

B4B350SY: Operační systémy

Lekce 9: Vstup/výstup, ovladače

Michal Sojka

michal.sojka@cvut.cz



November 20, 2022

1 Úvod

2 Úložiště

- Jak funguje hardware úložiště?
- Přístup k datům, stránková cache
- Disková pole

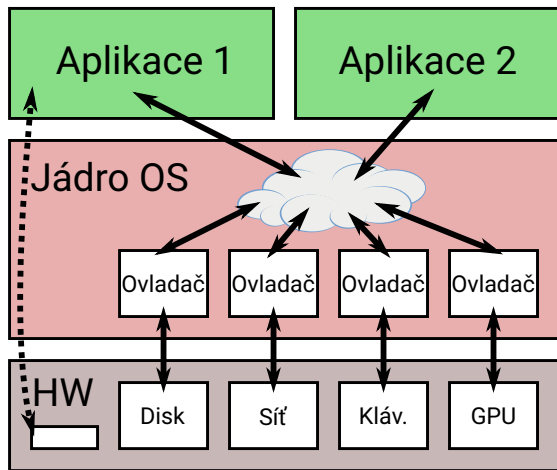
3 Síťová rozhraní

- Hardware
- Příjem a odesílání dat
- Rozvrhování rámců

4 Ovladače

- Linux
- Windows
- Ovladače v uživatelském prostoru

Vstup a výstup v OS



- Takto to vypadá v OS s monolitickým jádrem (Linux, Windows)
 - Pro přístup k HW musí aplikace využívat služby jádra
 - Privilegované aplikace (např. root v Linuxu) mohou přistupovat k některému HW přímo (viz sekci Ovladače v uživatelském prostoru)
- V OS s μ -jádry jsou ovladače většinou samostatné procesy mimo jádro, tedy také v uživatelském prostoru

Vstup a výstup

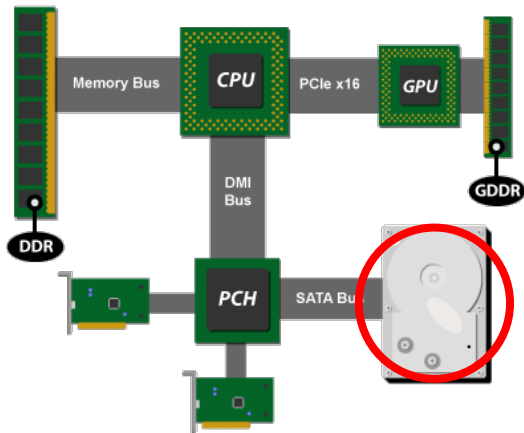
Input/Output (I/O)

- Způsob, jak počítač komunikuje s okolním světem
 - Datová úložiště (disky)
 - Síť
 - Klávesnice, monitor, ...
- Uživatelská aplikace nemá přímý přístup k perifériím (HW)
 - Aplikace, která nepoužívá služby jádra OS může pouze číst a zapisovat do (virtuální) paměti
- Pro přístup k perifériím musí používat služby OS, které
 - zajišťují „bezpečné“ **sdílení** periférií mezi aplikacemi a
 - **abstrahují** hardwarové detaily a poskytují jednotné API pro všechny periferie stejné třídy.
 - K tomu využívají služeb **ovladačů zařízení**, které naopak řeší všechny detaily práce s konkrétním hardwarem.

Úložiště

- HW pro ukládání velkého množství dat
- Není možné číst data po jednotlivých bytech, ale po tzv. blocích či sektorech
- Pevný disk – velikost bloku 512 B, 4 kB, ...
 - Rotační
 - Solid-state (SSD)
- Flash paměť – někdy lze číst po bytech, ale mazat jde jen po blocích – typicky 128 kB
 - Typicky v embedded zařízeních
 - Základem pro SSD disky

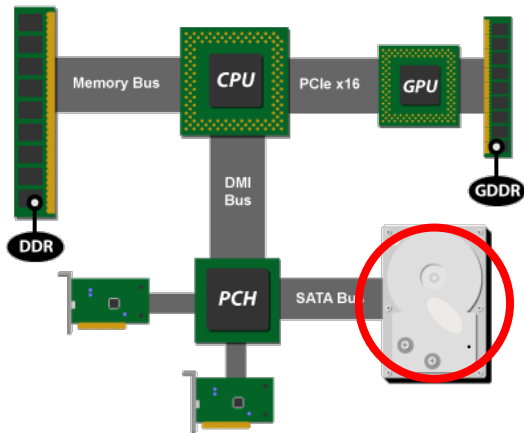
Model HW



- Pevný disk (HDD či SSD) je malý počítač, který komunikuje s hlavním CPU pomocí sběrnice.
- Přístup k disku je řádově pomalejší než přístup k paměti
- CPU posílá příkazy, disk je autonomně vykonává
- Disk využívá tzv. *Direct Memory Access* (DMA), také označovaný jako *Bus Master*.
 - Data proudí do paměti bez zásahu software v CPU

Platforma Intel's P55. Zdroj: ArsTechnica

Model HW



Platforma Intel's P55. Zdroj: ArsTechnica

■ Typické příkazy:

- Ulož do sektorů 123456–123460 data z paměti na adrese 0x2f003200
- Načti 32 sektorů počínaje č. 7654 a ulož je do paměti na adresu 0x302f1200
- Disky umí zpracovávat víc příkazů najednou (typicky 32)
 - Interně provádí optimalizace (např. změna pořadí vykonávání či slučování požadavků).
 - O dokončení operace je CPU (přesněji ovladač běžící na CPU) informován přerušením.

Přístup aplikací k úložišti

- Aplikace typický nepřistupují k úložišti přímo, ale skrze **souborový systém** (viz příští přednášku)
- OS optimalizuje přístup k úložišti:
 - Spravuje vyrovnávací paměť pro rychlejší přístup k datům na disku
 - OS sám předem načítá data, o kterých předpokládá, že budou brzy potřeba
 - Pro pomalé rotační disky:
 - Slučuje požadavky aplikací do větších
 - Rozvrhuje, kdy který požadavek vykonat – optimalizace přejezdů hlaviček – tzv. IO scheduler.

Stránková vyrovnávací paměť

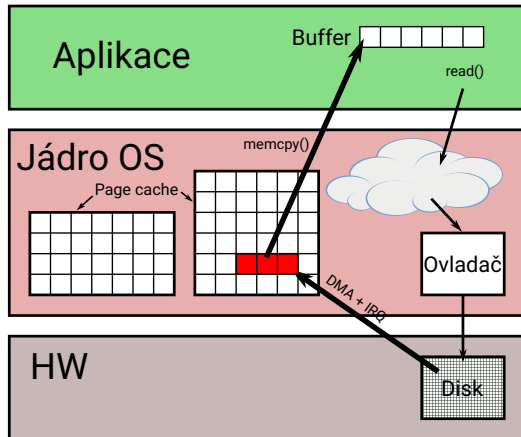
Page cache – název používaný Linuxem pro vyrovnávací paměť disku

- Data čtená z disku resp. zapisovaná na disk jsou uchovávána v paměti pro případné další použití
- OS se snaží využít veškerou volnou paměť jako diskovou cache
- Spravována po stránkách (4 kB)
 - I když starší disky používaly 512 B sektory, OS (téměř) vždy načítá celé 4 kB.

Čtení a zápis

■ Čtení z disku:

- 1 Aplikace zavolá `read()`
- 2 Jádro OS (JOS) přepośle požadavek správnému ovladači
- 3 Ovladač disku pošle příkaz pro načtení dat a uložení do page cache
- 4 Disk sám o sobě posílá data do paměti (DMA) a o dokončení informuje přerušením (interrupt request, IRQ)
- 5 V reakci na IRQ, JOS zkopíruje data z page cache do paměti aplikace



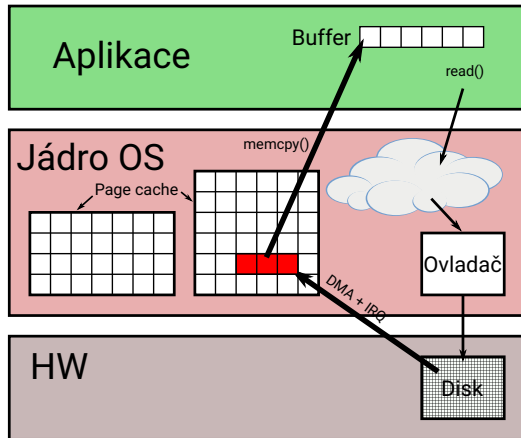
Čtení a zápis

■ Zápis na disk:

- 1 Aplikace zavolá `write()`
- 2 JOS data zkopíruje z aplikace do page cache a `write()` se vrátí.
- 3 Čas od času JOS zapisuje „špinavé stránky“ na disk. V Linuxu označováno jako „writeback“.
 - Zápis se dá vynutit systémovým voláním `fsync()` (Linux)

Pozn.: `fsync()` vs. `fflush()`

`fsync()` ukládá data z page cache na disk, `fflush()` posílá data z bufferu aplikace (schovaný v libc) do jádra (page cache v případě práce se soubory soubory).



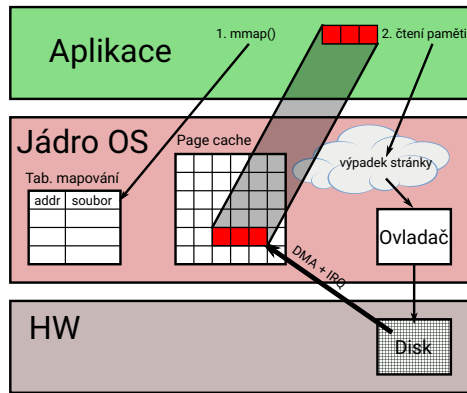
Anketa

Jaký je rozdíl mezi `fsync` a `fflush`?

- A** Různá jména pro stejnou funkcionalitu.
- B** `fsync` synchronizuje „stdio“ buffery v aplikaci s buffery v jádře
`fflush` posílá obsah bufferů v jádře (page cache) hardwaru
- C** `fsync` posílá obsah bufferů v jádře (page cache) hardwaru
`fflush` synchronizuje „stdio“ buffery v aplikaci s buffery v jádře

Čtení a zápis bez zbytečného kopírování dat

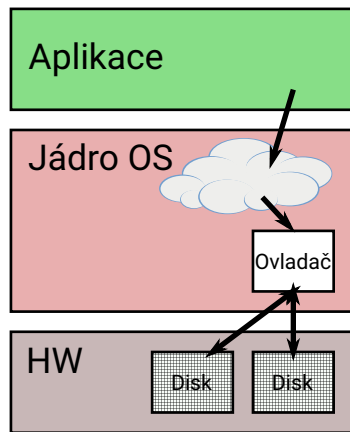
- Aplikace může požádat JOS, aby „namapovalo“ stránku diskové cache do jejího adresního prostoru (v UNIXu systémové volání `mmap()`)
- Při prvním přístupu k paměti vrácené funkcí `mmap` dojde k výjimce (výpadku stránky), protože žádná data nejsou v paměti načtena
 - 1 JOS se podívá do tabulky mapování (pro Vás viditelná v `/proc/<PID>/maps`), aby zjistil, jaký soubor je potřeba načíst a načte data z disku do stránkové cache
 - 2 Poté modifikuje stránkovací tabulku procesu a vrátí se z obsluhy výjimky na instrukci, která výjimku způsobila
 - 3 Tentokrát se instrukce provede úspěšně a aplikace pokračuje
- Zápis se provádí stejně jako čtení – prostým přístupem (zápisem) k namapované paměti
- Čas od času JOS zapisuje „špinavé stránky“ na disk.
- Může aplikace zjistit (nebo zajistit), že jsou data bezpečně uložena na disku?



- Pouze při použití `msync()` máme jistotu, že jsou data uložena na disku (pro případ výpadku napájení)
- Sdílení dat jednoho souboru mezi procesy se uskutečňuje prostřednictvím page-cache a není vázáno na uložení na disk

Disková pole

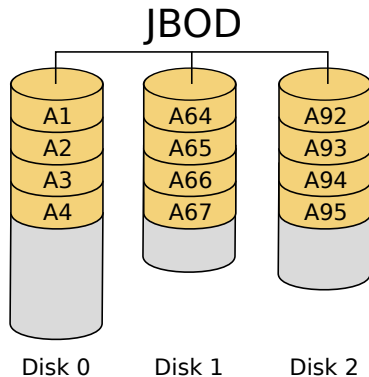
RAID – Redundant Array of Independent Disks



- Pokud se disk porouchá, přijdeme o (cenná) data
- Redundance – data jsou uložena na více místech najednou
- Možnost implementace v HW nebo v SW (OS)
- Rychlost SW implementace – čtení typicky rychlejší (paralelní čtení z více disků), zápis o něco pomalejší.
- Nahradí RAID zálohování dat?
 - Požár v serverovně – záloha na jiném místě
 - Administrátor omylem smaže data

Typy diskových polí

- RAID0 – spojení více disků do jednoho virtuálního (bez redundance)
- RAID1 – zrcadlení, efektivita: 50%
- RAID5 – prokládání dat a parita, min. 3 disky, toleruje ztrátu jednoho disku, efektivita $\frac{n-1}{n}$
 $A_p = A_1 \oplus A_2 \oplus A_3$ (xor)
Při poruše 1. disku:
 $A_1 = A_p \oplus A_2 \oplus A_3$
- RAID6 – toleruje ztrátu dvou disků



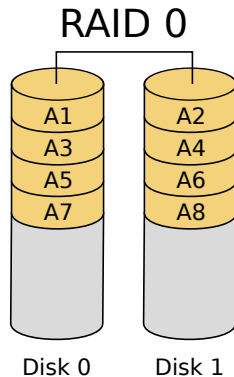
Autor: en>User:Cburnett – Vlastní dílo, CC BY-SA 3.0

Typy diskových polí

- RAID0 – spojení více disků do jednoho virtuálního (bez redundance)
- RAID1 – zrcadlení, efektivita: 50%
- RAID5 – prokládání dat a parita, min. 3 disky, toleruje ztrátu jednoho disku, efektivita $\frac{n-1}{n}$
$$Ap = A1 \oplus A2 \oplus A3 \text{ (xor)}$$

Při poruše 1. disku:

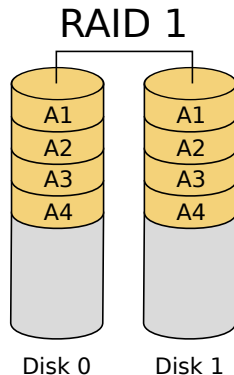
$$A1 = Ap \oplus A2 \oplus A3$$
- RAID6 – toleruje ztrátu dvou disků



Autor: en>User:Cburnett – Vlastní dílo, CC BY-SA 3.0

Typy diskových polí

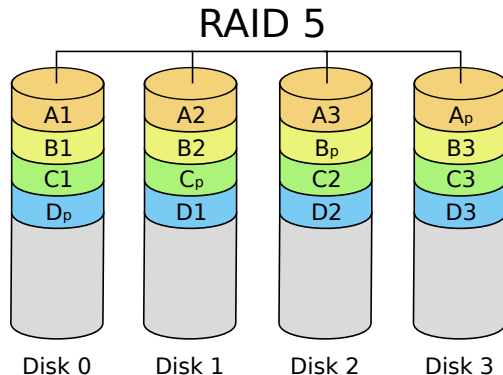
- RAID0 – spojení více disků do jednoho virtuálního (bez redundance)
- RAID1 – zrcadlení, efektivita: 50%
- RAID5 – prokládání dat a parita, min. 3 disky, toleruje ztrátu jednoho disku, efektivita $\frac{n-1}{n}$
 $A_p = A_1 \oplus A_2 \oplus A_3$ (xor)
Při poruše 1. disku:
 $A_1 = A_p \oplus A_2 \oplus A_3$
- RAID6 – toleruje ztrátu dvou disků



Autor: en>User:Cburnett – Vlastní dílo, CC BY-SA 3.0

Typy diskových polí

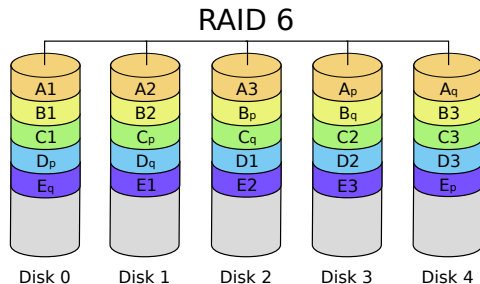
- RAID0 – spojení více disků do jednoho virtuálního (bez redundance)
- RAID1 – zrcadlení, efektivita: 50%
- RAID5 – prokládání dat a parita, min. 3 disky, toleruje ztrátu jednoho disku, efektivita $\frac{n-1}{n}$
 $A_p = A1 \oplus A2 \oplus A3$ (xor)
Při poruše 1. disku:
 $A1 = A_p \oplus A2 \oplus A3$
- RAID6 – toleruje ztrátu dvou disků



Autor: en>User:Cburnett – Vlastní dílo, CC BY-SA 3.0

Typy diskových polí

- RAID0 – spojení více disků do jednoho virtuálního (bez redundance)
- RAID1 – zrcadlení, efektivita: 50%
- RAID5 – prokládání dat a parita, min. 3 disky, toleruje ztrátu jednoho disku, efektivita $\frac{n-1}{n}$
 $A_p = A_1 \oplus A_2 \oplus A_3$ (xor)
 Při poruše 1. disku:
 $A_1 = A_p \oplus A_2 \oplus A_3$
- RAID6 – toleruje ztrátu dvou disků

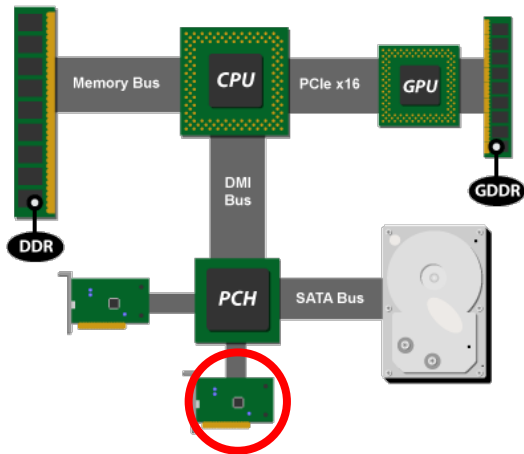


Autor: en:User:Cburnett – Vlastní dílo, CC BY-SA 3.0

Sítě

- Síťový subsystém OS řeší mnoho různých aspektů:
 - Ethernet, Wi-Fi, Bluetooth, CAN, ...
 - Routování
 - Virtuální sítě – VPN, ...
- Ethernet představuje základní model sítě používaný OS
 - Základní funkce všech technologií jsou stejné: **posílání a příjem rámců**
 - Jednotlivé síťové technologie se liší především nastavováním parametrů (WiFi: SSID, Ethernet: bitrate,
- OS reprezentuje síťový HW pomocí tzv. síťových rozhraní
- Sítě jsou velmi rychlé – dnes až 100 Gbps
- Síťový subsystém OS musí být velmi efektivní, aby OS nebyl úzkým hrdlem
- Úložiště a sítě mají z pohledu OS mnoho společného
 - Do nedávna nebyla efektivita diskového subsystému důležitá, ale s nástupem rychlých SSD disků nabývá na důležitosti a síťování je zde inspirací

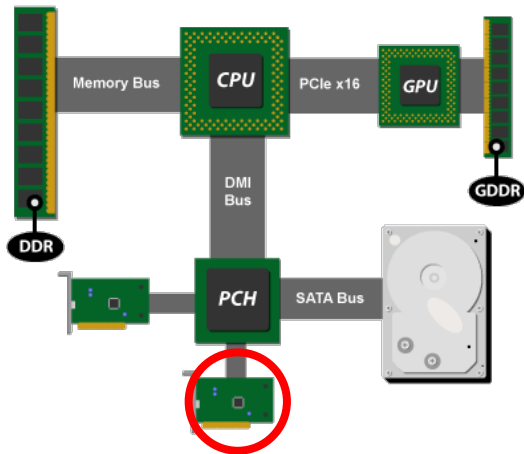
Sítový hardware



Platforma Intel's P55. Zdroj: ArsTechnica

- Sítové rozhraní je malý počítač, který komunikuje s hlavním CPU pomocí sběrnice.
- CPU posílá příkazy, sítové rozhraní je autonomně vykonává
- Používá se tzv. *Direct Memory Access* (DMA), také označovaný jako *Bus Master*.
 - Data proudí z/do paměti bez zásahu software v CPU

Síťový hardware

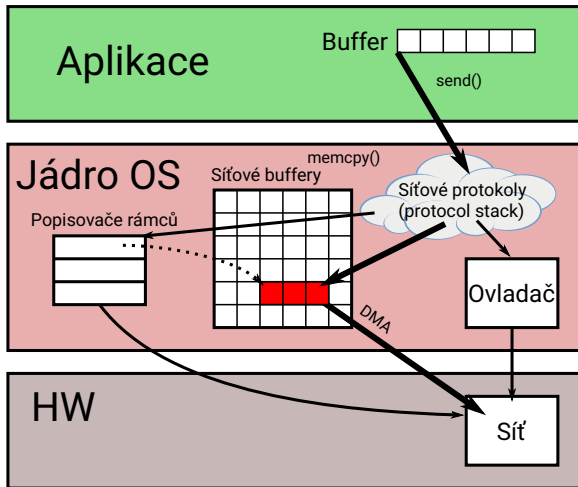


Platforma Intel's P55. Zdroj: ArsTechnica

■ Typické „příkazy“:

- Pošli rámec, který je uložený na adrese 0x2f003200.
- Pokud přijmeš rámec, ulož ho na adresu 0x302f1200.
- Implementováno pomocí tabulky popisovačů rámců (packet descriptor table) – ovladač vytvoří v paměti tabulku ukazatelů na rámce a síťové rozhraní se do ní „kouká“ při příjmu či odesílání rámců.

Odesílání dat aplikacemi



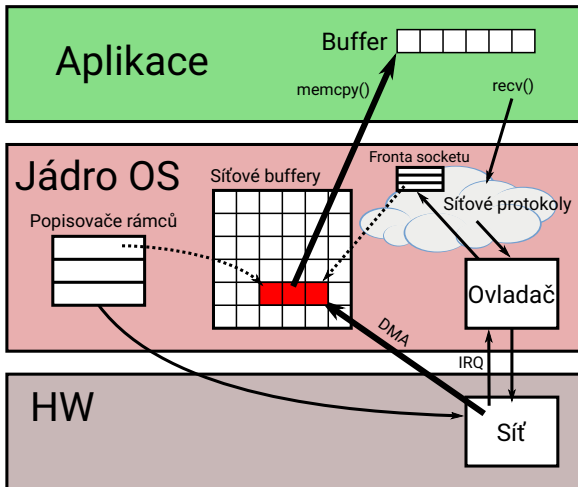
- 1 Aplikace zavolá send()/write()
- 2 JOS si zkopíruje odesílaná data do síťových bufferů
- 3 JOS (tzv. protocol stack) přidá k aplikačním datům potřebné hlavičky a upozorní ovladač
- 4 Ovladač upraví tabulku popisovačů rámců, a dá vědět (jak? zápisem do registru v síťovém HW) síťovému HW, že se tabulka popisovačů změnila.
- 5 Síťový HW začne číst data z paměti (DMA) a odešle je.

Anketa

Jak ovladač upozorní HW, že data v paměti může začít odesílat na síť?

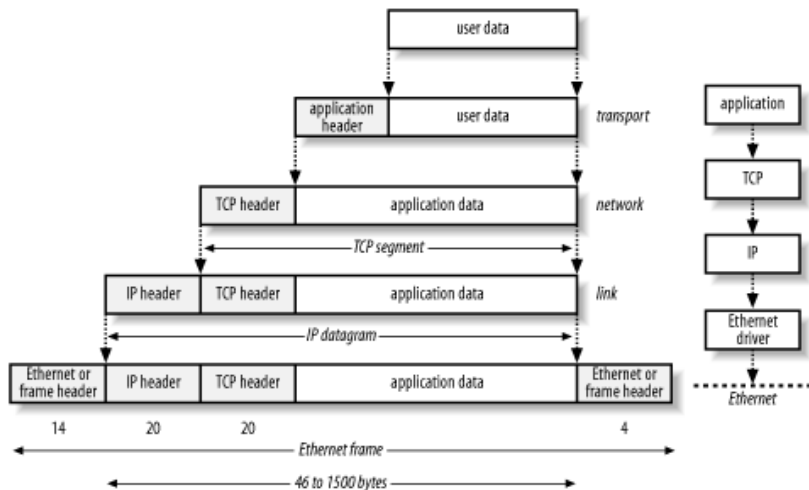
- A** Zasláním přerušení do HW.
- B** Zápisem do registru daného HW.
- C** Nijak. HW sám časem zjistí, že se v paměti objevila nová data k odeslání.

Příjem dat aplikacemi



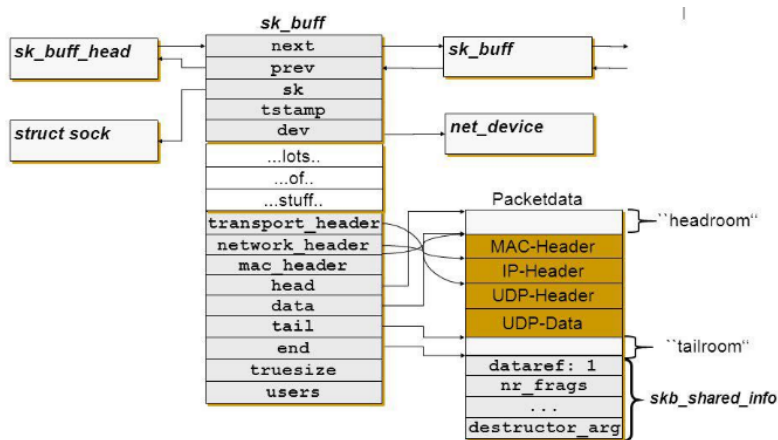
- 1 Aplikace zavolá `recv()/read()`
- 2 JOS zkontroluje, zda fronta socketu obsahuje nějaká přijatá data. Pokud ano, pokračuje se krokem 6, v opačném případě se volající vlákno zablokuje a čeká.
- 3 Sítové rozhraní autonomně ukládá přijímané rámce do paměti (DMA).
- 4 Po dokončení příjmu je upozorněn ovladač (přerušení) a ten pak aktivuje zpracování rámce síťovými protokoly.
- 5 Poté JOS zařadí rámec do fronty příslušného socketu.
- 6 JOS přijatá data nakopíruje ze síťových bufferů v jádře do aplikačního bufferu a systémové volání se vrátí do aplikace.

Síťové protokoly



Datová struktura pro práci se síťovými rámcí

struct skbuff v Linuxu



- Možnost přidávat hlavičky před data, bez nutnosti jejich kopírování
- Scatter-gather DMA – hardware si umí sestavit rámec „za běhu“ z více částí

Příjem a odesílání dat bez kopírování

Zero-copy networking

- Podobný „trik“, jako s diskovou vyrovnávací pamětí
- `socket(AF_PACKET, ...)` + `mmap()`
- Síťový HW přijímá/odesílá rámce rovnou do/z paměti kontrolované aplikací
- Nevýhody:
 - Aplikace si musí sama řešit přidávání a odebrání hlaviček
 - Aplikace nesmí modifikovat rámce (např. kvůli chybě v programu) , které jsou v procesu odesílání.

Rozvrhování rámců při odesílání

Traffic scheduling/control

- Další z činností, kterou řeší síťový stack OS
- Prioritizace interaktivní komunikace
- Spravedlivé rozdělení šířky pásma mezi uživatele (zákazníky)
- Problém zvaný „buffer-bloat“
 - Ovladač může do odesílací fronty (popisovač rámců k odeslání) uložit velké množství rámců.
 - Síťový HW odesílá rámce v pořadí, v jakém jsou tam uvedeny.
 - Pokud je na konci fronty rámec, který by měl být odeslán přednostně, musí dlouho čekat.
 - Řešení:
 - Fronta ovladače se udržuje krátká, aby kritické rámce mohly „předbíhat“
- Moderní síťový hardware implementuje více front pro odesílání (i příjem)
 - Rámce jsou rozvrhovány (i) v hardwaru – výběr fronty
 - Využívá se ve vícejádrových systémech, kde má každé jádro samostatnou frontu a není potřeba v ovladači ztrácet čas synchronizací (mutex) mezi různými CPU
 - Někdy lze využít i k prioritizaci rámců – každá fronta má jinou prioritu

Ovladač zařízení

Device driver

- Software, který
 - 1 **ovládá konkrétní zařízení** (disk, síťová karta, GPU, ...) a
 - 2 zbytku OS **nabízí jednotné rozhraní (API)**
- Se zařízením typicky komunikuje pomocí do paměti mapovaných registrů
 - V Linuxu viz příkaz `lspci -v`

```
01:00.0 Network controller: Intel Corporation Wireless 8260 (rev 3a)
Subsystem: Intel Corporation Wireless 8260
Flags: bus master, fast devsel, latency 0, IRQ 129
Memory at ef100000 (64-bit, non-prefetchable) [size=8K]
Capabilities: <access denied>
Kernel driver in use: iwlwifi
Kernel modules: iwlwifi
```
- Obsluhuje přerušení od zařízení (IRQ)

Spolehlivost ovladačů

- Ovladače bývají nejméně spolehlivou částí jádra OS
 - Chyba kdekoli v jádře OS (tedy i v ovladači) může způsobit nestabilitu celého systému
 - Ne každý programátor ovladačů rozumí všem potřebným detailům
 - Ovladače se nedají testovat, pokud není k dispozici konkrétní HW
 - Velmi špatně se testuje obsluha chybových stavů, protože je potřeba donutit HW, aby signalizoval chybu
 - Microsoft zavedl povinné digitální podepisování ovladačů, aby měl částečnou kontrolu nad jejich kvalitou
- Dnešní OS umožňují, aby některé ovladače běžely v uživatelském prostoru (jako aplikace), podobně jako je to běžné u μ -jader (viz např. UIO dále)

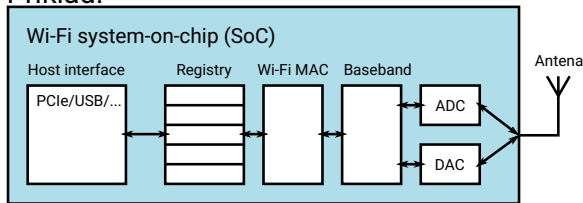
Příklad – ovladač klávesnice

- 1 Aplikace zavolá **getch()/scanf()/...** na standardní vstup
- 2 libc vyvolá systémové volání **read()** na deskriptoru souboru 0 (stdin)
- 3 Standardní vstup je připojen k terminálu (klávesnice + obrazovka) – JOS tedy předá požadavek na vstup **ovladači** klávesnice
 - Ovladač klávesnice spravuje frontu stisknutých znaků (kláves)
- 4 Pokud je fronta prázdná, ovladač **uspí volající vlákno**, jinak se pokračuje krokem 7
 - Interně k tomu použije semafor – vlákno přidá do fronty semaforu
 - Poté zavolá plánovač, aby vybral jiné vlákno, které poběží
- 5 Po stisku klávesy HW vyvolá **přerušení**
- 6 Ovladač klávesnice přerušení obslouží:
 - Přečte z HW (registru) jaká byla stisknuta klávesa a uloží ji do fronty
 - Zavolá operaci up/post na semafor
- 7 Uspané vlákno aplikace se **probudí** (je stále v jádře), vyčte z fronty ovladače stisknuté znaky a zkopíruje je do bufferu v aplikaci.
- 8 Proveďte se **návrat** ze systémového volání zpět do aplikace, funkce getch/scanf se dokončí.

Variabilita a složitost HW

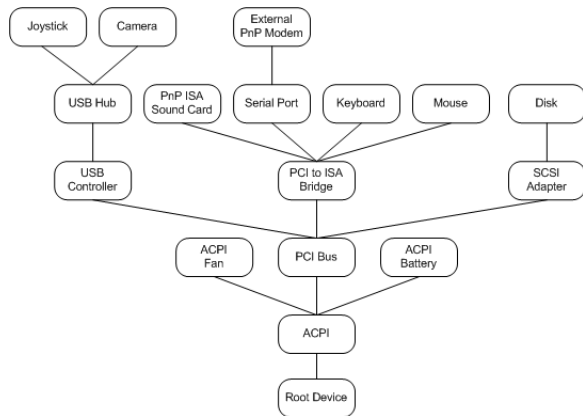
- Dnešní hardware je složitý, zařízení mohou obsahovat stovky či tisíce registrů
- I vývojáři HW mají v oblibě „Ctrl-C, Ctrl-V“ – jeden čip existuje v mnoha variantách, ale všechny mají téměř stejné registry
 - Např. Wi-Fi čip – jedna verze se připojuje k PCIe, jiná k USB
- Struktura ovladačů je modulární – chceme ovladač napsat jednou a používat pro všechny varianty čipu

Příklad:



Hierarchie ovladačů

Topologie hardwaru



Source: Microsoft

- Ovladače reflektují topologii HW
- Každý uzel má vlastní ovladač nezávislý na okolí
- Plug-and-Play (PnP)
 - Ovladač sběrnice (USB, PCI) detekuje připojená zařízení a automaticky načte potřebný ovladač zařízení

Ovladače v Linuxu

- Aplikace mohou s ovladači komunikovat různými způsoby:
 - Nepřímo – např. přes síťové API, práci se soubory, stdin/out
 - Přímě – většina zařízení je reprezentována jako speciální soubor v adresáři /dev (např. sériová linka /dev/ttyUSB0).
 - Pomocí knihoven – aplikace často k souborům v /dev přistupují pomocí knihoven (např. libusb), které nabízejí vyšší úroveň abstrakce, než API OS.
- Ovladač poskytuje aplikacím následující operace (nízkoúrovňové API) pro přímý přístup k ovladačům:
 - **open** – slouží pro „navázání spojení“ aplikace s ovladačem
 - **read** – čtení dat ze zařízení (např. zkuste si spustit `hexdump /dev/input/mice`)
 - **write** – zápis dat do zařízení (např. `tty; echo XXX > /dev/pts/3`),
 - **ioctl** – vše ostatní, co není čtení či zápis, často nastavování (man `ioctl_list`, `ioctl_tty`, ...)
 - **close** – ukončení komunikace s ovladačem

Nejjednodušší ovladač

Ovladač virtuálního zařízení /dev/null

```
#define NULL_MAJOR 1          /* dev major number */
#define NULL_MINOR 3         /* dev minor number */

ssize_t read_null(struct file *file, char *buf, size_t count, loff_t *ppos)
{
    return 0;
}

ssize_t write_null(struct file *file, const char *buf, size_t count, loff_t *ppos)
{
    return count;
}

const struct file_operations null_fops = {
    .read = read_null,
    .write = write_null,
};

void init()
{
    register_chrdev(NULL_MAJOR, NULL_MINOR, "null", &null_fops)
}
```

Trochu zjednodušeno; skutečná implementace ovladače /dev/null: <https://elixir.bootlin.com/linux/v5.9.10/source/drivers/char/mem.c#L673>

Druhý nejjednodušší ovladač

Ovladač virtuálního zařízení /dev/zero

```
#define ZERO_MAJOR 1          /* dev major number */
#define ZERO_MINOR 5         /* dev minor number */

ssize_t read_zero(struct file *file, char *buf, size_t count, loff_t *ppos)
{
    memset(buf, 0, count);
    return count;
}

ssize_t write_zero(struct file *file, const char *buf, size_t count, loff_t *ppos)
{
    return count;
}

const struct file_operations zero_fops = {
    .read = read_zero,
    .write = write_zero,
};

void init()
{
    register_chrdev(ZERO_MAJOR, ZERO_MINOR, "zero", &zero_fops)
}
```

Komunikace mezi ovladači (Linux)

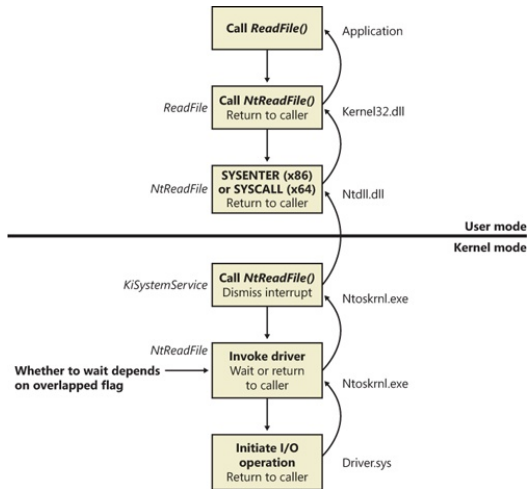
- Linux je monolitické jádro.
- Jednotlivé ovladače se volají vzájemně úplně stejně, jako se volají funkce v uživatelských aplikacích.
- Často se funkce nevolá přímo, ale přes ukazatel
 - Např. Každý ovladač si registruje ukazatel na funkci, která se má vyvolat, když aplikace zavolá `read()` (viz `struct file_operations` na předchozích slidech).
- Data se předávají skrze argumenty funkcí (buď přímo nebo pomocí ukazatelů).

Přístup k ovladačům ve Windows

- Z pohledu aplikace konceptuálně podobné Linuxu:

	Linux	Windows
Otevření ovladače	open	CreateFile
Operace s ovladačem	read, write, ioctl	ReadFile, DeviceIoControl, ...
Uzavření ovladače	close	CloseHandle
Jmenný prostor	/dev/	\\.\
Příklad	/dev/ttyUSB0	\\.\COM6

Přístup k ovladačům ve Windows

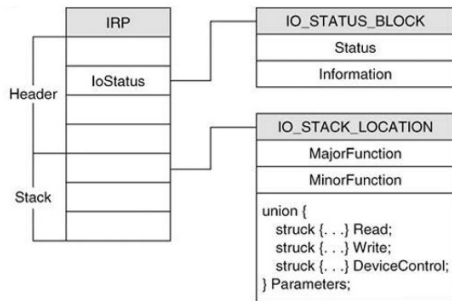


Source: Microsoft

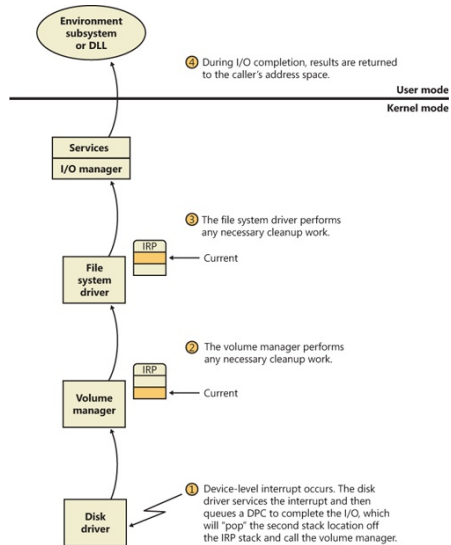
Komunikace mezi ovladači v jádře Windows

- Ovladače ve Windows nepoužívají přímé volání funkcí, ale komunikují pomocí předávání zpráv
- Windows Driver Model je navržen tak, aby bylo teoreticky možné pouštět ovladače v oddělených adresních prostorech
- Kvůli rychlosti ale běží většina ovladačů v jednom monolitickém adresním prostoru jádra.
- Zprávy, které si ovladače vyměňují, se nazývají **I/O request packet (IRP)**

Cesta IRP jádrem



- IRP se alokuje jen jednou
- Každý ovladač „po cestě“ má svůj slot
- File system vyplní slot pro volume manager a pošle IRP dál.
- Po dokončení požadavku IRP „cestuje“ zpět (obr. vpravo).



Ovladače v uživatelském prostoru

- Chyba v ovladači může způsobit pád systému
- Nekvalitní ovladače jsou také zdrojem mnoha bezpečnostních problémů
- Ovladače v uživatelském prostoru:
 - Podporovány jak Linuxem (UIO), tak Windows
 - Spouštěny jako běžná aplikace
 - Přístup k registrům HW: Pomocí volání *mmap()*
 - Obsluha přerušení – OS upozorní aplikaci, pokud nastalo přerušení
 - UIO subsystém v Linuxu:

```
int uio = open("/dev/uio0", ...);  
read(uio, ...); // waits for interrupt  
handle_interrupt();
```
 - Při chybě ovladače ho lze jednoduše restartovat
 - Ostatní aplikace nevolají ovladač pomocí systémových volání, ale pomocí meziprocesní komunikace (např. fronty zpráv)
- OS založené na mikrojádře mají (téměř) všechny ovladače v uživatelském prostoru