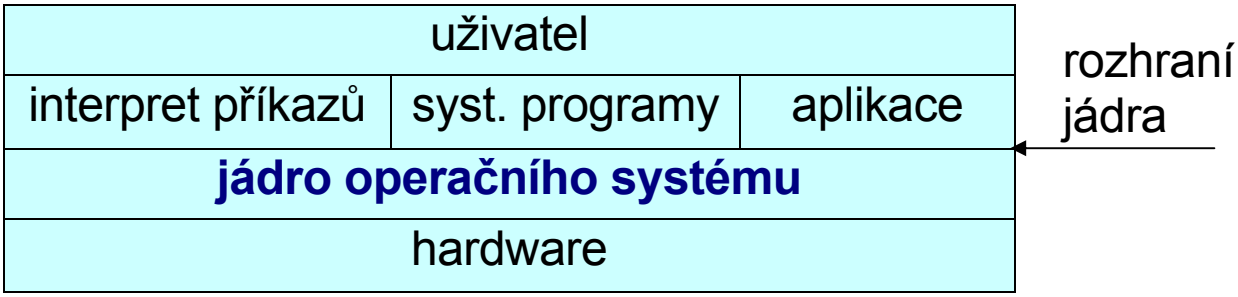
1. Základní pojmy

Operační systém (OS)

- most mezi hardwarem a běžícím programem
- most mezi hardwarem a uživatelem
- vytváří prostředí pro běh programů (procesů, vláken)



hardware - přečti instrukci, inkrementuj PC, proveď instrukci
uživatel - abstrakce souborů, programů, síť. spojení, GUI

Proč je třeba OS?

uživatel - uživatelské prostředí = shell, systém souborů, nástroje
programátor - práce na vyšší úrovni abstrakce než holý hardware = V/V, procesy, synchronizace, alokace prostředků, ochrana
obecně - efektivita - zajištění plného využití všech prostředků výpočetního systému

Koncepční pohled na OS:

1. **Abstrakce –** nutná pro zvládnutí složitosti, skrývá složitost přístupu k hardware na nejnižší úrovni, detaily architektury a implementace.

   Jádro operačního systému poskytuje základní množinu operací na vyšší úrovni než jsou instrukce – rozšiřuje instrukční repertoár (*extended machine*), vytváří abstraktní model funkce hardware (*open/read/write*).

Typické operace jádra (POSIX):

- vytváření procesů, běh procesů - *fork()*, *execl()*, *waitpid()*
- přidělování paměti, správa paměti - *mmap()*, *break()*
- vstup/výstup - *read()*, *write()*
- soubory - *open()*, *read()*, *write()*, *close()*, *chmod()*, *chown()*

2. Virtualizace – sdílení prostředů (na úrovni procesů, uživatelů)

Běh procesů v režimu sdílení času (*time sharing*) procesoru vytváří iluzi vlastního procesoru pro každý proces, podobně virtualizace paměti poskytuje každému procesu iluzi vlastního adresového prostoru (vlastní paměti), atd.

3. Správa prostředků – maximalizace využití, bezpečnost.
OS = správce sdílených hardwarových prostředků:

1. procesor,
2. paměť,
3. V/V

Operační systém musí tyto prostředky přidělovat tak, aby byly maximálně využity.

Příklad: v době čekání procesu na provedení V/V operace může systém přidělit procesor jinému procesu. K tomu ale musí být tento proces v paměti, takže operační systém se musí snažit udržovat v paměti všechny spustitelné procesy.

Operace s prostředky:

- přidělování
- evidence přidělení a využití
- ochrana proti nesprávnému použití
- odebírání, uvolňování
- řešení výjimek a chybových stavů

Služby operačního systému:

- operace jádra (rozšíření instrukčního repertoáru)
- uživatelské rozhraní – interpret příkazů (shell), GUI
- autentizace, účtování
- správa služeb (spouštění, monitorování, restart)
- systémové služby (tisk, zálohování, správa systému, atd.)
- síťové služby (souborové, DNS, DHCP, Web, DB, atd.)

Operační systém = jádro + systémové programy

Otázka: které služby musí být v jádru? Co mimo jádro?

Různé přístupy:
- vše v jádru (starší dávkové systémy, MS-DOS)
- minimum v jádru, vše ostatní mimo (mikrojádro)
- v jádru je to, co tam musí být z důvodů efektivity (Unix)

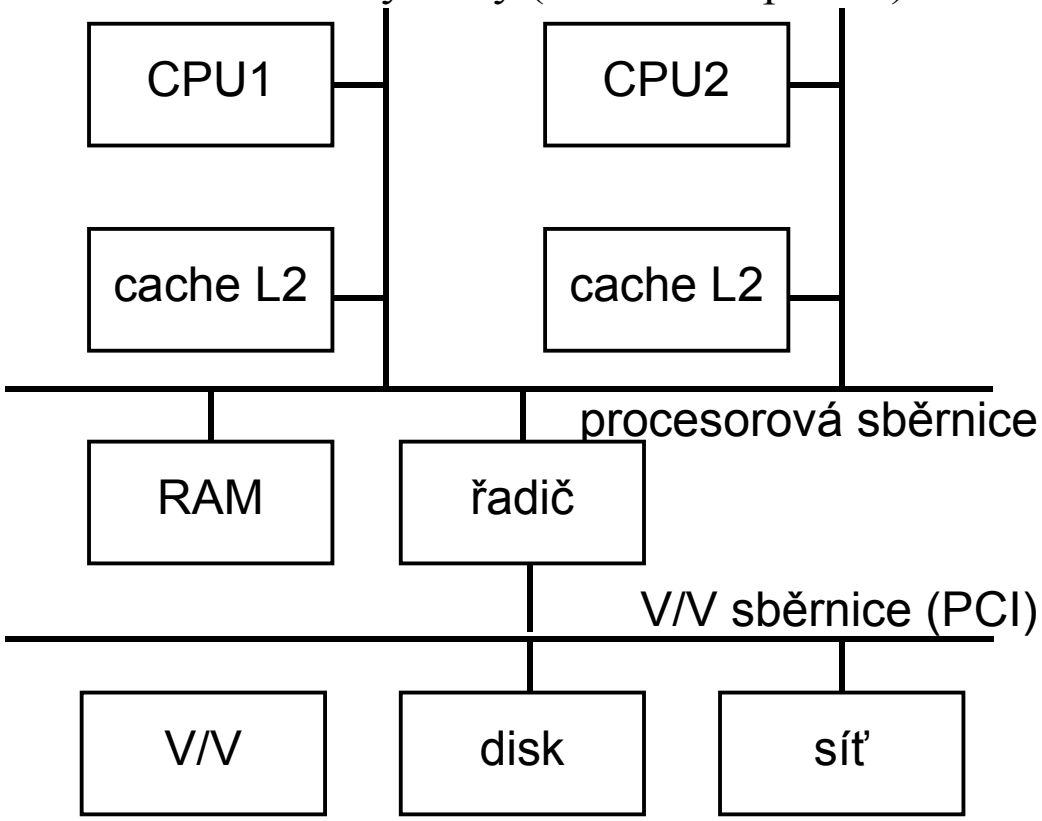
Co je to jádro OS?
- Složitý paralelní program, který je zaveden do paměti při startu počítače a řídí činnost hardware počítače.
- Uživatelské a systémové programy běžící vně jádra si můžeme představit jako podprogramy volané z jádra.
- Hlavní kritéria návrhu jádra:
  - efektivita využití hardware
  - spolehlivost
  - bezpečnost
- Hlavní komplikace návrhu jádra:
  - asynchronní (nedeterministický) paralelní běh různých částí jádra volaných z rozhraní jádra,
  - asynchronní (nedeterministický) vnější zdroj přerušení spouštějící paralelní běh různých částí jádra.

Architektura počítačů

- jeden nebo více procesorů,
- V/V řadiče a procesory komunikují každý s každým (sběrnicí nebo křížovým přepínačem) nebo omezeně (V/V pouze jeden),
- procesory a V/V řadiče pracují nezávisle a paralelně.

Počet procesorů a organizace paměti (synchronizace):

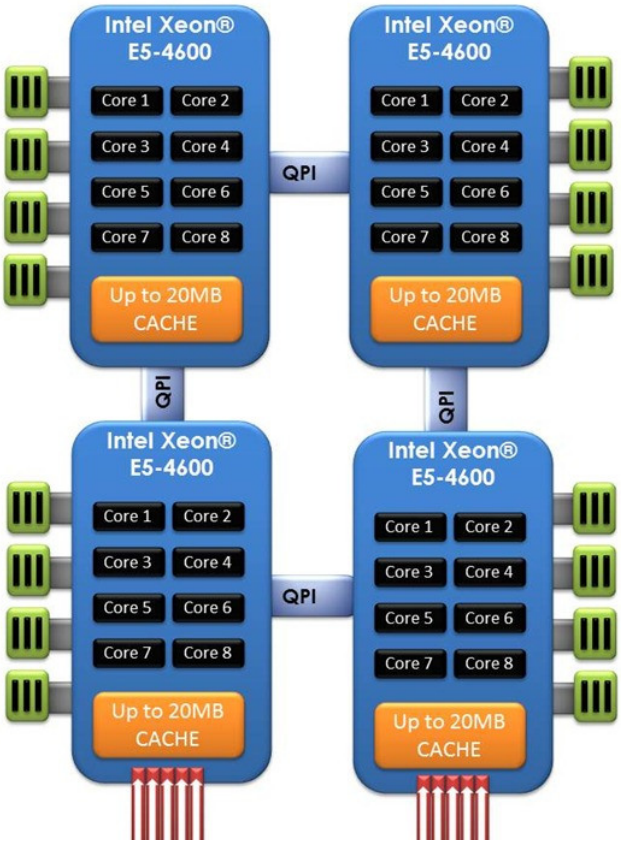
- jednoprocesorové
- víceprocesorové:
  - jedna sdílená paměť (UMA),
  - rozprostřená sdílená paměť (NUMA),
  - distribuované systémy (bez sdílené paměti) – zde ne!



Problém: zatížení procesorové sběrnice roste lineárně s počtem procesorů (prakticky max. 8)

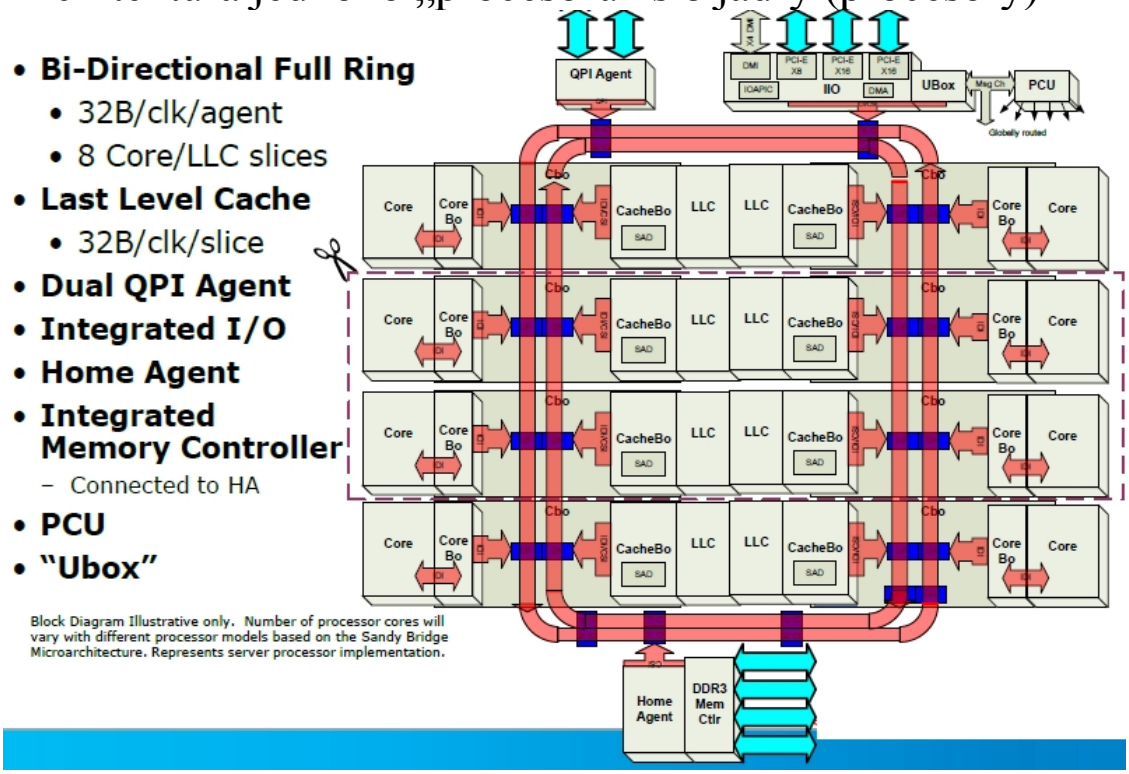
⇒ náhrada sběrnice křížovým přepínačem (Sun, SGI) nebo

⇒ rozprostření operační paměti (NUMA), každý procesor (skupina procesorů) má přímo přístupnou lokální paměť, přístup k lokální paměti jiných procesorů je pomalejší, ale je **zajištěna konzistence obsahu všech pamětí na úrovni hardware**.



Dnešní 4S(ocketový) server, do socketu je zasunut „procesor“ obsahující 4-32 jader procesorů (zelené jsou paměti, 4 paměťové kanály, červené PCI-Express), propojení vysokorychlostní plně duplexní sběrnicí

Architektura jednoho „procesoru“ s 8 jádry (procesory)



Zůstává zachován princip – společná paměť!

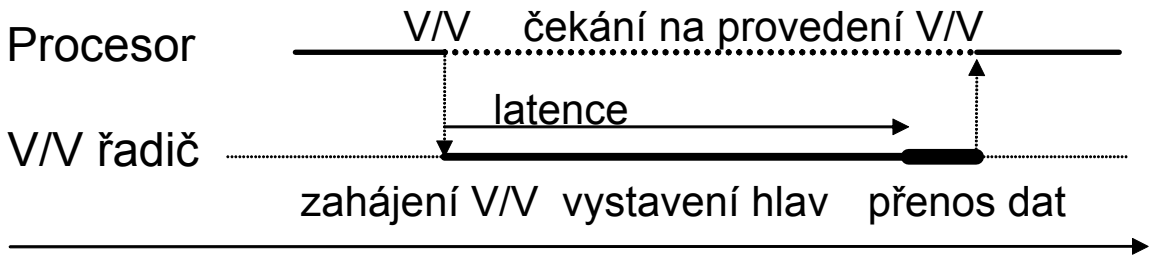
Problém efektivního využití zdrojů

	latence	**přenosová rychlost (P4)**
ALU (registry)	< 1 ns	>1000 GB/s
cache L1	< 1 ns	500 GB/s
cache L2	2-5 ns	300 GB/s
RAM	40ns	50-100 GB/s
disk	3-7 ms	200-500 MB/s
síť	ms	10-80 MB/s (Gigabit Ethernet)
uživatel	s	0,1-4 B/s

Rozdíly v rychlosti 10 řádů!
Přenosová rychlost není jediným faktorem, větší problém je často latence.

Příklad: Správa V/V – disková operace čtení
1. OS zahájí V/V operaci, případně předá data (zápis)
2. řadič provádí operaci (čtení z disku, odeslání rámce, apod.)
3. řadič zašle výsledek operace (zápis) a případně data (čtení)

Jaká skutečná rychlost?



latence = doba od zaslání příkazu do zahájení přenosu dat
přenosová rychlost = rychlost přenosu dat z/na médium
délka bloku = délka dat při jedné V/V operaci

$$efektivní\_rychlost = \frac{délka\_bloku}{latence + \frac{délka\_bloku}{přenosová\_rychlost}}$$

Příklad pevný disk:
latence 5 ms, přenosová rychlost 40 MB/s, délka bloku 4 KB
efektivní = 4 KB/(0,005+0,0001) = 800 KB/s - kde je chyba?

Příklad disk SSD:
interní struktura flash paměti – stránky 2-32KB (čtení, zápis), bloky velikosti 64KB-512KB (mazání)
externí přístup – zápis sektorů 512B
Co s tím?

- radikální řešení, změna velikosti diskového bloku
- kompatibilní řešení, inteligentní emulace

Kontrolní otázka: jak rychlý bude zápis náhodných sektorů u SSD?

Redukce latence v zařízení:

1. Minimalizace režie – doba zpracování, interpretace příkazů (nezapočítali jsme, může být řádově ms)
2. Zkracování doby vybavení informace (přístupová doba) – např. cache na sekvenčně čtená data z jedné stopy disku, nelze redukovat u náhodných přístupů

Řešení:

1. Čtení dopředu, zápis v pozadí (zůstává latence režie zpracování příkazů při sériovém provádění) – musí řešit OS
2. Paralelně zpracovávat více příkazů současně (pipeline): odstraní se latence režie, zařízení může optimalizovat přístupovou dobu (sekvenční čtení, rotační zpoždění), např. SCSI disky (Tagged Command Queueing), SATA-II (NCQ) – opět musí řešit OS

Typy procesů: I/O bound (převážně čeká na V/V), CPU bound (převážně počítá). Interaktivní uživatelské procesy jsou většinou I/O bound.

Co má dělat operační systém v době provádění V/V?

1. Může aktivně čekat na výsledek - programové řízení V/V (testování flagu) ⇒ procesor bude většinu doby nevyužit, V/V zařízení nemůže zpracovávat paralelně příkazy, protože nebudou žádné další k dispozici.
2. Může dělat něco užitečného (provádět jiný program, kód jádra, výsledkem mohou být další V/V operace na zařízení).

Podmínka 2.: V/V řadič musí mít možnost signalizovat ukončení operace a přerušit běh procesoru (aby se proces čekající na dokončení V/V někdy dočkal výsledku) ⇒ nutný **přerušovací systém!**

Přerušovací systém a jeho vztah k návrhu jádra

Zpracování přerušení (interrupt):

1. dokončení právě prováděné instrukce
2. zablokování dalších přerušení (všech nebo menší priority)
3. uložení stavu procesoru (registry, stav, adresový prostor)
4. skok do obslužného podprogramu přerušení v systémovém (privilegovaném) režimu činnosti procesoru
5. obsluha přerušení (může být přerušena přerušením vyšší prio)
6. obnova stavu procesoru (pokračování v přerušené práci)

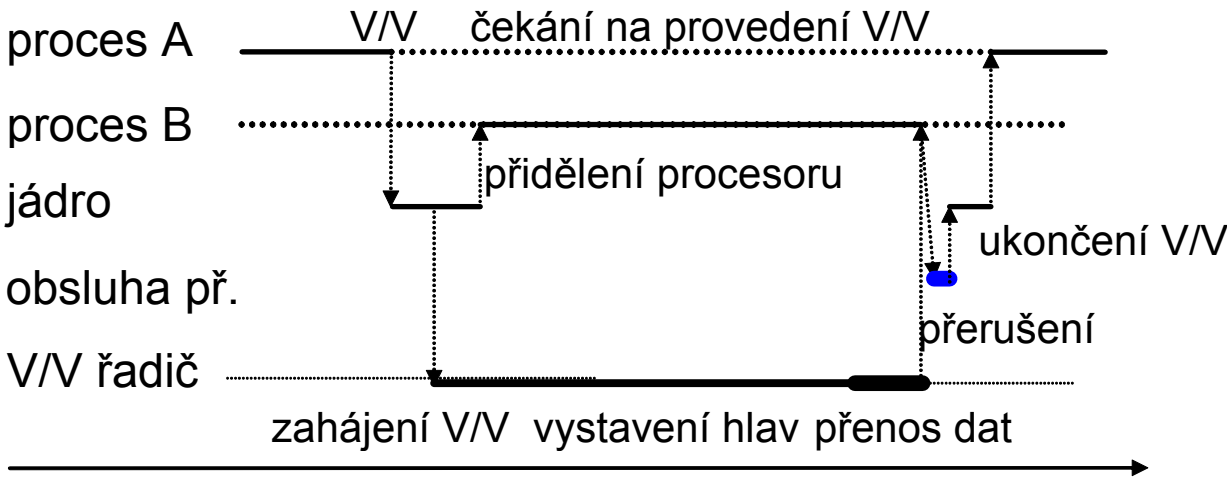
ad 5) Možnosti:

Klasické jádro - podprogram obsluhy přerušení má speciální postavení z hlediska synchronizace. Pokud běželo v bodě 1 jádro, nemůže podprogram obsluhy přerušení volně manipulovat s datovými strukturami jádra – nutná synchronizace přístupu k datovým strukturám používaným z podprogramu obsluhy přerušení a jádra samotného (*spl*, *spinlock*). Tato synchronizace nesmí dlouhodobě blokovat běh (podprogram obsluhy přerušení nemůže čekat na semafor).

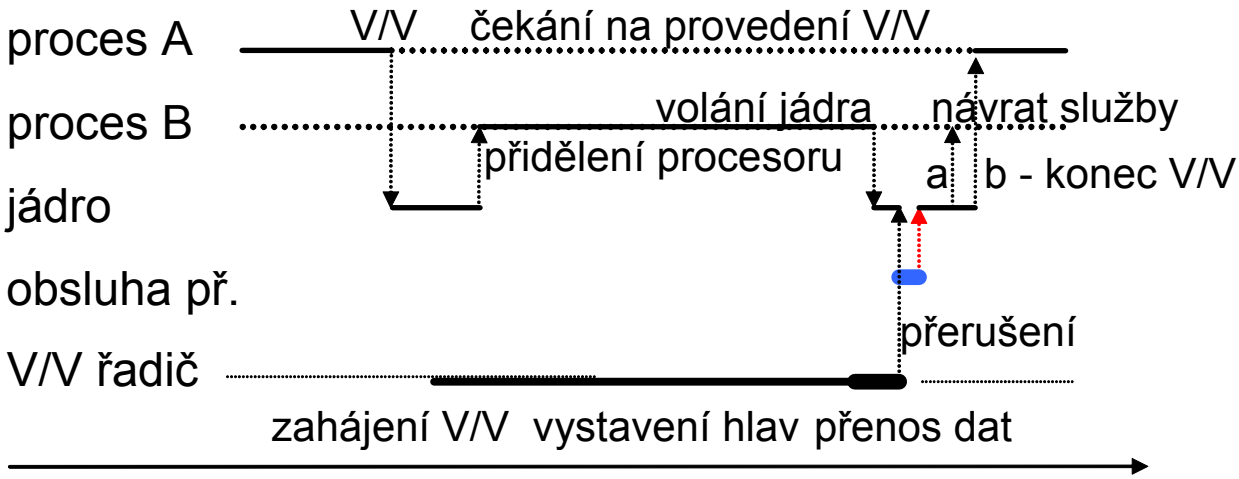
Koncepce mikrojádra - obsluha přerušení ve formě procesu – přerušení pouze odblokuje odpovídající proces v jádře (na odblokování platí podmínky viz výše), ten dále používá normální synchronizaci uvnitř jádra.

ad 6) Návrat do místa přerušení:

a) návrat do uživatelského režimu – není problém, struktury jádra nejsou používány, lze před návratem dělat cokoli (dokončit V/V, aktivovat čekající proces)



b) návrat do systémového režimu = byl přerušen běh jádra systému, kam se vrátit?



- do přerušeného místa → nelze na základě přerušení spustit nějakou aktivitu v jádře, přerušení pouze nastaví stav procesu a příznak čekající aktivity, ten se později testuje (v klasickém Unixu před návratem do uživatelského režimu), **nepreemptivní jádro**
- jinam = **lze pozastavit přerušenou rozpracovanou službu** jádra a vrátit se v jádře jinam – velký problém se synchronizací, vyžaduje plně preemptibilní jádro (režie nízkoúrovňové synchronizace a zamykání).

Problém režie služeb jádra

Architektura výpočetních systémů je dnes víceprocesorová, nezáleží na tom, jestli jsou procesory na jednom čipu – označované jako jádra (*core*) nebo na nezávislých čipech. Komunikace mezi nimi je vždy stejná – pouze přes společnou paměť. Z úvodu do OS víme, že při práci se společnými daty z více procesů/procesorů musíme data zamykat:

```
P1(CPU A):          P2 (CPU B):
            int cnt;        /* sdílená proměnná */
lock(sem);          lock(sem);
cnt++;              cnt++;
unlock(sem);        unlock(sem);
```

Jak je implementována operace *lock()/unlock()*? Její režie musí být vzhledem k chráněné sekci malá. Může se v nich volat jádro? Lze použít k této synchronizaci komunikaci mezi procesory (např. IPI – *InterProcessor Interrupt*)?
Ne – proces P1 neví, zda proces P2 poběží na procesoru B, či nějakém jiném, či zda vůbec poběží, nemůže tedy zasílat při každé operaci *lock()* signál všem ostatním procesorům (zpracování přerušení trvá řádově stovky instrukcí), navíc je zaslání IPI privilegovaná operace, takže ji nelze generovat z uživatelského procesu.

Příklad: režie synchronizace (100 mil. krát ve 2 procesech):
a) lokální čítač 0,3 sekundy (každý proces má svůj čítač)
b) sdílený čítač 0,9 sekundy (ale nepočítá správně)
c) atomický čítač 6,9 sekund (nejrychlejší implementace)
d) POSIX mutex 26 sekund (semafor ve vláknech)
e) Unix semafor 398 sekund (semafor mezi procesy)
Poučení: volání jádra něco stojí.

Problém sdílené paměti (nesdílené proměnné)

```
P1(CPU A):          P2 (CPU B):
static int cnt1, cnt2;
cnt1++;             cnt2++;  /* 100 mil. krát */
```

Jak dlouho poběží?
- měl by běžet jako a) (nezávislé proměnné pro každý proces)
- v praxi poběží rychlostí b) až d)

„Předělal jsem program na vícevláknový, měl by tedy běžet N-krát rychleji, ale on běží desetkrát pomaleji!?!“

Proč?
Konzistence stavu paměti tak, aby všechny procesory viděly stejný obsah paměti, něco stojí. Z hlediska rychlosti není možné hlídat jednotlivé buňky paměti, ale větší úseky (řádek cache L1, obvykle 64 byte). Pokud se sejdou nezávislé proměnné v tomto úseku (v daném příkladu to hrozí), pak se procesory hádají, který má aktuální stav (každý zápis znamená rezervaci řádku cache a tím jeho invalidaci v cache druhého procesoru a naopak).

Poučení:

Využití více procesorů není jednoduché. V tomto případě lokální (nesdílená) data každého procesu musí být uložena odděleně v paměti (buď na zásobníku nebo *malloc()*, ale pozor na alokátory, které nezohledňují lokalitu procesů).

Typy operačních systémů

A) Monoprogramové - MS-DOS:

- aktivní pouze jeden proces,
- jednodušší implementace - nenastává souběžnost provádění,
- využití zdrojů slabé, obvykle jen jeden uživatel.

B) Multiprogramové (multitasking, multiprogramming):

- aktivních více procesů současně,
- efektivnější využití prostředků,
- jednodušší implementace vyšších vrstev (GUI, síť. rozhraní),
- nutnost pro víceuživatelský systém.

B-1) Jednoprocesorové (uniprocessor, UP)
B-2) Víceprocesorové, paralelní (multiprocessor, MP)
B-2-a) Symetrické multiprocesorové systémy (SMP)

- kód jádra i kód procesů je prováděn na všech procesorech,
- procesory mají rovnocenný přístup k operační paměti, V/V zařízením a přerušovacímu systému.

B-2-b) Nesymetrické multiprocesorové systémy

Jak realizovat současný běh více procesů než je počet procesorů?
Režim sdílení času (*timesharing*):

- v daný okamžik je prováděno pouze tolik procesů, kolik je procesorů,
- každý proces má procesor přidělen pouze omezenou dobu, poté je vystřídán jiným aktivním procesem,
- během delšího intervalu se rozprostře výpočetní výkon procesorů mezi všechny aktivní procesy a tím se vytváří iluze současného provádění všech běžících procesů,

Podmínka: operační systém musí umět pozastavit rozpracovaný proces a poté v něm pokračovat tak, aby to nebylo z procesu pozorovatelné.

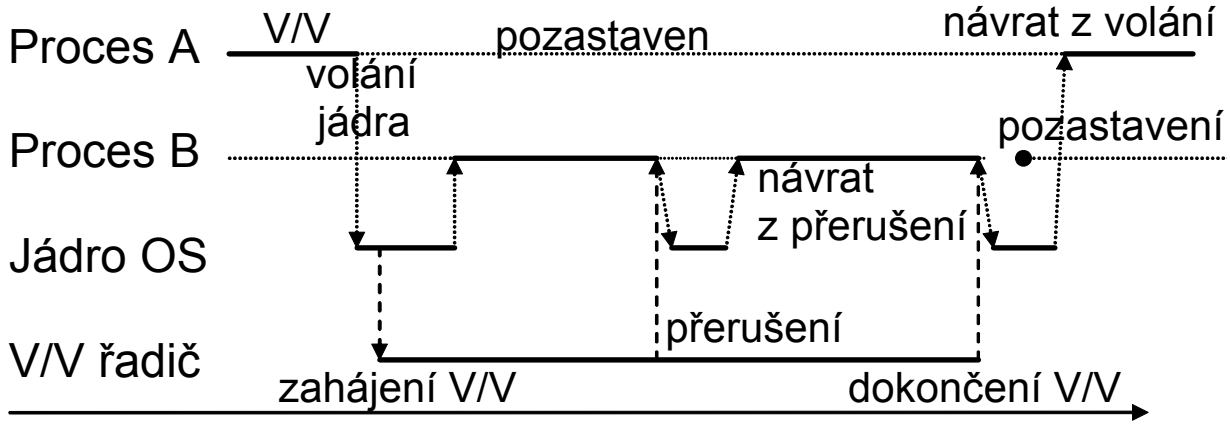
Ochrana operačního systému:

1. znemožnění modifikace kódu a datových struktur jádra OS,
2. zabránění provádění V/V operací,
3. zabránění přístupu do paměti mimo přidělený prostor,
4. odolnost vůči chybám (odebrání procesoru při zacyklení).

Nutná podpora na úrovni hardware:

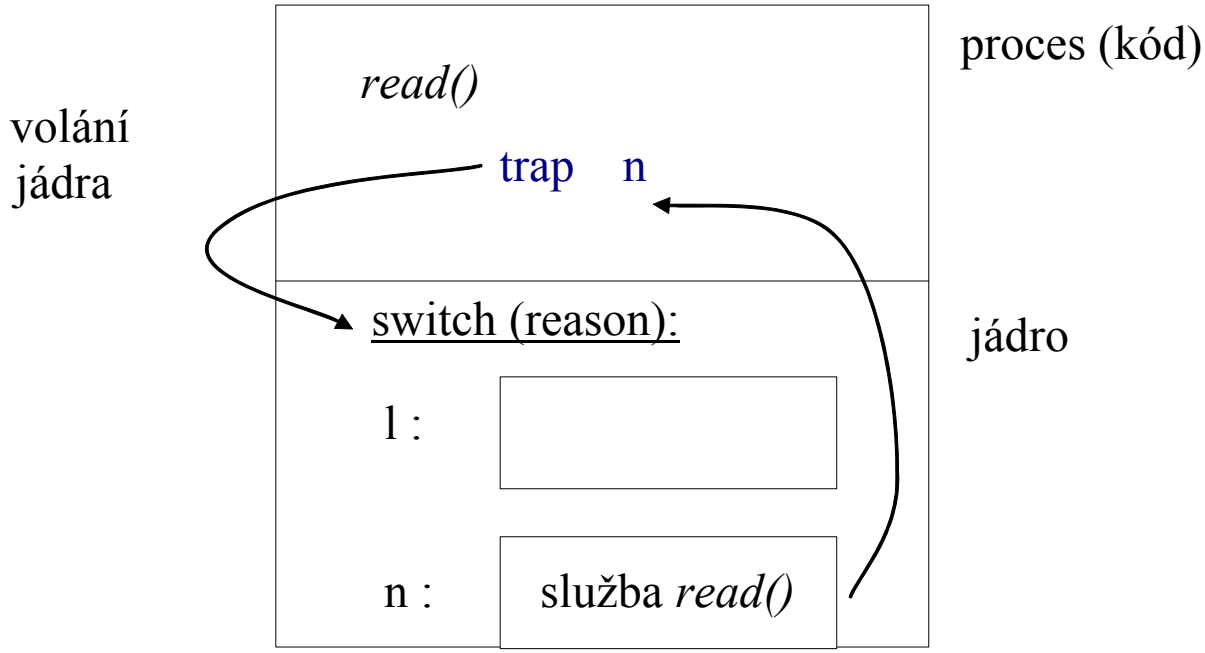
1. Dva režimy činnosti procesoru,
2. privilegované instrukce povolené pouze v systémovém režimu (V/V, změna režimu, zpracování přerušení),
3. ochrana paměti – definuje přístupné úseky adresového prostoru pro běžící proces,
4. přerušovací systém, přerušení převede procesor do systémového režimu,
5. generátor pravidelných přerušení (časovač),
6. pro efektivní V/V nutné DMA nebo inteligentní řadič (přenosy velkých objemů dat)

Uživatelský režim – provádění uživatelských procesů
Systémový režim – provádění kódu jádra OS



Přechody mezi režimy:
a) Přerušení – vždy do systémového režimu, na definovanou adresu
b) Návrat z obsluhy přerušení – návrat zpět (IRET)
c) Volání jádra – privilegovaná instrukce, přechod do systémového režimu, na pevnou adresu, parametry obvykle na zásobníku v uživatelském adresovém prostoru
d) Návrat z volání jádra – privilegovaná instrukce pro přechod do uživatelského režimu a nastavení programového čítače (PC)

Volání jádra



- volání jádra je realizováno speciální instrukcí (SVC, lcall, int, trap),
- jediný styk procesu s okolním prostředím,
- vše je zprostředkováno jádrem operačního systému,
- proces je zapouzdřen, operační systém pro něj vytváří iluzi virtuálního počítače.

Problém: spekulativní provádění (Spectre, Meltdown), zapouzdření sice nelze přímo prolomit, ale měřením rychlosti provádění instrukcí lze odhalit obsah nepřístupných dat.

Definice rozhraní jádra

A) Na úrovni binární:

```
#   na zásobníku fd, buf, length
_read:  lea $0x3,%eax
        lcall $7,$0         (Linux - int $0x80)
        jb error
        ret
error:  movl %eax,_errno
        movl $-1,%eax
        ret
```

Binární rozhraní je systémově závislé (procesor, verze systému)
Standardy: iBCS-2 (Intel),), MIPS ABI (SGI), ARM ABI, atd.

System V AMD64 ABI (Linux, BSD, Solaris, Mac OS X):

```
# parametry v registrech RDI,RSI,RDX,RCX,R8,R9
_read:  mov $0x3,%eax       # číslo služby read()
        mov %rcx,%r10
        syscall             # volání jádra
        jb error
        retq
error:  pushq %rax          # výsledek uložit
        call __error
        popq %rcx           # výsledek do rcx
        movl %ecx,(%rax)    # uložení errno
        movl $-1,%rax       # návrat s chybou
        movl $-1,%rdx       # (pro 128bit)
        retq
```

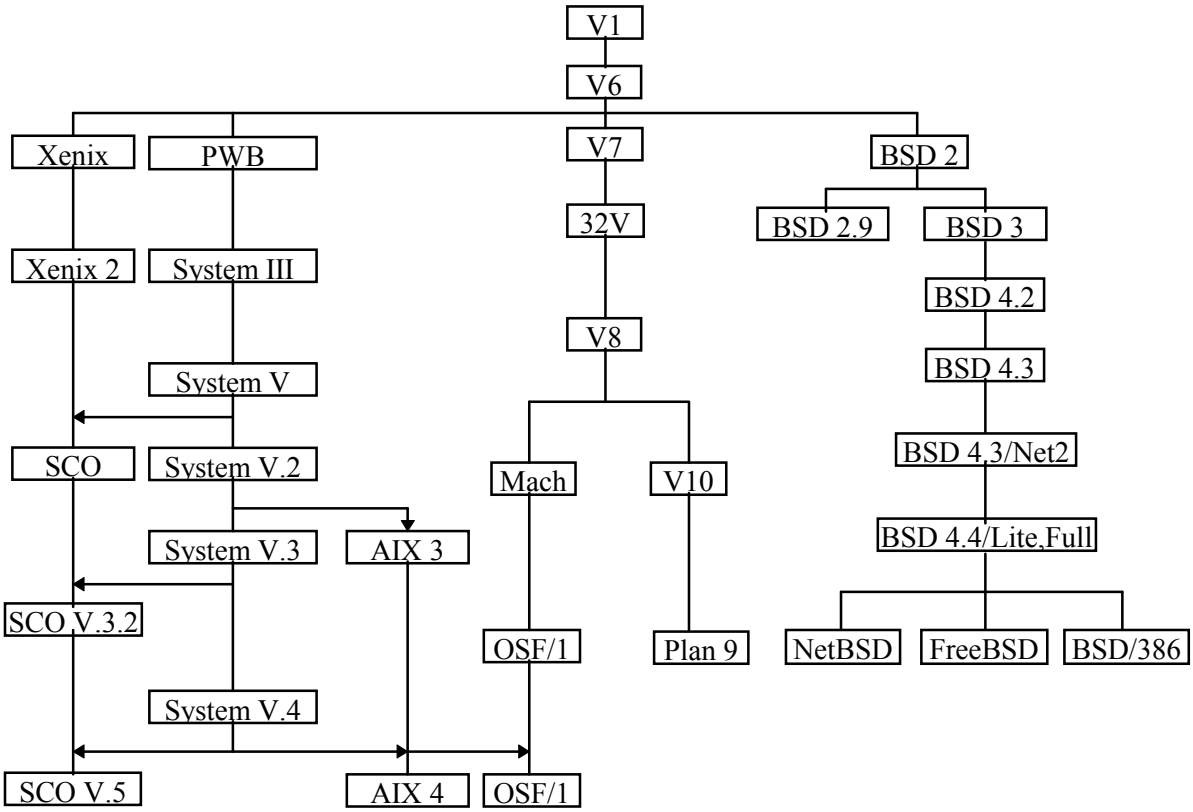
(AT&T asembler, %eax je 32bitový,%rax 64bitový registr AX)

B) Na úrovni zdrojové:

```
#include <unistd.h>
ssize_t  read(int fd, void *buf, size_t len);
```

Historicky také systémově závislé (např. datový typ *len*)

Standardizace rozhraní jádra - příklad Unixu



Různé verze - různá rozhraní, základ stejný
Problematická přenositelnost programů i na zdrojové úrovni – datové typy, signály, terminály, synchronizace, síťové rozhraní.

Sjednocování rozhraní Unixu (1980-2000):
SVID - UNIX System V Interface Definition

- definice rozhraní originálního komerčního Unixu z AT&T USL, publikována pro verze SVR3 a SVR4, základ pro komerční verze Unixu (HPUX, AIX, SOLARIS, IRIX)

X/Open (Open Group) – X/Open Portability Guide (XPG)

- sdružení výrobců, později základ Open Group
- širší definice rozhraní jádra, včetně knihoven, síťových rozhraní
- bez omezení na konkrétní verzi Unixu

IEEE/ISO POSIX 1003.1 – rozhraní jádra systému pro jazyk C

- součást standardů POSIX
- mezinárodní standard, vychází z Unixu, ale specifické rysy vypouští (mknod) a doplňuje scházející (signály)
- první verze 1990 jen základní služby, dnes všechny verze Unixu bez problému splňují,
- verze 1996 doplňuje vlákna (1003.1c-1995) a real-time rozšíření,
- verze 2001 doplňuje další služby, síťové rozhraní, uživatelské rozhraní
- verze 2003 je základem SUS UNIX03
- verze 2008 odpovídá SUS Version 4 edition 2008
- edice 2013 odpovádá SUS Version 4 edition 2013
- edice 2016 odpovádá SUS Version 4 edition 2016

Open Group (http://www.opengroup.org/)

- nezávislá organizace, zajišťuje certifikaci (nástroje)
- publikuje standardy Single Unix Specification - Version 1 (UNIX95), Version 2 (UNIX98), Version 3 (UNIX03), základ POSIX + XPG + nové
- UNIX V7 - SUS Version 4 edition 2008, 2013, 2016 – totožné s odpovídající verzí POSIX 1003.1

Přehled standardů IEEE POSIX 1003

Označení	schválen	název
1003.1	1988 1990 1996 2001 2004 2008	Programové rozhraní systému pro jazyk C, druhá verze sloučení s 1003.1b, 1003.1c a 1003.1i sloučení s SUSv2, 1003.2a, 1003.2b, 1003.2d, 1003.1a, 1003.1c, 1003.1d, část 1003.1g doplnění Single Unix rozšíření
1003.1a		Rozšířené rozhraní (symbolické odkazy, apod.)
1003.1b (1003.4)	1993	Rozšíření pro práci reálném čase
1003.1c (1003.4a)	1995	Rozšíření pro paralelní programování (*threads*)
1003.1d (1003.4b)	1999	Další rozšíření pro práci v reálném čase
1003.1e (1003.6)	zrušeno	Rozšíření pro ochranu a zabezpečení systému
1003.1f (1003.8)	zrušeno	Transparentní přístup ke vzdáleným souborům
1003.1g (1003.12)	2000	Protokolově nezávislé síťové rozhraní (*sockets*)
1003.2a (1003.2)	1992	Uživatelské rozhraní systému
1003.2d (1003.15)	1994	Rozšíření pro dávkové zpracování
1003.3	zrušeno	Metody testování a verifikace
1003.5	1992	Programové rozhraní systému pro jazyk Ada
1003.5b (1003.20)	1998	Rozšíření pro reálný čas v jazyce Ada
1003.9	1992	Programové rozhraní systému pro jazyk FORTRAN-77
1003.10	1995	Profil aplikačního prostředí pro superpočítače
1003.11	zrušeno	Transakční zpracování
1003.13	1998	Profily prostředí systémů reálného času
1003.14	zrušeno	Profily prostředí multiprocesorových systémů
1003.16	zrušeno	Jazykově nezávislé rozhraní systému
1003.17 (1224.2)	1993	Adresářové služby
1003.18	zrušeno	Základní profil prostředí systému
1003.19	zrušeno	Programové rozhraní systému pro jazyk FORTRAN-90
1003.21		Distribuované zpracování v reálném čase

POSIX 1003.1-2008 je ekvivalentní XPG7, obsahuje Base Definitions, System Interfaces, Shell & Utilities.

Jak správně použít rozhraní POSIX?

1. Před prvním hlavičkovým souborem vyžádat patřičnou verzi POSIX/XOPEN:

```
#define _POSIX_C_SOURCE 200809L
```

nebo (pokud je třeba XSI rozšíření):

```
#define _XOPEN_SOURCE  700
```
(definuje předchozí)

starší verze 200112L (zahrnuta v XPG6 = 600)

2. pro vlákna dle staršího POSIX 1003.1c-1992

```
#define _REENTRANT
```

3. pro nejstarší POSIX 1003.1-1990

```
#define _POSIX_SOURCE
```

4. používat hlavičkové soubory a prototypy funkcí (gcc –Wall)

```
#include <stdio.h>
```

5. jak zjistit implementovanou verzi POSIXu?

při překladu:

```
#include <unistd.h>
#if _POSIX_VERSION > = 200112L
...
#endif
```

při běhu programu:

```
printf("POSIX version %ld\n",
                (long)sysconf(_SC_VERSION));
```

Jaké verze můžeme mít:
198808L POSIX.1-1988
199009L POSIX.1-1990
199309L POSIX.1b-1993
199506L POSIX.1c-1996
200112L POSIX.1-2001
200809L POSIX.1-2008

6. nedeklarovat `extern int errno` (viz vlákna)

7. explicitně přetypovat všude kde hrozí použití implicitní konverze (parametry printf):

```
/* je size_t long nebo int? */
    size_t offset;
    printf("pozice=%ld", (long)offset);
/* nebo ISO C99 modifikátor z pro size_t */
    printf("pozice=%zd", offset);
/* je pid short, int nebo long? */
    pid_t pid;
    printf("pid=%ld", (long)pid);
```

8. služby jádra:

```
    výsledek = funkce(arg1, arg2, ...);
```

vždy testovat výsledek, chyba je obvykle -1 nebo NULL a současně je naplněno *errno!* Netestovat chybu nulováním *errno* a testováním errno bez indikace chyby výsledkem.

9. zjišťovat limity dynamicky, příklad délka jména souboru:

```
long name_max;
char *name;                 /* jméno souboru */
#ifdef NAME_MAX             /* statický limit */
    name_max = NAME_MAX;
#else
    name_max = pathconf(".", _PC_NAME_MAX);
    if (name_max == -1) {
        perror("pathconf");
        name_max = _POSIX_NAME_MAX; /* minimum */
    }
#endif
    printf("NAME_MAX=%ld\n", name_max);
    name = malloc(name_max+1);
```

10. některá rozhraní jsou volitelná, nemusí být implementována, např. thread cpu time:

```
#include <time.h>
#if _POSIX_THREAD_CPUTIME
    if (timer_create(CLOCK_THREAD_CPUTIME_ID,
&sigev, &tid)) == -1) ...
#else
#error "TCT extension missing"    /* co s tím */
#endif
```

11. vyhnout se rezervovaným idenfikátorům jazyka C a Posix

12. správně používat jazykovou lokalizi (*setlocale()*):

```
#include <locale.h>
char *lang;

/* nastavení lokalizace čísel na implicitní */
lang = setlocale(LC_NUMERIC, "");
/* zjištění nastavené lokalizece */
lang = setlocale(LC_NUMERIC, NULL);
/* vypnutí lokalizace */
setlocale(LC_NUMERIC, "C");
```

Kategorie:

`LC_COLLATE`	strcoll(), strxfrm()
`LC_CTYPE`	funkce is...(), toupper(), tolower(), mblen(), mbtowc(), ...
`LC_MONETARY`	localeconv()
`LC_MESSAGES`	gettext()
`LC_NUMERIC`	printf(), scanf()
`LC_TIME`	strftime()
`LC_ALL`	vše

Jádro OS

A) Správa procesoru

Proces – prováděný program (v uživatelském režimu)
Virtuální procesor – jádro OS zajišťuje provádění kódu programu se zabráněním vlivu činnosti jiných procesů, V/V a přerušení – program je prováděn tak, jakoby měl procesor pouze pro sebe. Jádro OS rozšiřuje instrukční repertoár o služby jádra.

Funkce správy procesoru:

- vytváření, rušení procesů
- pozastavení, pokračování provádění procesu
- synchronizace procesů (čekání na ukončení spuštěného)
- komunikace mezi procesy (předávání parametrů, dat)

Def.: Přepnutí kontextu – pozastavení prováděného procesu a pokračování v provádění jiného pozastaveného procesu = předání procesoru mezi procesy:
Preemptivní – bez spoluúčasti procesů,
Nepreemptivní – kooperativní předání procesoru.

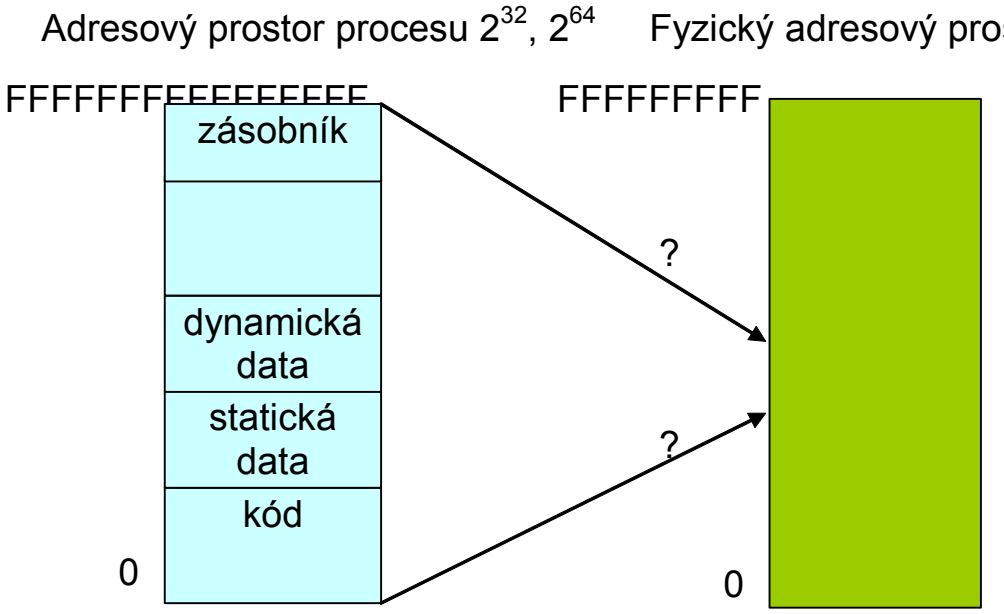
Kdy nastává přepnutí kontextu:

- vyčerpání časového kvanta (plánování v režimu sdílení času)
- zahájena blokující operace (čtení, zápis)
- proces je pozastaven (*sleep*, *wait*)
- má běžet proces s vyšší prioritou

Def.: Adresový prostor - všechny adresy dostupné programu (logický AP) - iluze virtuálního počítače, ochrana
Def.: Stav procesu – všechny informace, které musí být uloženy při pozastavení procesu = registry, PC, SP, adresový prostor.

B) Hlavní paměť

- Adresový prostor procesů je u většiny moderních operačních systémů oddělený, tj. začíná od logické adresy 0 a končí maximální adresovatelnou buňkou.



- Transformaci adresy z logického adresového prostoru procesu na fyzickou adresu operační paměti zajišťuje MMU (jednotka transformace adres).
- Operační systém musí zajistit správné naplnění MMU před přidělením procesoru danému procesu (při přepnutí kontextu) a dále při každé změně obsazení (využití) adresového prostoru (problém – vícevláknový proces běžící na více procesorech).

Funkce správy paměti:

- virtualizace adresového prostoru
- nastavení a aktualizace transformace adres
- udržování procesů v paměti (co nejvíce)
- přidělování a evidence přidělených úseků paměti
- zavádění a odkládání procesů
- sdílení paměti mezi procesy (sdílení kódu, dat)
- ochrana paměti

C) V/V

Standardní rozhraní mezi procesy a V/V zařízeními. Realizace abstraktních V/V operací (čti řádek, zapiš blok):

- zahájení V/V
- obsluha přerušení
- ukončení V/V
- systémový časovač

Správa V/V - společná vrstva rozhraní V/V a ovladačů, detekce a autokonfigurace, systémové buffery, podpora pro vrstvené hierarchické ovladače.
Síťové rozhraní (sockets, streams), systém souborů, atd.

Typy operačních systémů

Plánování:

- dávkové (batch),
- sdílení času (timesharing),
- systémy reálného času (real-time)

Použití:

- univerzální
- specializované (souborový server, databázový server, RT)

Počet uživatelů:

- jednouživatelské,
- víceuživatelské

Struktura OS

Vztah HW-jádro-proces:

- monolitické jádro – monitor, nestrukturováno (MS-DOS)
- klasické jádro (Unix)
- mikrojádro: zasílání zpráv, model klient-server
- exokernel: služby jádra součástí procesu
- virtuální počítač

Struktura OS:

- vrstvené struktury: hierarchie procesů, funkční hierarchie, virtuální počítač, modulární jádro
- klasické jádro (Unix)

1. Monolitické jádro (monitor)

- bez vnitřní struktury (big mess)
- volání mezi moduly voláním podprogramů
- žádné omezení volání a vztahů mezi moduly
- uživatelský proces = podprogram jádra
- často bez rozdělení na systémový/uživatelský režim

2. Jádro (kernel)

- jádro běží v systémovém režimu
- procesy běží v uživatelském režimu a volají jádro (jádro je pasivní), striktní rozhraní mezi procesy a jádrem
- jádro vytváří pro proces abstrakci virtuálního počítače

Služby jádra jsou podprogramy procesu.

Příklad: klasický Unix (jednoprocesorový)

- kód jádra běží v kontextu procesu – systémová fáze procesu, tento běh v jádře je bez přepínání kontextu, jádro je zcela pasivní.

- Problém: pasivní koncepce jádra – obsluha událostí vzniklých mimo kontext procesu je problematická (pseudoplánování aktivit vykonávaných před návratem do uživatelského režimu).

3. Mikrojádro (Mach)

- Služby jádra částečně v systémovém režimu (mikrojádro), částečně v uživatelském režimu (systémové procesy)

Minimální jádro:

- přepínání kontextu
- přidělování paměti
- ochrana paměti, nastavení adresového prostoru

Ostatní služby řešeny samostatnými procesy nad mikrojádrem:

- prostředí procesů, spouštění procesů
- autentizace, autorizace, účtování
- virtualizace paměti, odkládání, zavádění
- V/V
- síťové vrstvy
- systém souborů

Systémové služby jsou realizovány procesy.

Rozhraní:

- klasické jádro = pasivní interní i vnější rozhraní jádra ve tvaru volání podprogramu, interní odlišné od vnějšího,
- mikrojádro = pasivní rozhraní mikrojádra, vně a mezi moduly nad mikrojádrem jednotné rozhraní zasíláním zpráv.
- **Problém** – efektivita komunikace mezi procesy, přepínání kontextu

Klasická koncepce mikrojádra se příliš nerozšířila, ale některé principy se používají v dnešních Unixových systémech (procesy uvnitř jádra, zpracování přerušení procesy, vyčlenění části správy paměti do procesů uvnitř jádra, apod.)

4. Exokernel

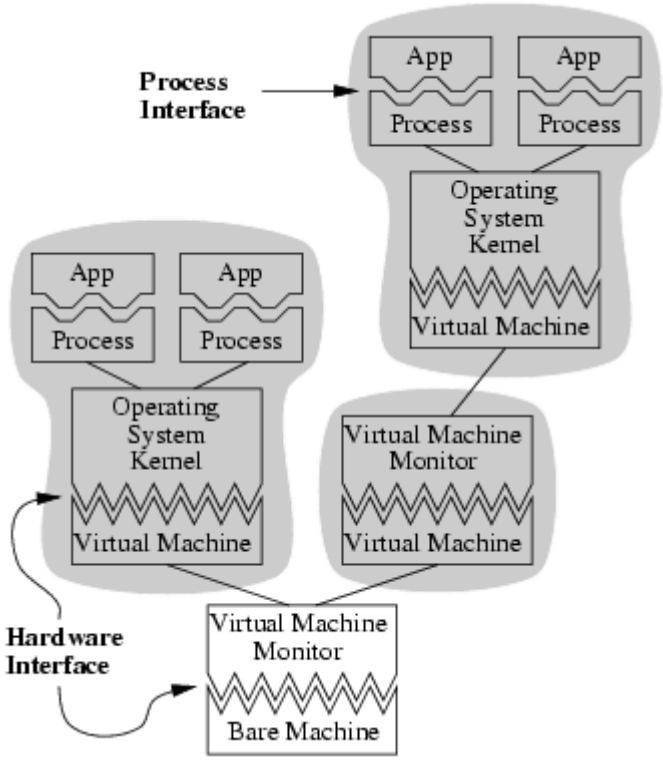
Motivace: jádro vykonává většinu služeb ve prospěch nějakého procesu, nemohl by to dělat ten proces sám?

Příklad: Jádro implementuje složitě systém souborů a snaží se odhadnout způsob práce programu se souborem (cache bloků, metadat, atd.). Mnohým programům ale stačí kus prostoru na disku s přímým přístupem, bez složité mezivrstvy. Struktura může být implementována formou knihovny. Problém: jak zajistit bezpečnost, přístupová práva na HW?

Služby jsou poskytovány jako podprogramy uvnitř uživatelského procesu. Jádro v systémovém režimu je voláno pouze pro synchronizaci a přidělování prostředků.

4. Virtuální počítač (Virtual Machine)

- jádro běží zcela v uživatelském režimu
- v systémovém režimu běží pouze monitor virtuálního počítače - zachytává a emuluje privilegované instrukce
- plně virtualizuje všechny prostředky
- nad monitorem mohou běžet plné operační systémy (IBM VM/370, emulace VM86 = MS-DOS okno, VMWare, Virtual PC, Xen)



Struktura vrstev (Dijkstra)

Princip - vrstvy s definovanou funkcí, volání pouze podřízených vrstev.

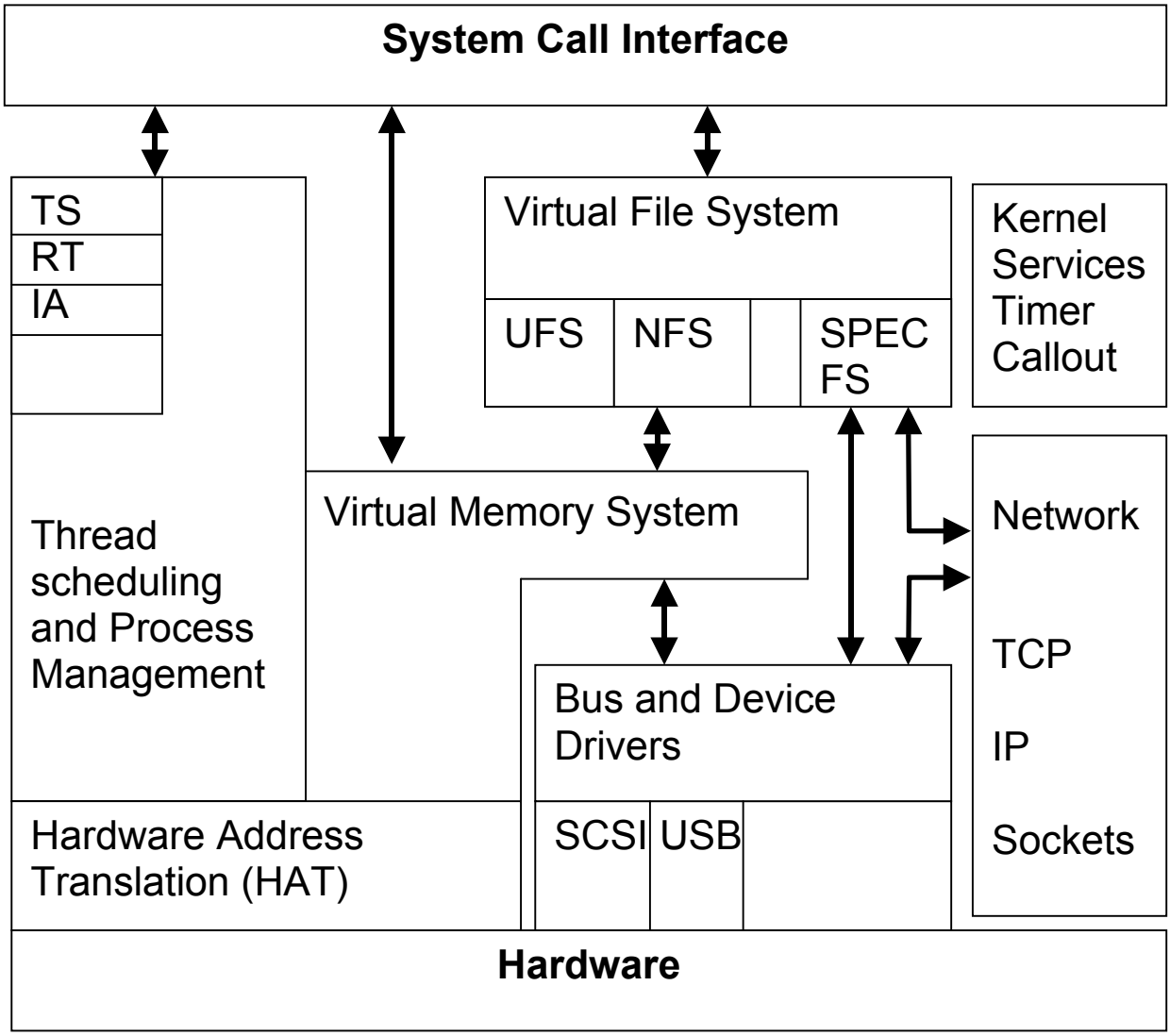
1. virtuální procesor - přepínání kontextu, synchronizace
2. přidělování paměti, uvolňování
3. plánování - přidělování procesoru, zastavení, synchronizace na vyšší úrovni
4. přidělování paměti na vyšší úrovni, DTA
5. V/V - zahájení, zpracování přerušení
6. Síťové vrstvy
7. spouštění a ukončování procesů
8. systém souborů
9. virtualizace paměti
10. rozhraní jádra, prostředí procesu

Problém - vztahy jsou komplikovanější, nelze pouze zdola nahoru.

Klasické jádro

- **rozdělení podle funkce** - rozhraní jádra, systém souborů, virtualizace paměti, plánování, přepínaní kontextu, přidělování paměti, zpracování přerušení, blokové operace a systémové buffery, synchronizace, společné podprogramy, ovladače, atd.
- **volání mezi moduly** - není striktně omezeno, pouze rozhraní ovladačů a V/V, synchronizace

Příklad: Unix V.4 (Solaris, UnixWare)



Směry vývoje:

Plně preemptivní SMP jádra – Solaris, Linux, FreeBSD – problém režie synchronizace

Mikrojádro: Mach, OSF/1, GNU Hurd, Chorus, Amoeba

Objektově orientované jádro: Sun Spring
Mikrojádro implementuje domény, vlákna a dveře (volání metod mezi doménami), ostatní služby jsou implementovány komunikujícími objekty nad jádrem.