# Operační systémy

Systémový pohled na OS - OS především jako správce prostředků počítače (CPU, operační paměť, disková paměť, I/O zařízení), Koordinátor, řídící složka - řídí spouštění programů, zabraňuje chybám   
a vzájemnému ovlivňování

Uživatelský pohled na OS - Dnes používáme typicky desktopy/notebooky vyhrazené pro jednoho uživatele. Dříve často terminály, OS plní požadavky programů řady uživatelů.

Definice OS – je jádro, to co dá výrobce do krabice.

Primární cíle: uživatelská přívětivost, efektivní využití zdrojů, ne všechny podmínky/cíle však implikují jasné způsoby návrhu/implementace (bezchybnost, spolehlivost)

Paralelní systémy

* Úzce vázané systémy. Několik vzájemně komunikujících CPU sdílející jednu paměť a hodinový signál Výhody: vyšší propustnost systému, ekonomické využití počítače
* SMP symetrický multiprocesorový systém. Všechny procesory jsou si rovné. Na všech běží stejná kopie OS. SMP dnes podporuje řada OS včetně Linuxu, Windows, FreeBSD apod.
* AMP – asymetrický multiprocesorový systém. Každý procesor – specifický úkol např. jeden procesor plánuje ostatním práci, nebo určité typy procesů běží na jednotlivých procesorech

Distribuované systémy - Volně vázané systémy

* Každý CPU má vlastní paměť.
* Nekomunikují tedy spolu sdílenou pamětí, ale pomocí komunikačních spojů (od speciálních vysokorychlostních sběrnic až po klasické komunikační linky)
* Výhody: sdílení zdrojů (tiskárny, diskové kapacity), vyšší spolehlivost
* Architektury: Klient-server – řada klientů komunikuje s jedním (nebo více) servery, Peer-to-peer sítě – všechny počítače jsou rovnocenné

Real-Time systémy

* Pracují v reálném čase
* Pro speciální aplikace např. vstřikování v automotoru
* Pracují s pevně stanovanými časovými limity
* Dělení HARD(přísné) OS proces přijme nebo odmítne. SOFT (tolerantní) procesy s vyšší prioritou mají přednost. Vhodné pro multimedia

Mobilní systém

* Omezená paměť, Relativně pomalé procesory, omezená baterie

# Historie

První počítače

* Obrovské stroje – patra budov
* Drahé
* I-O zařízení: děrné štítky, děrné pásky, magnetické pásky
* Job: program + data + řídící informace
* Jeden uživatel: programátor a operátor v jedné osobě
* OS neustále v paměti. Úkolem předávat z jednoho jobu na druhý
* Využití počítače nebylo efektivní

Multiprogramování

* Zvyšuje využití CPU. Přidělování CPU jednotlivým úlohám
* V paměti je zároveň několik úloh současně (ne všechny, ochrana úloh navzájem)
* CPU je přidělen úloze, jakmile úloha požádá o I/O operaci, je úloha pozastavena a CPU dostává jiná úloha. Plánovací algoritmus vybírá úlohy. Dokud úloha nepožádá o I/O tak má CPU k dispozici

Multitasking:

* Značně komplexní, správa a ochrana paměti, synchronizace procesů, cpu plánovací algoritmy

## Základní vlastnosti počítačového systému

* I/O zařízení typicky mají vlastní vyrovnávací paměť
* CPU a I/O zařízení mohou pracovat paralelně např. řadič disku může ukládat data a cpu počítat. Musí se neustále synchronizovat

### Proces bez přerušení

* Neustálý cyklus – loop, získej další instrukci, proveď instrukci, end loop
* Pokud chci provést I/O operaci musím se periodicky ptát I/O zařízení
* CPU není v těchto chvílích efektivně využito

### Proces s přerušením

* Základní cyklus obohacen o kontrolu příznaku přerušení – loop, získej další inst., je-li nastaven příznak přerušení ulož adresu právě prováděného kódu a začni provádět obslužné rutin, endloop

### Přerušení

* Signál od I/O zařízení, že se stalo něco, co by OS měl zpracovat
* Přichází asynchronně - OS nemusí aktivně čekat na událost
* Další přerušení jsou maskována, aby nedocházelo ke ztrátě přerušení
* OS je řízený přerušeními
* Při příchodu přerušení je spuštěna obslužná rutina OS
* Přerušení může generovat jak HW, tak SW

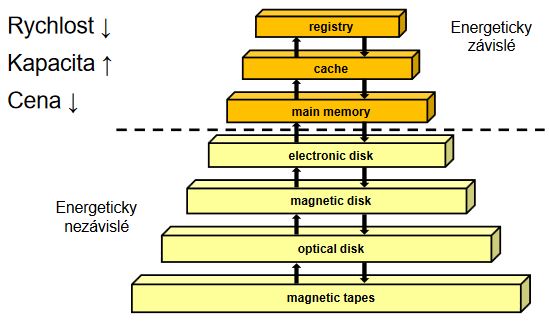
### I/O přístupy

* Synchronní – proces požádá o I/O operaci, řízení se uživatelskému procesu vrací až po ukončení I/O operace. Výhody: jednoduchost Nevýhody: možná neefektivnost
* Asynchronní – proces požádá o I/O operaci a řízení se procesu vrací okamžitě. Výhody: je možné paralelně pracovat s několika I/O. Nevýhody: komplexnější systém, tabulka stavů I/O

### C:\Users\zakja\AppData\Local\Microsoft\Windows\INetCache\Content.Word\Výstřižek.jpgDMA

* Způsob jak rychle přenášet data mezi I/O a pamětí
* CPU může požádat řadič o přenos celého bloku dat
* Po přenosu se o tom dozví pomocí přerušení
* V době přenosu dat soupeří I/O řadič a CPU o přístup do paměti

### Struktura primární paměti

* Jediná větší paměť, kterou může CPU adresovat přímo
* Energeticky závislá

### Struktura sekundární paměti

* Energ. Nezávislá
* Vysoká kapacita
* Relativně pomalá doba přístupu
* Magnetický disk

### Cache, mezipaměť

* Použití rychlejší paměti pro uchování posledně použitých dat z paměti pomalejší
* Case procesoru vs. Hlavní paměť
* Hl. paměť vs. Disková paměť
* Pokud jsou v cachi, tak se použijí, pokud ne, tak se přenesou i s okolními daty

### Režimy procesoru

* Běžný způsob ochrany je dvojí režim činnosti – uživatelský a privilegovaný
* Některé instrukce jdou jen v privilegovaném např. instrukce pro I/O
* Z privilegovaného do uživatelského se CPU dostane instrukcí naopak při zpracování přerušení

### Ochrana paměti

* Minimálně musíme chránit vektor přerušení a rutiny obsluhy přerušení, každému procesu vyhradíme jeho paměť, jinou paměť nemůže proces používat (ochranu zajišťuje CPU, např. báze a limit. Proces má přístup jen k adresám báze + 0 až báze + limit)
* Přístup k nepovoleným adresám způsobí přerušení

### Ochrana CPU

* Časovač generuje přerušení
* Přerušení obsluhuje OS ten rozhodne co dál
* Časovač může generovat přerušení pravidelně
* Časovače v PC: PIT, RTC, HPET

# Struktura a rozhraní OS

### Komponenty OS

* Správa procesů
* Správa operační paměti
* Správa souborů
* Správa I/O zařízení
* Správa sekundárních pamětí
* Správa síťových služeb
* Ochranný systém
* Interpret příkazů

### Správa procesů

* Je jednotkou práce systému. Procesy jádra OS a uživatelské procesy
* Potřebují zdroje – procesor, paměť, soubory. Zdroje se alokují při spouštění nebo při běhu
* Vytváří a ukončuje systém. A uživatel. procesy, potlačuje a obnovuje procesy, mechanismy pro synchronizaci

### Správa operační paměti

* Při spuštění procesu musíme program nahrát do paměti
* Později si může vyžádat dodatečnou paměť
* Jakmile proces skončí OS musí veškerou paměť uvolnit
* Aktivita: znalost, která část paměti je využívána a kým, alokace a dealokace paměti, který proces a kdy zavést do paměti

### Správa souborů

* Soubor – kolekce souvisejících informací
* I/O na kterých jsou soubory uloženy jsou různého typu. Optické, magnetické disky
* Aktivita: vytváření a rušení souborů a adresářů, základní manipulace se soubory a adresáři, zálohování

### Správa I/O zařízení

* Ovladače jednotlivých HW komponent
* Řízení bufferů, kešování, spooling

### Správa sekundární paměti

* Disky
* Obvykle formou souborů
* Aktivita: správa volného místa, přidělování místa, plánování činnosti

### Správa síťových služeb

* Komunikace je řízena protokolem
* Snaha o transparentnost: API podobné jako přístup k souborům

### Ochranný systém

* Chránit procesy navzájem
* Ochrana je mechanismus, který řídí přístup programů, procesů a uživatelů ke zdrojům počítačového systému.
* Rozlišuje autorizované a neautorizované použití prostředků

### Interpret příkazů

* Comman-line neboli shell.
* Považujeme GUI
* Některý součástí jádra

### Interní služby OS

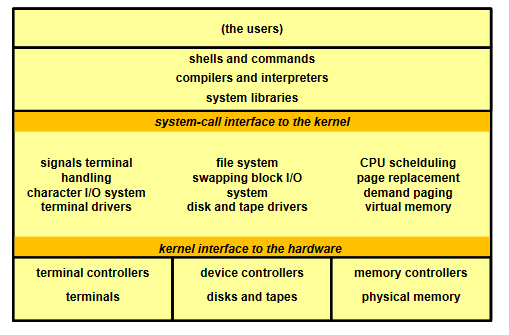
* Zabezpečuje efektivní provoz OS
* Alokace zdrojů – plánovací algoritmy pro přidělování CPU, účtování, ochrana

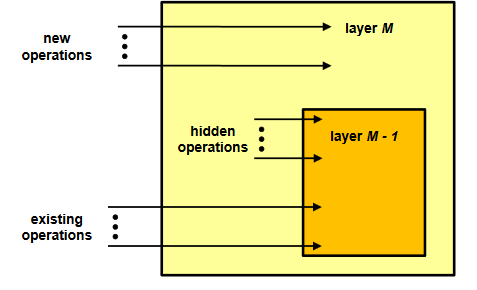
### Systémové volání

* Rozhraní mezi uživatelským procesem a OS. Instrukce assembleru. Vyšší program. Jazyky obsahují systémová volání (např. open, write)
* Různé OS a různé HW mívají různé způsoby jak volat služby OS
* Existují standardy - POSIX, Win32 (win API)
* Podle standardu bude kód přeložitelný na kompatibilních platformách. V praxi mnoho výjimek.
* WIN API – programátorské rozhraní není definováno na úrovni systémového volání jádra. Nižší vrstva ntdll.dll
* LINUX – systémové volání přes instrukci syscall, číslo sys. Volání v registru eax, výsledek v eax

# Principy výstavby OS

### Vnitřní struktura OS

* Jedno velké monolitické jádro, modulární – hierarchický přístup, malé jádro a samostatné procesy
* MS – DOS – využívá služeb: spuštěných rezidentních programů, operačního systému, biosu, přímo HW
* Cíl- maximální možná funkcionalita v co nejmenším prostoru
* Výsledek – modulová architektura není aplikovaná. Jednotlivé komponenty nejsou separovány a uspořádány
* UNIX – skládá se ze systémových programů a jádra (vše co je pod rozhraním volání systému a nad fyzickým hardware, funkce z oblasti systému souborů, plánování CPU. Vrstvová architektura existuje, ale hodně funkcí na jedné úrovni
* XENIX – microsoft koupil. V 80. letech nejrozsirenejsi linux



### Hierarchická vrstvová architektura

* OS se dělí do jistého počtu vrstev
* Každá vrstva je budována na funkcionalitě nižších vrstev. Každá využívá funkce a služby n -1 vrstvy
* Nejnižší hardware, nejvyšší uživatelské rozhraní
* Řeší problém velké složitosti velkého systému
* Každá úroveň řeší podmnožinu funkcí
* Nižší vrstva nemůže požadovat provedení služeb vyšší vrstvy
* Používají se přesně definované rozhraní
* Výhoda - modularita OS
* Nevýhoda – vyšší režie, pomalejší vykonávání systém. Volání
* Aby byl efektivní, tak pouze omezený počet úrovní pokrývající vyšší funkcionalitu

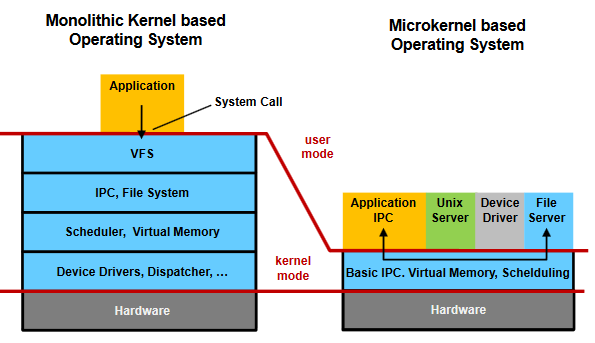
### Provádění služeb v OS

* Klasický OS je prováděn jako samostatná entita v privilegovaném režimu. Procesy jen uživatelské programy
* Služba se provádí jako součást jádra a v rámci procesů
* Lze celý OS provádět v kontextu uživatelského procesu
* Přerušení vyvolává implicitně pouze přepnutí režimu z uživatel. Do privileg. Ne změnu kontextu
* K tomu dochází jen když je to nutné z hlediska plánování
* Pro volání procedur v jádře se používá zásobník
* Program a data OS sdílí všechny uživatelské procesy

### Služby v procesově konstruovaném OS

* OS je souhrnem systémových procesů
* Jádro je separuje, ale neumožňuje jim synchronizaci a komunikaci
* Snaha o provádění co nejmenší části kódu v privilegovaném režim
* Mikrojádro
* Snadno implementované na multiprocesorových systémech

### Struktura s mikrojádrem

* Malé jádro plnící pouze několik funkcí: primitivní správa paměti, komunikace mezi procesy
* Většina funkcí z jádra se přesouvá do uživatelské oblasti (ovladače HW, virtualizace paměti)
* Výhody: snadná přenositelnost, vyšší bezpečnost, flexibilita
* Nevýhody: zvýšená režie – volání nahrazeno výměnou zpráv mezi procesy
* OS s mikrojádrem: Mach 80. léta
* Windows NT používá hybridní strukturu. Další vývoj těchto subsystémů. Win32 se nyní jmenuje Windows API a zahrnuje API na 64bitových systémech

### Linux modularita

* Při běhu přidávat kód – moduly LKM
* Přesto je Linux. Jádro monolitické. Moduly běží v privilegovaném režimu. Jde o modularitu kódu ne jádra
* LKM umožňují přidávání funkčností za běhu, snižují paměťové nároky jádra (nahrávají moduly, které potřebujeme). Možnost změny chování jádra
* Moduly se většinou nepodepisují, ale kouká se na licence. REdHat podepisuje kvůli bezpečnému Bootu
* Modifikace jádra za běhu: možnost přímo číst/měnit paměť jádra za běhu

### Linux: Tabulka systémových volání

* Nejjednodušší způsob jak implementovat rootkit je modifikovat tabulku rutin obsluhujících systémová volání.
* Snaha omezit možnost LKM modifikovat tabulku systémových volání
* Dnes není tento symbol exportován a není tak možné LKM přímo použít

### Windows: Modularita

* Můžeme vkládat ovladače do jádra za běhu
* Běží v privilegovaném režimu

### Podpisy ovladačů

* Jde o spolehlivost systému, kvalitu kódu běžícího v privilegovaném režimu, bezpečnost jádra, DRM
* Snaha snížit množství kódu v jádře

### Implementace systému

* Tradičně býval napsán OS v assembleru
* Častěji se píší v C a C++ - lze naprogramovat rychleji, výsledek je kompaktnější, srozumitelnější, lépe se ladí, přenositelnější

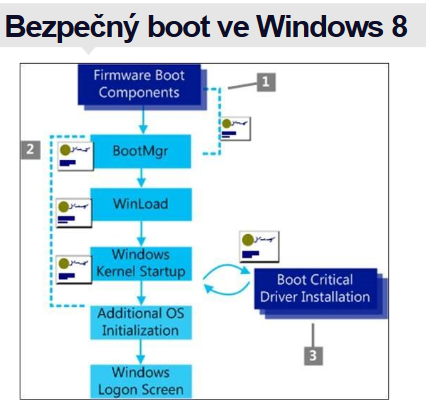
### SYSGEN

* Operační systém je navržen, tak aby mohl běžet na jisté třídě architektur
* OS musí být konfigurovaný na konkrétní sestavu
* Program SYSGEN – získává informace týkající se konkrétní konfigurace konkrétního hardwarového systému
* Bootování – spuštění činnosti počítače zavedením jádra a předáním řízení na vstupní bod jádra pro spuštění činnosti
* Bootstrap program – program uchovaný v ROM, který je schopný nalézt jádro a zavést ho do paměti a spustit jeho provedení

### Bootování IBM PC

* Provede se inicializace HW
* Na základě uložené konfigurace zjistíme z kterého zařízení se má OS zavést
* V případě pevného disku se spustí kód uložený v MBR. Ten spustí boot a ten načte soubory s jádrem OS do paměti
* Bios kontroluje HW
* Z vybraného zařízení se získá a spustí zavaděč boot loader
* Spuštění zavaděče. Nejprve se zavede přístup k disku. Potom zbytek
* Výběr OS a parametrů – GRUP je schopen pracovat s ex2, ex3 souborovými systémy. LILO souborové systémy nezná a pracuje s přímými adresami souborů na disku

### Bootování LINUXU

* Spuštění jádra, z disku přečten obraz jádra, dekomprimuje se, provede se základní inicializace, spustí se jádro
* Jádro – nastaví systém (ovladače přerušení, inicializace zařízení), vytvoří proces Init (číslo 1)

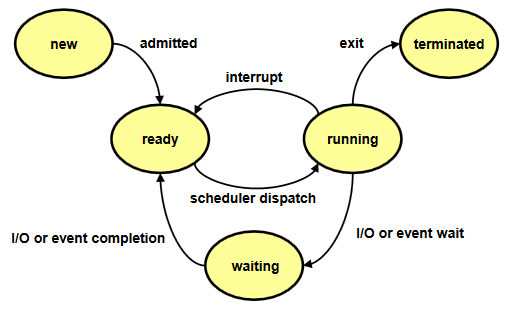
### GPT

* MBR je limitován velikostí disku 2TB
* Intel vyvinul nový standart GUID
* Podporována na většině OS
* Informatice pro členění disku

### Bezpečný boot

* Snaha o bezpečné zavedení OS
* Ochrana zavádění komponent OS, prevence rootkitů

# Procesy

* Společné pojmenování pro spuštěný program je proces (task)
* Vlákno je pro dílčí proces v rámci procesu
* Proces obsahuje: čítač instrukcí, zásobník, datovou sekci, program
* Hierarchie procesů – rodič, potomek, sourozenci (vytvořeni jedním rodičem)
* Prokládáním procesů maximalizujeme využití procesoru a minimalizujeme dobu odpovědi
* Stavy procesu: **nový** (právě vytvořený), **běžící**(procesor vykonává jeho instrukce), **čekající**(čeká na určitou událost), **připravený**(čeká na přidělení času procesoru), **ukončený**(ukončil své provádění)
* Informace OS o precesoru si ukládá do Process Control Block – tabulka obsahující informace potřebné pro definici a správu procesu (stav procesu, čítač instrukcí, registry procesoru, informace potřebné pro správu paměti, info pro správu I/O, účtovací informace)
* Přepnutí kontextu: vyžádá se služba, akceptuje se některé asynchronní přerušení. Když OS potřebuje CPU z procesu X na Y musí. Uchovat stav původně běžícího procesu a zavést stav nově běžícího procesu
* Přepnutí kontextu představuje režijní strátu. Dopa přepnutí závisí na konkrétním HW.
* Fronty plánování procesů: **Fronta úloha**(seznam procesů v OS), **Fronta připravených procesů**, **Fronta na zařízení** (seznam procesů čekající na I/O), **Fronta odložených procesů** (seznam procesů čekající na místo v hlavní paměti)

### Strategický plánovač

* Nemusí mít všechny úlohy, které chci spustit v operační paměti
* Fronta všech úloh může být značně dlouhá a plánovač musí rozhodnout, které půjdou do paměti
* Úkoly (dlouhodobého) plánovače: vybírá, které procesy zařadit mezi připravené procesy, nemusí být super rychlí, moc se nespouští

### Krátkodobý plánovač

* Přiděluje procesor připraveným procesům
* Je spouštěn často, musí být rychlý

### Odkládání procesů

* Každý proces musíme jednou zavést do RAM
* Příliš mnoho procesů v RAM však snižuje výkonnost
* Některé musí odložit na disk
* Zavedeny kvůli tomu dva nové stavy: odložený čekající, odložený připravený

### Střednědobý plánovač

* Vybírá, které zařadit mezi odložené procesy a které odložené mezi připravené
* Náleží do správy operační paměti

### Vytvoření procesu

* Rodič vytváří potomky, potomci mohou vytvářet další potomky
* Sdílení zdrojů: rodič a potomek sdílí zdroje rodiče nebo jen nějakou množinu nebo nesdílí nic
* Běh: mohou běžet souběžně nebo rodič čeká na ukončení potomka

### Ukončení procesu

* Proces provede poslední příkaz a sám požádá OS o ukončení – výstupní data procesu se předají rodiči. Zdroje procesu se uvolní z paměti
* O ukončení procesu žádá jeho rodič: potomek překročil stanovovanou kvótu nebo již ho dále nepotřebuje

### Kooperující procesy

* Nezávislé procesy – nemohou se ovlivňovat
* Kooperující – mohou ovlivnit běh jiných procesů
* Přínosy: sdílení informací, urychlení výpočtu, pohodlí při programování
* Příklady úloh kooperace: klient – server

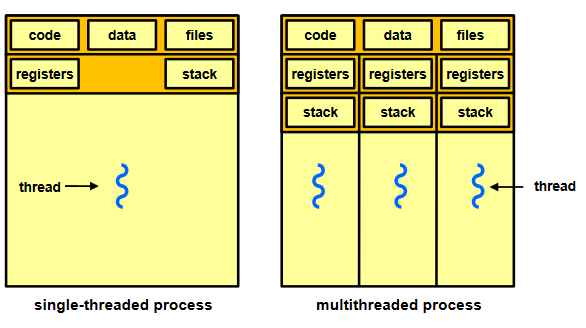
### MS-DOS

* Neumožnuje spouštět procesy paralelně
* Služby OS: spusť proces a čekej na ukončení potomka, ukonči proces, zjisti návratovou hodnotu procesu

### UNIX

* OS s preemptivním multitaskingem
* Služby OS: fork – vytvoří nový proces jako kopii rodiče, exit – ukončí proces, wait – čeká na ukončení potomka

# Vlákna

* Program – soubor definovaného formátu obsahující instrukce, data, …
* Proces – systémový objekt charakterizovaný svým paměťovým prostorem
* Vlákno (sled) – objekt, který vzniká v rámci procesu, je viditelný pouze uvnitř procesu a je charakterizován svým stavem
* Model – jen procesy – jednotka plánovací činnosti i vlastnící zdroje
* Model – procesy a vlákna: Proces (jednotka vlastnící zdroje), vlákno(.. plánovací činnosti)
* Vlákno si udržuje – zásobník, PC (program counter), registry, TCB
* Může přistupovat k paměti a ostatním zdrojů procesu. Když něco jedno vlákno změní globálně uvidí to všechny ostatní vlákna procesu. Soubor otevřený jedním vláknem mají k dispozici ostatní vlákna
* Využití u multiprocesorových strojů
* Jednodušší programování. Příklad: jedno vlákno provádí uživatelem zadaný úkol a druhé to vykresluje na obrazovku
* 1:1 UNIX systém (MS-DOS) – pojem vlákno neznámý, každé vlákno je procesem s vlastním adresovým prostorem a s vlastními prostředky
* 1:M Windows XP – v rámci 1 procesu lez vytvořit více vláken. Proces je vlastníkem zdrojů

### Výhody vláken

* Vlákno se vytvoří rychleji než proces a taky se rychleji ukončí
* Rychle ji se přepíná než mezi procesy, jednoduší programování
* Příklad: síťový server – musí vyřizovat řadu požadavků klientů, 1 vlákno zobrazuje menu a čte vstup od uživatele a pro vytváření každého požadavku

### Problém konzistence

* Program se skládá z několika vláken, které běží paralelně
* Výhody: když jedno čeká na ukončení I/O, může běžet jiné vlákno procesu. Vlákna jednoho procesu sdílí paměť a deskriptory
* Konzistence – vlákna jedné aplikace se proto musí mezi sebou synchronizovat
* Příklad: 3 proměnné ABC. 2 vlákna: T1, T2. Vlákno T1 počítá C = A+B. vlákno T2 přesouvá hodnotu X z A do B. Pokud to udělají zaráz, tak T1 může získat špatný výsledek.

### Stavy vláken

* Běží, připravený, čekající
* Ukončení procesu ukončuje všechny vlákna

### Vlákna na uživatelské úrovni

* ULT – správa vláken se provádí prostřednictvím vláknové knihovny „Threads library“
* Jádro o jejich existenci neví. Přepojování mezi vlákny nepožaduje provádění funkcí jádra
* Plánování přepínání vláken je specifické pro konkrétní aplikace. Aplikace si volí pro sebe nejvhodnější algoritmus
* Threads library obsahuje funkce pro: vytváření a rušení vláken, plánování běhu vláken
* Co dělá jádro pro vlákna na uživatelské úrovni: neví o aktivitě vláken, proto manipuluje s celými procesy. Když některé vlákno zavolá službu jádra, je blokován celý proces, dokud se služba nesplní
* Výhody: přepojování mezi vlákny nepožaduje provádění jádra, plánování je specifické pro konkrétní aplikaci (volí si algoritmus), ULT mohou běžet pod kterýmkoliv OS
* Nevýhody: většina volání služby OS způsobí blokování celého procesu (všech vláken), jádro může přidělovat procesor pouze procesům, dvě vlákna stejného procesu nemohou běžet na dvou procesorech

### Vlákna na úrovni jádra KLT

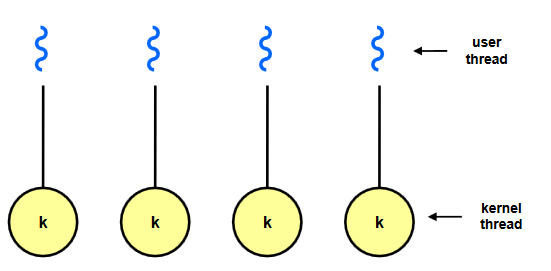
* Správu vláken podporuje jádro, nepoužívá se „thread library“
* Používá se API pro vláknové služby jádra, informace o kontextu procesů a vláken udržuje jádro, přepojování mezi vlákny aktivuje jádro
* Příklady: Linux, Solaris, Windows, XP,7,8
* Výhody:jádro může současně plánovat běh více vláken stejného procesu na více procesorech, k blokování dochází na úrovni vlákna, i programy jádra mohou mít multivláknový charakter
* Nevýhody: přepojování mezi vlákny je pomalejší, při přepnutí vlákna 2x přepíná režim procesoru

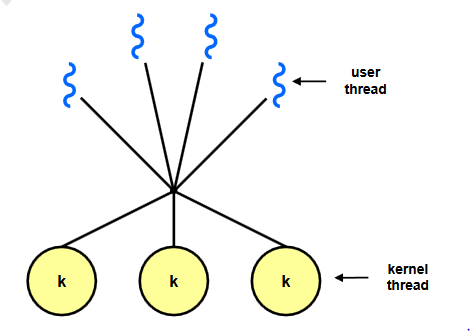
### Kombinace vláken ULT/KLT

* Vytvářejí se v uživatelském prostoru
* Většina plánování a synchronizace se dělá v uživ. Prostoru
* Programátor může nastavit počet vláken na úrovni jádra
* Lze kombinovat přínosy obou přístupů

### Modely

* N:1 – více ULT a zobrazuje do 1 KLT (na systémech které KLT nepodporují)
* 1:1 – každý ULT se zobrazuje do 1 KLT – Windows 7, 8, XP
* N:M více ULT se může zobrazovat do více KLT – Solaris 2





# Plánování CPU

* Multiprogramování zvyšuje využití CPU, pokud jeden proces čeká na dokončení I/O, může jiný proces CPU využít
* Proces obvykle střídá části využívající CPU a vyžadující I/O. Při zahájení I/O je proces zařazen mezi čekající na událost. Po skončení I/O se dostává mezi připravené

### Krátkodobý plánovač – dispečer

* Vybírá proces, kterému bude přidělen CPU
* Proces musí být zaveden v operační paměti a musí být připravený
* Nepreemptivní plánování (bez předbíhání) – proces přechází ze stavu běžícího na čekající nebo končí
* Preemptivní plánování (s předbíháním) – proces ze stavu běžící do připravený nebo čekající do stavu připravený

### Dispečer

* Výstupní modul krátkodobého plánovače nebo plánovač sám, který předává procesor procesu vybranému krátkodobým plánovačem
* Předání zahrnuje: přepnutí kontextu, přepnutí režimu procesoru na uživatelský režim, skok na odpovídající místo v uživatelském programu pro opětovné pokračování v běhu procesu
* Dispečerské zpoždění: Doba, kterou potřebuje dispečer pro pozastavení běhu jednoho procesu a start běhu jiného procesu

### Kritéria plánování

* Využití CPU (maximalizace): cílem je udržení CPU v kontinuální užitečné činnosti
* Propustnost (maximalizace): počet procesů, které dokončí svůj běh za jednotku času
* Doba obrátky (minimalizace): doba potřebná pro provedení konkrétního procesu
* Doba čekání (minimalizace): doba, po kterou proces čekal ve frontě připravených procesů
* Doba odpovědi (minimalizace): dobo od zadání požadavku do první reakce

### Algoritmus FCFS

* „Kdo dřív přijde, ten dřív mele“
* 3 procesy každý má nějakou hodnotu. Lze vypočítat průměrná hodnota. Sečteme je a vydělíme počtem procesů
* Krátké procesy následující po dlouhém procesu ovlivňuje konvojový efekt

### Algoritmus SJF

* Shorted-job-first
* Musíme znát délku příštího požadavku na dávku CPU pro každý proces
* Vybírá se proces s nejkratším požadavkem na CPU
* Dvě varianty: nepreemptivní, bez přerušení (Jakmile se CPU předá vybranému procesu, tento nemůže být předběhnut žádným jiným procesem, dokud CPU dávku nedokončí) a preemptivní, s předbíháním (jakmile se ve frontě připravených procesů objeví proces s délkou dávky CPU kratší, než je doba zbývající k dokončení dávky právě běžícího procesu, je běžící proces předběhnut. Nazývá se to také **SRTF**
* Je to optimální algoritmus, dává minimální průměrnou délku

### Prioritní plánování

* S každým procesem je spojeno prioritní číslo: pč preference procesu pro výběr příštího běžícího procesu. CPU přiděleno procesu s největší prioritou. Většinou odpovídá nejnižší číslo
* Nepreemptivní, bez předbíhání: jakmile proces získá přístup k CPU nemůže být předběhnut jiným procesem
* Preemptivní: jakmile se ve frontě připravených procesů objeví proces s vyšší prioritou, tak je předběhnut
* SJF je prioritní plánování
* Stárnutí: procesy s nižší prioritou se nemusí nikdy provést. Řešení: priorita se časem zvyšuje

### Round Robin RR

* Každý proces dostává CPU na malou jednotku času – časové kvantum: desítky až stovky ms
* Po uplynutí této doby je běžící proces předběhnut nejstarším procesem ve frontě připravených procesů a zařazuje se na konec této front
* Pokud je ve frontě n připravených procesů a q je časové kvantum, pak proces získává 1/n doby CPU, najednou nejvýše q časových jednotek
* Žádný proces nečeká na přidělení CPU déle než (n-1)\*q časových jednotek
* Výkonnostní hodnocení: q velké -> ekvivalent FIFO, q malé -> velká režie, v praxi musí být q dostatečně velké s ohledem na režii přepínání kontextu
* Typicky delší průměrná doba obrátky než SJF, ale doba odpovědi výrazně kratší

# Správa paměti

* Pro běh procesu je nutné, aby program, který je vykonáván byl umístěn v operační paměti
* Z programu se stává proces provedením řady kroků: naplnění tabulek, umístění do OP,..
* Program je složen z částí se vzájemně odlišnými vlastnostmi: moduly s instrukcemi jsou označovány „jen ke spuštění“, datové modelu jsou buďto „read-only“ nebo „read/write“, některé moduly jsou „private“ jiné public
* Více procesů může sdílet společnou část paměti, aniž by se tím porušovala ochrana paměti

### Vázání adres

* Při kompilaci – umístění v paměti je známé. Lze generovat absolutní kód
* Při zavádění – umístění není známé v době kompilace. Generuje se přemístitelný kód
* Za běhu – jestliže proces může měnit svoji polohu během provádění, vázání se zpožďuje na dobu běhu. Musí být dostupná hardwarová podpora

### Memory – management unit

* Hardwarový modul převádějící logické adresy na fyzické
* Uživatelský program pracuje s logickými adresami, uživatelský program nevidí fyzické adresy

### Adresový prostor

* Logický adresový prostor (LAP), fyzický adresový prostor (FAP)
* LAP – logická adresa je dána adresou ve strojovém jazyku, generuje CPU
* FAP – fyzická adresa je akceptovaná operační pamětí
* LAP a FAP se shodují v době kompilace a v době zavádění
* Mohou být rozdílné při vázání v době běhu

### Dynamické zavádění

* Programový kód se nezavádí, dokud není zavolán
* Dosahuje se lepšího využití paměti
* Užitečná technika při velkých modulových programech s velkými moduly, které tam nejsou často např. plag-in moduly
* Program nevyžaduje žádnou speciální podporu od OS

### Dynamické vázání

* Vázání je odkládáno na dobu běhu
* Pro umístění příslušných knihovních programů rezidentních v operační paměti se používá kód malého rozsahu
* Při zavádění stub nahradí sám sebe adresou skutečné funkce a předá ji řízení
* OS musí kontrolovat, zda funkce je mapována do paměti procesu

### Překryvy, Overlays

* V OP se uchovávají pouze ty instrukce a data, která jsou potřeba po celou dobu běhu
* Technika, která je nutná v případech, kdy je přidělený prostor paměti menší než souhrn potřeb procesu
* Je implementováno programátorem
* Je to značně složité ze strany programátora

### Přidělování paměti

* First-fit: přiděluje se první dostatečně dlouhá volná oblast
* Best-fit: přiděluje se nejmenší dostatečně dlouhá volná oblast
* Worst-fit: přiděluje se největší dostatečná dlouhá volná oblast
* První dvě jsou lepší než poslední z hlediska rychlosti, kvality využití paměti

### Problém fragmentace

* Vnější fragmentace: souhrn volné paměti je dostatečný, ale ne v dostatečné souvislé oblasti
* Vnitřní fragmentace: přidělená oblast paměti je větší než požadovaná velikost, tj. část přidělené paměti je nevyužita
* Snižování vnější fragmentace setřásáním: přesouvají se obsahy paměti s cílem vytvořit velký volný blok (použitelné jen s dynamickou realokací, provádí se za běhu)

### Stránkování

* LAP procesu nemusí být jedinou souvislou sekcí FAP, LAP se zobrazuje do sekcí FAP
* FAP se dělí na sekce zvané rámce (pevná délka – v násobcích 2)
* Udržujeme seznam volných rámců
* Program délky n stránek je umístěn do n rámců
* Překlad logická adresa -> fyzická adresa – pomocí překladové tabulky nastavené OS a interpretované MMU
* Vzniká vnitřní fragmentace, neboť paměť je procesu přidělována v násobcích velikosti rámce

### Tabulka stránek

* Je uložena v operační paměti
* Její počátek a konec je odkazován registrem
* Zpřístupnění údaje/instrukce v operační paměti vyžaduje dva přístupy do operační paměti
* Problém zhoršení efektivnosti dvojím přístupem lze řešit speciální rychlou hardwarovou cache pamětí

### Dvouúrovňová tabulka stránek

* 32-bitový procesor s 4KB stránkou
* Logická adresa: číslo stránky: 20 bitů a adresa ve stránce 12 bitů
* Číslo stránky se dále dělí na: číslo stránky: 10 bitů a adresa v tabulce stránek 10 bitů

### Invertovaná tabulka stránek - hešovací

* 1 položka v PT je každý rámec paměti
* Obsahuje – logickou adresu stránky, informaci o procesu, který stránku vlastní
* Snižuje se velikost paměti potřebné pro uchování PT
* Zvyšuje se doba přístup do PT, indexový přístup je nahrazen prohledáváním

### Segmentování

* Logická adresa je dvojice (segment s, offset d)
* ST - Tabulka segmentů (base – počáteční adresa, limit – délka segmentu)
* STBR register – odkaz na ST v paměti
* Relokace – dynamická, pomocí ST

# Virtuální paměť

* Separace LAP a FAP
* Ve FAP se mohou nacházet pouze části programů nutné pro bezprostřední řízení
* LAP může být větší než FAP
* Adresové prostory lze sdílet mezi jednotlivými procesy
* Lze efektivněji vytvářet procesy
* Techniky implementace: Stránkování na žádost, Segmentování na žádost

### Běh procesů ve virtuální paměti

* Část procesu umístěnou ve FAP nazýváme „rezidentní množinou“
* Odkaz mimo rezidentní množinu způsobuje přerušení výpadek stránky
* OS označí proces za „čekající“
* OS spustí I/O operace a provede nutnou správu paměti pro zavedení odkazované části do FAP
* Mezitím běží jiný proces
* Po zavedení odkazované části se generuje I/O přerušení
* Překlad adresy LAP na FAP se dělá indexováním tabulky PT/ST pomocí CPU

### Vlastnosti

* Ve FAP lze udržovat více procesů najednou – čím více je procesů ve FAP, tím větší pravděpodobnost, že stále bude alespoň jeden připravený
* Lze realizovat procesy požadující více paměti, než umožňuje FAP: aniž se o řešení tohoto problému staral programátor.
* Uložení obrazu LAP v externí paměti. Nepoužívá se standart. systém souborů OS. Používá se speciální partition disku optimalizovaný pro tento účel
* Stránkování / segmentaci musí podporovat hardware
* OS musí být schopen organizovat tok stránek mezi vnitřní a vnější paměť

### Kdy zavádět stránku

* Stránkování na žádost – stránka se zobrazuje do FAP při odkazu na ni, pokud ve FAP není již zobrazena
* Předstránkování – umístění na vnější paměti sousedních stránek v LAP. Princip lokality. Zavádí se více stránek, než se žádá. Vhodné při inicializaci procesu

### Kam stránku zavést?

* Segmentace – best, first, worstfit
* Stránkování – nemusíme řešit
* Kombinace těchto dvou – nemusíme řešit

### Kterou stránku nahradit?

* Uplatňuje se v okamžiku, kdy je FAP plný
* Typicky v okamžiku zvýšení stupně multitaskingu
* Politika nahrazování: určení oběti, nelze každá stránka obětovat (jsou zamčeny, typicky buffery)

### Stránkování na žádost

* Stránka se zavádí do FAP jen když je potřebná
* Přínosy: méně I/O operací, menší požadavky na paměť, rychlejší reakce
* Kdy je stránka potřebná? Byla odkazovaná, legální reference (nachází se ve FAP, přeloží se logická adresa na fyzickou), nelegální reference (porušení ochrany) -> abort procesu

### Výpadek stránky

* Pokud se stránka nenachází ve FAP je aktivován OS pomocí přerušení „page fault“
* OS zjistí nelegální referenci např. signál SIGSEGV. Zjistí legální, ale stránka chybí, tak ji zavede
* Zavedení stránky – získání prázdného rámce, zavedení stránky do tohoto rámce
* Pak se opakuje instrukce, která způsobila „page fault“

### Volný rámec

* Pokud není žádný rámec aktuálně označen jako volný, musíme nějaký uvolnit
* Nalezneme oběť – nepotřebnou stránku ve FAP
* Je – li nutné před uvolněním stránky ji zapíšeme na disk
* Potřebujeme algoritmus pro hledání „oběti“ a řešení náhrady

### Princip lokality

* Odkazy na instrukce programu a na data mají tendenci tvořit shluky
* Časová lokalita a prostorová lokalita: Provádění programu je s výjimkou skoků a volání podprogramů sekvenční.
* Lze dělat rozumné odhady o částech programu potřebných v nejbližší budoucnosti
* Vnitřní paměť se může zaplnit – něco umístit do FAP, pak znamená nejdříve něco odložit z FAP

### Algoritmus určení oběti

* Čím více rámců máme, tím méně často dojde k výpadku stránky
* Optimální algoritmus – příští odkazy na oběť je nejpozdější ze všech následných odkazů na stránky. Generuje nejmenší počet výpadků. Neimplementovatelný, používá se jen pro srovnání
* Algoritmus LRU – oběť = nejdéle neodkazovaná stránka
* Algoritmus FIFO – oběť = stránka nejdéle zobrazená ve FAP
* Poslední šance - FIFO + vynechání z výběru těch stránek, na které od posledního výběru bylo odkazováno

### FIFO

* Často používané stránky jsou v mnoha případech právě ty nejstarší. Pravděpodobnost výpadku takové stránky je vysoká.
* Algoritmus zbytečně odkládá často používané stránky
* Jednoduchá implementace

### Optimální algoritmus

* Obětí je nejpozdější ze všech následných odkazů na stránky
* Potřebuje znát budoucí posloupnost referencí

### Algoritmus LRU

* Obětí je nejdéle neodkazovaná stránka: Princip lokality říká, že pravděpodobnost jejího brzkého použití je velmi malá
* Výkon blízký optimální strategii
* Implementace: v položkách PT se udržuje čítač posledního přístupu ke stránce, zásobní čísel stránek
* Aproximace: iniciálně je bit nastaven na 0, při každém přístupu je nastaven bit přístupu na 1. Neznáme pořadí přístupu ke stránkám. Víme jen které stránky byly použity

### Druhá šance

* Také nazývané „Máš ještě šanci“
* Výběr oběti cyklické prochází
* Odkazem na stránku stránka se nastaví příznak
* Oběť která nemá nastaven příznak a je na řadě při kruhovém procházení je nahrazovaná
* Experimenty ukazují, že optimalita algoritmu se blíží skutečnému LRU

### Srovnání algoritmů

* OPT – super, ale nutnost znát budoucnost
* LRU – časově čítače u rámců, nulované odkazem, oběť = rámec s nevyšší hodnotou čítače. Imlementace má vysokou režii
* FIFO – snadná implementace Špatná heuristika
* Druhá šance – upravené FIFO, z výběru obětí se vynechává alespoň jednou odkázaná stránka od posledního výběru.