše, kde jsou potřeba **privilegované instrukce** (IN, OUT)
 - Správa procesů a vláken
 - Správa adresního prostoru

Mikrojádرو

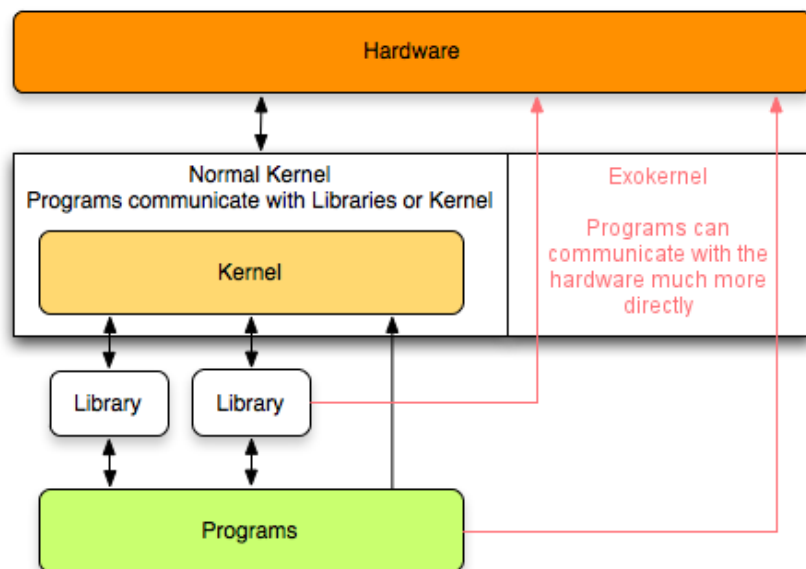


Zdroj obrázku: wikipedia

V kernel mode běží jen nutné minimum činností

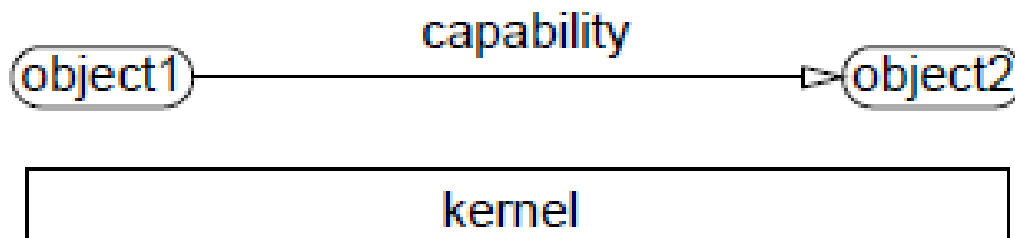
Exokernel

- Menší než klasické monolitické jádro
- Jádro alokuje diskové bloky, paměťové stránky, čas CPU
- Více přímého přístupu k HW



Objektově orientovaná struktura

- Systém je množina **objektů** (soubory, HW zařízení)
- Capability = odkaz na objekt + množina práv definujících operace, spravuje jádro
- Jádro si vynucuje tento abstraktní pohled
- Jádro kontroluje přístupová práva
- Př: částečně Windows (NT,2000, XP, 7..) - hybridní



Hybridní jádro

- Kombinuje vlastnosti monolitického a mikrojádra
- Část kódu součástí jádra (monolitické)
- Jiná část jako samostatné procesy (mikrojádro)
- Příklady
 - Windows 11,10, 7
(NT, Win 2000, Win XP, Windows Server 2003, Windows Vista,..)
 - Windows CE (Windows Mobile)
 - BeOS

GNU/Linux, GNU/Hurd

GNU/Linux

- GNU programy, např. gcc (R. Stallman)
- **Monolitické** jádro OS – Linux (Linus Torvald)

GNU/Hurd

- GNU programy, např. gcc
- **Mikrojádru** – mikrojádru Mach a množina serverů Hurd běžících nad mikrojádrem



Linux

- Pojem Linux jako takový označuje jádro operačního systému
- Pokud hovoříme o operačním systému, správně bychom měli říkat **GNU/Linux**, ale toto přesné označení používá jen málo distribucí, např. *Debian GNU/Linux*
- Flame war – debata mezi Linusem Torvaldsem a Andrewem Tanenbaumem, že Linux měl být raději mikrokernel

Linux - odkazy

Interaktivní mapa Linuxového jádra:

<https://makelinux.github.io/kernel/map/>

Jaké jádro je nyní aktuální? (např. 5.14.6)

<http://kernel.org/>

Spuštění operačního systému bez instalace:

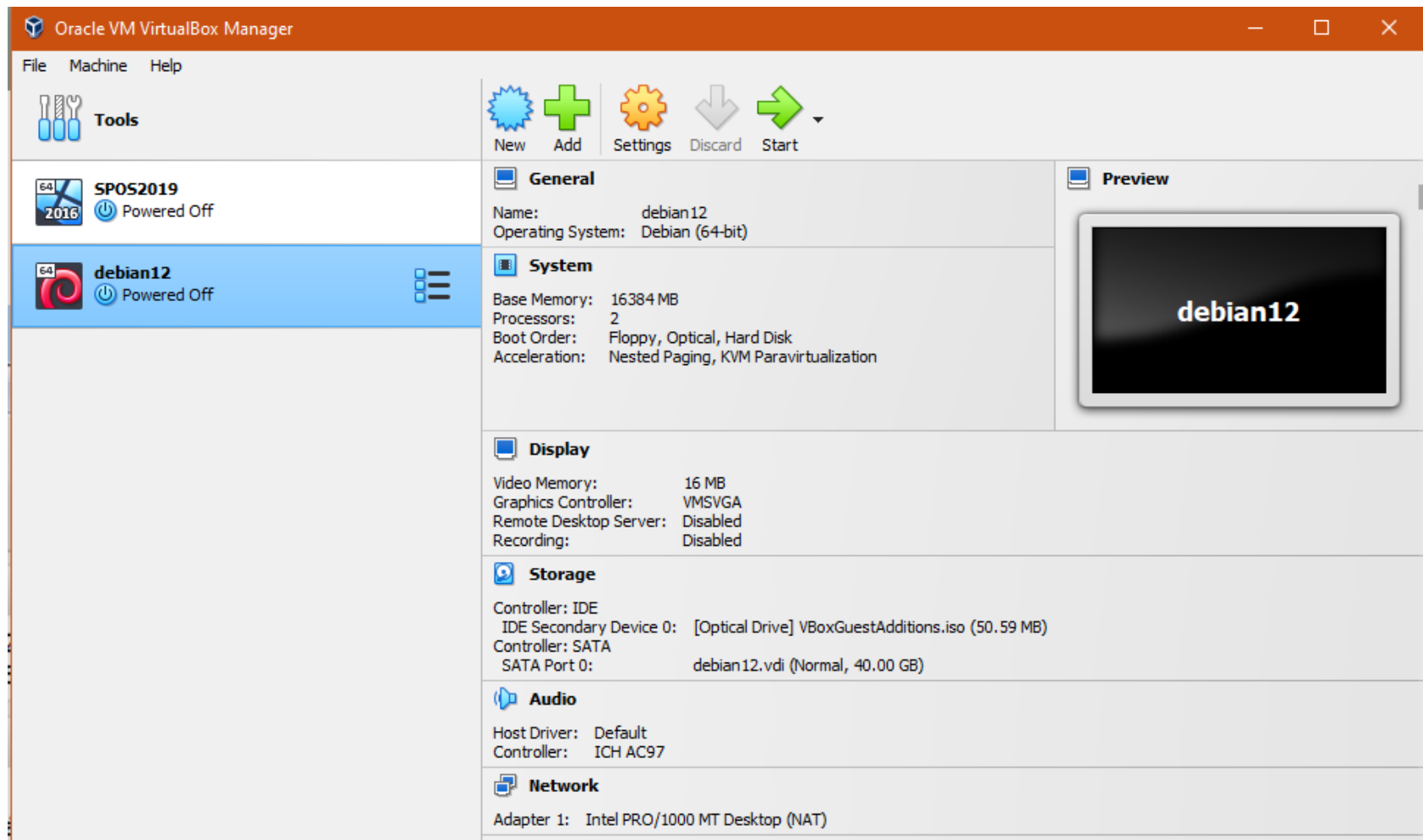
Live CD, Live DVD, Live USB

Např. Knoppix (<http://knoppix.org/>)

Virtualizace

- Možnost nainstalovat si virtuální počítač a provozovat v něm jiný operační systém, včetně přístupu k síti aj.
- Hypervisor běží buď na fyzickém železe, nebo na nainstalovaném OS
- VirtualBox
- VmWare
- Xen (např. virtualizace serverů), KVM aj.
- Hyper-V

VirtualBox



Literatura, použité zdroje

Obrázky z některých slidů pocházejí z knížky

[Andrew S. Tanenbaum: Modern Operating Systems](#)

vřele doporučuji tuto knihu, nebo se alespoň podívat na slidy ke knize dostupné mj. na webu předmětu v *Přednášky -> Odkazy*