# Kapitola 1

1.1 Co je operační systém.

Z kapitoly 1.2 pojmy: dávka, dávkový systém, multiprogramování, spooling, sdílení času.

1.3 Koncepce OS, procesy a soubory.

1.4 Systémová volání. Rozšíření o skupiny funkcí ze cvičení.

* + **1.1 Co je operační systém.**
    1. **1.1.1 Operační systém jako stroj s rozšířenými možnostmi**
       1. Počítačová architektura je velmi primitivní a obtížně programovatelná, zejména pokud jde o vstupy a výstupy (V/V). Operační systém poskytuje programátorům jednoduché a vysokoúrovňové rozhraní pro práci s hardwarem. Například místo složitého programování diskových operací nabízí OS jednoduché operace se soubory.
       2. **Virtuální stroj**: OS vytváří virtuální prostředí, které je pro programování jednodušší a uživatelsky přívětivější než přímé programování hardware. Tento virtuální stroj ukrývá složitosti hardware a poskytuje uživateli a programátorovi příjemné a jednoduché rozhraní.
       3. **Abstrakce hardware**: Například řadič NEC PD765 používaný ve většině osobních počítačů má 16 příkazů, každý specifikovaný 1 až 9 byty. Tyto příkazy jsou pro čtení a zápis dat, pohyb čtecí hlavy, formátování stopy, inicializaci, identifikaci, reset a kalibraci řadiče i mechaniky. Příkazy READ a WRITE vyžadují 13 parametrů zabalených do 9 bytů, které specifikují údaje jako adresa diskového bloku pro čtení, počet sektorů na stopu, záznamový režim použitý na médiu, oddělovací mezeru mezi sektory a další. Důležité zmínit, že programátor musí také spravovat stav motoru mechaniky, aby se předešlo přehřátí.
    2. **1.1.2 Operační systém jako správce prostředků**
       1. Operační systém spravuje a koordinuje přístup k procesoru, paměti a V/V zařízením mezi různými programy a uživateli. Zajišťuje, aby nedocházelo ke konfliktům a aby byly zdroje efektivně využívány.
       2. **Správa zdrojů**: OS spravuje a koordinuje přístup k procesoru, paměti a V/V zařízením mezi různými programy a uživateli. Zajišťuje, aby nedocházelo ke konfliktům a aby byly zdroje efektivně využívány.
       3. **Ochrana a sdílení**: OS chrání paměť a další zdroje před neoprávněným přístupem a umožňuje sdílení informací mezi uživateli a programy. Například při více uživatelích na jednom počítači OS zajišťuje, aby si uživatelé nepřekáželi a vzájemně se neobtěžovali.
       4. **Řízení přístupu**: OS může přinést pořádek do potenciálního chaosu ukládáním všech výstupů na tiskárnu na disk. Až jeden program skončí, může OS zkopírovat jeho výstup z diskového souboru, kde byl uložen, na tiskárnu, zatímco souběžně mohou ostatní programy pokračovat v generování dalších výstupů.
  + **Z kapitoly 1.2 pojmy: dávka, dávkový systém, multiprogramování, spooling, sdílení času.**
    1. **1.2.2 Druhá generace (1955-65) - tranzistory a dávkové systémy**
       1. **Dávka**: Program nebo skupina programů, které jsou zpracovány počítačem bez interakce s uživatelem. Programátor napíše program na papír, převede jej na děrné štítky a předá obsluze, která jej načte do počítače. Počítač zpracuje dávku a výsledky jsou vytištěny nebo uloženy na výstupní médium. Batch (.bat). Pracujeme v offline rezimu.
       2. **Dávkový systém**: Systém, který zpracovává dávky postupně. Vstupní dávky jsou načteny na magnetickou pásku, která je pak zpracována hlavním počítačem. Výstupy jsou ukládány na pásku a následně tištěny. Tento systém minimalizuje časové ztráty způsobené manuální obsluhou a zvyšuje efektivitu využití počítače.
    2. **1.2.3 Třetí generace (1965-80) - integrované obvody a multiprogramování**
       1. **Multiprogramování**: Technika, která umožňuje, aby více programů běželo současně. Paměť je rozdělena na bloky, každý obsahuje jednu dávku. Když jedna dávka čeká na V/V operaci, jiná může využívat CPU. To zvyšuje efektivitu využití CPU a minimalizuje dobu nečinnosti procesoru.
       2. **Spooling (Simultaneous Peripheral Operation On Line)**: Technika, která umožňuje současné zpracování vstupů a výstupů. Dávky jsou načteny z děrných štítků na disk a zpracovány postupně. Výstupy jsou ukládány na disk a následně tištěny. To eliminuje potřebu přenášet pásky a zvyšuje efektivitu. Spooling také umožňuje, aby více programů současně využívalo stejná V/V zařízení bez konfliktů.
    3. **1.2.4 Čtvrtá generace (1980-nyní) - personální počítače**
       1. **Sdílení času (timesharing)**: Systém, který umožňuje více uživatelům současně pracovat na jednom počítači pomocí terminálů. CPU je alokováno pro různé úlohy na základě krátkých časových úseků, což umožňuje interaktivní práci. Uživatelé mohou rychle ladit své programy a získávat okamžité výsledky. Timesharing zajišťuje, že každý uživatel má pocit, že má počítač k dispozici neustále, i když ve skutečnosti sdílí jeho zdroje s ostatními uživateli.
  + **1.3 Koncepce OS, procesy a soubory.**
    1. **1.3 Koncepce operačního systému**
       1. Rozhraní mezi operačním systémem a uživatelským programem je definováno jako množina „rozšiřujících instrukcí“, které operační systém vykonává. Tyto rozšířené instrukce jsou tradičně nazývány jako systémová volání (system calls). Abychom porozuměli, co operační systém dělá, musíme prozkoumat toto rozhraní zblízka. Volání dostupná v rámci rozhraní se liší mezi jednotlivými operačními systémy, i když základní koncepce je podobná.
       2. Systémová volání MINIXu se dělí zhruba na dvě vymezující kategorie: manipulace s procesy a se soubory.
    2. **1.3.1 Procesy**
       1. Proces je v podstatě vykonávaný program. Každý proces má svůj **adresový prostor**, který obsahuje proveditelný program, jeho data a zásobník. Proces má také skupinu **registrů**, včetně čítače instrukcí, zásobníku a dalších registrů potřebných pro běh programu.
       2. **Správa procesů**: Operační systém pravidelně rozhoduje o zastavení jednoho procesu a spuštění dalšího. Pokud je proces dočasně pozastavený, musí být později znovu spuštěn ve stejném stavu, v jakém byl před zastavením. Informace o procesu jsou uloženy v **tabulce procesů (process table)**.
       3. **Tabulka procesů (process table)**: Tato tabulka je pole nebo zřetězený seznam s položkou pro každý existující proces. Obsahuje všechny informace o procesu, kromě jeho adresního prostoru. Když je proces pozastaven, jeho stav je uložen v této tabulce, aby mohl být později obnoven.
       4. **Vytváření a ukončování procesů**: Procesy mohou vytvářet nové procesy (potomky) a ukončovat je. Potomci mohou vytvářet své vlastní potomky, což vede ke stromové struktuře procesů. Procesy mohou komunikovat a synchronizovat své aktivity pomocí mezi-procesní komunikace (IPC).
       5. **Mezi-procesní komunikace (IPC)**: Procesy často potřebují komunikovat a synchronizovat své aktivity. OS poskytuje mechanismy pro tuto komunikaci, jako jsou signály a roury. IPC umožňuje procesům vyměňovat si data a synchronizovat své činnosti.
       6. **Signály**: Signály umožňují předat informaci běžícímu procesu, který na ni nečeká. Signál způsobí dočasné pozastavení procesu a spuštění speciální funkce pro obsluhu signálu. Po ukončení funkce se proces znovu spustí ve stavu, v jakém byl před signálem.
       7. Každý uživatel má své **UID (user identification)** a každý proces má UID osoby, která jej spustila. Super-user (root) má speciální práva a může porušovat mnoho ochranných mechanismů.
    3. **1.3.2 Soubory**
       1. Operační systém zakrývá zvláštnosti disků a jiných V/V zařízení a nabízí programátorovi přívětivý a jasný abstraktní model souborů nezávislých na zařízení.
       2. **Správa souborů**: Systémová volání jsou potřebná pro vytvoření, smazání, čtení a zápis souborů. Soubory musí být otevřeny před čtením nebo zápisem a po ukončení práce musí být zavřeny.
       3. **Adresáře**: Soubory jsou organizovány do adresářů, což umožňuje jejich seskupování. Systémová volání umožňují vytvoření a smazání adresářů, umístění souborů do adresářů a jejich odstranění.
       4. **Ochrana souborů**: Soubory a adresáře jsou chráněny přístupovými právy, které určují, kdo může soubor číst, zapisovat nebo spouštět. Přístupová práva jsou definována pomocí ochranné masky.
       5. **Montování souborového systému**: MINIX umožňuje připojit souborový systém z výměnného média (např. diskety) do hlavního adresářového stromu.
       6. **Speciální soubory**: Speciální soubory zajišťují přístup k zařízením jako k souborům. Existují blokové speciální soubory pro zařízení s náhodně adresovatelnými bloky (např. disky) a znakové speciální soubory pro zařízení, která přijímají nebo vysílají proud znaků (např. tiskárny, modemy).
       7. **Roury (pipes)**: Roura je typ pseudo-souboru, který umožňuje propojení dvou procesů. Data zapisovaná do jednoho konce roury mohou být čtena z druhého konce.
  + **1.4 Systémová volání. Rozšíření o skupiny funkcí ze cvičení.**
    1. **1.4 Systémová volání**
       1. Systémová volání jsou rozhraní mezi operačním systémem a aplikačními programy. Umožňují programům komunikovat s operačním systémem a požadovat služby, jako je správa souborů, procesů a zařízení. MINIX má 53 systémových volání, která jsou rozdělena do šesti skupin. Níže jsou popsána nejdůležitější systémová volání, zejména ta, která jsou relevantní pro vaše cvičení.
    2. **Správa procesů**
       1. **fork**: Vytvoří nový proces (potomka), který je identický s rodičovským procesem. Potomek dědí kopii adresního prostoru rodiče.
       2. **waitpid**: Čeká na ukončení specifikovaného potomka a vrací jeho stav.
       3. **wait**: Starší verze waitpid, čeká na ukončení libovolného potomka.
       4. **execve**: Nahrazuje obraz aktuálního procesu novým programem. Používá se k vykonání jiného programu.
       5. **exit**: Ukončí aktuální proces a vrátí stavový kód operačnímu systému.
       6. **pipe**: Vytvoří rouru, což je unidirekcionální kanál pro mezi-procesní komunikaci.
       7. **dup2**: Duplikuje popisovač souboru na specifikovaný popisovač, často se používá pro přesměrování stdin a stdout.
       8. **exec**: Rodina funkcí (např. execv, execvp) pro nahrazení aktuálního procesu jiným programem.
    3. **Správa souborů**
       1. **open**: Otevře soubor a vrátí popisovač souboru.
       2. **close**: Uzavře otevřený soubor.
       3. **read**: Čte data ze souboru do bufferu.
       4. **write**: Zapisuje data z bufferu do souboru.
       5. **access**: Ověří přístupová práva k souboru.
       6. **stat**: Získá informace o souboru.
       7. **lstat**: Podobné jako stat, ale pro symbolické odkazy.
       8. **fstat**: Získá informace o otevřeném souboru.
    4. **Signály**
       1. **kill**: Zašle signál procesu.
       2. **sigaction**: Definuje obsluhu signálu.
       3. **sigprocmask**: Přebere nebo změní masku signálů.
       4. **sigpending**: Vrátí množinu blokovaných signálů.
       5. **sigsuspend**: Vymění masku signálů a pozastaví proces.
    5. **Poll a sokety**
       1. **poll**: Monitoruje více souborových deskriptorů a čeká na události.
       2. **socket**: Vytvoří koncový bod pro komunikaci.
       3. **bind**: Přiřadí adresu k soketu.
       4. **listen**: Označí soket jako pasivní, který bude přijímat připojení.
       5. **accept**: Přijme připojení na soketu.
       6. **connect**: Naváže spojení se vzdáleným soketem.
    6. **Vlákna**
       1. **pthread\_create**: Vytvoří nové vlákno.
       2. **pthread\_exit**: Ukončí vlákno.
       3. **pthread\_join**: Čeká na ukončení specifikovaného vlákna.
       4. **Semafory a sdílená paměť**
       5. **sem\_init**: Inicializuje anonymní semafor.
       6. **sem\_wait**: Sníží hodnotu semaforu (čeká, pokud je nula).
       7. **sem\_post**: Zvýší hodnotu semaforu.
       8. **sem\_open**: Otevře pojmenovaný semafor.
       9. **sem\_close**: Uzavře pojmenovaný semafor.
       10. **shm\_open**: Otevře sdílenou paměť.
       11. **shm\_unlink**: Odstraní sdílenou paměť.
    7. **Fronta zpráv**
       1. **msgget**: Vytvoří nebo získá frontu zpráv.
       2. **msgsnd**: Odesílá zprávu do fronty.
       3. **msgrcv**: Přijímá zprávu z fronty.

# Kapitola 2

2.1, 2.1.1 Úvod do procesů, model procesu, stavy procesu.

2.1.2 Implementace procesů, vlákna (pouze implementovaná systémem).

2.2, 2.2.1, 2.2.2, IPC, souběh (race condition), kritická sekce, podmínky dobrého řešení.

2.2.3, Vzájemné vyloučení: zákaz přerušení, zamykací proměnná, TLS.

2.2.4, 2.2.5, Uspání a probuzení, semafor, monitor, fronta zpráv.

2.3 Klasické IPC problémy: výroba a spotřeba, spící holič, večeřící filozofové, čtenáři a spisovatelé.

2.4 Plánování procesů.

* + **2.1, 2.1.1 Úvod do procesů, model procesu, stavy procesu.**
    1. **2.1 Úvod do procesů**
       1. Moderní počítače musí provádět více úloh současně. Zatímco běží program uživatele, počítač může číst data z disku, vypisovat text na obrazovku a tisknout na tiskárnu. V multiprogramovém systému se CPU přepíná mezi programy po desítkách nebo stovkách milisekund, což vytváří dojem paralelismu, známý jako pseudoparalelismus. Skutečný paralelismus je dosažen v multiprocesorových systémech s více procesory a sdílenou pamětí.
    2. **2.1.1 Model procesu**
       1. V tomto modelu jsou všechny spustitelné programy, včetně operačního systému, organizovány jako řada sekvenčních procesů. Proces je běžící program společně s hodnotami programového čítače, registrů a proměnných. Každý proces má svůj vlastní virtuální CPU, i když reálně CPU neustále přepíná mezi procesy. Tento systém rychlého a neustálého přepínání se nazývá multiprogramový.
    3. **Hierarchie procesů**
       1. Operační systém musí poskytnout metodu na vytvoření všech potřebných procesů. V MINIXu se proces vytváří systémovým voláním fork, které vytvoří identickou kopii volajícího procesu. Všechny procesy v systému tvoří jeden strom s procesem init jako kořenem.
    4. **Stavy procesu**
       1. Proces může být v jednom ze tří stavů:
          1. **Running (běžící)**: Proces právě využívá CPU.
          2. **Ready (připravený)**: Proces je připraven běžet, ale CPU je přiděleno jinému procesu.
          3. **Blocked (zablokovaný)**: Proces nemůže běžet, dokud nepřijde nějaká externí událost.
       2. Mezi těmito stavy jsou čtyři možné přechody:
          1. Proces zjistí, že nemůže pokračovat a přejde do stavu Blocked.
          2. Plánovač rozhodne, že proces běžel dostatečně dlouho a přepne jej do stavu Ready.
          3. Plánovač vybere proces ze stavu Ready a přepne jej do stavu Running.
          4. Proces čekající na externí událost přejde ze stavu Blocked do stavu Ready, když událost nastane.
  + **2.1.2 Implementace procesů, vlákna (pouze implementovaná systémem).**
    1. **2.1.2 Implementace procesů**
       1. Pro implementaci modelu založeného na procesech musí mít operační systém tabulku nebo pole struktur, které se nazývá **tabulka procesů** (process table). Každý záznam v této tabulce obsahuje informace o alokaci paměti, stavu otevřených souborů, účtovacích a plánovacích informacích a dalších detailech o procesu. Tyto informace jsou nezbytné pro uložení stavu procesu při přepínání mezi procesy.
       2. V MINIXu je správa procesů, paměti a souborů implementována v samostatných modulech, přičemž tabulka procesů je rozdělena na bloky a každý modul se stará o své položky v tabulce.
       3. **Klíčové položky tabulky procesů zahrnují:**
          1. **Registry**: Ukazatel kódového segmentu, ukazatel datového segmentu, ukazatel na BSS segment.
          2. **Stav procesu**: Stav procesu, čas spuštění, čas použití CPU, čas příštího alarmu.
          3. **Správa souborů**: UMASK, root dir, pracovní adresář, deskriptory souborů.
       4. **Postup pri** **obsluze přerušení** a **plánování procesů**:
          1. **Hardware uloží na zásobník čítač instrukcí atd.**: Když dojde k přerušení, hardware automaticky uloží aktuální hodnoty čítače instrukcí a dalších registrů na zásobník.
          2. **Hardware vezme nový čítač instrukcí z tabulky přerušení.**: Hardware načte nový čítač instrukcí z tabulky přerušení, což umožní přechod na obslužnou rutinu přerušení.
          3. **Podprogram v assembleru uloží registry.**: Obslužná rutina v assembleru uloží všechny registry do tabulky procesů, aby se zachoval stav přerušeného procesu.
          4. **Dále nastaví nový zásobník.**: Nastaví se nový zásobník pro obsluhu přerušení.
          5. **Spustí obsluhu přerušení v jazyce C.**: Po provedení základních operací v assembleru se spustí obslužná rutina přerušení napsaná v jazyce C.
          6. **Plánovač označí čekající proces jako připravený.**: Plánovač změní stav procesu, který čekal na dokončení přerušení, na připravený.
          7. **Plánovač rozhodne, který proces bude následně spuštěn.**: Plánovač vybere další proces, který bude spuštěn, na základě aktuálního stavu a priorit.
          8. **Obsluha v C se vrátí zpět do assembleru.**: Po dokončení obsluhy přerušení v jazyce C se řízení vrátí zpět do assembleru.
          9. **Podprogram v assembleru znovu spustí nový aktuální proces.**: Assemblerový podprogram obnoví registry a další stavové informace nového procesu a spustí jej.
          10. Tento postup je klíčový pro správné přepínání mezi procesy a zajištění, že každý proces může být správně obnoven a pokračovat ve své činnosti po přerušení.
    2. **Vlákna**
       1. V tradičních procesech je jedno řídící vlákno a jeden programový čítač v každém procesu. Moderní operační systémy však podporují **vícenásobná řídící vlákna** uvnitř procesu, nazývaná vlákna nebo lehké procesy.
       2. Co musí evidovat vlakna – viz obr str 104
       3. **Příklad použití více vláken:**
          1. **Souborový server**: Může obdržet příkaz ke čtení a zápisu do souboru a odeslat zpět požadovaná data. Pro zlepšení výkonnosti má server vyrovnávací paměť aktuálně používaných souborů v paměti. Když přijde požadavek, je předán vláknu k vyřízení. Pokud se vlákno zablokuje čekáním na přenos z disku, jiné vlákno může stále pracovat.
       4. **Implementace vláken:**
          1. **Tabulka vláken**: Některé položky z tabulky procesů jsou pro každé vlákno v separátní tabulce vláken, například registry, status a programový čítač.
          2. **Plánování vláken**: Operační systém musí mít seznam všech vláken, podobně jako seznam procesů. Přepnutí vlákna v adresním prostoru uživatele je rychlejší než přepnutí voláním jádra.
       5. **Problémy s vlákny:**
          1. **Fork**: Když má rodič více vláken, má je mít potomek také? Pokud ne, proces nemusí fungovat správně.
          2. **Sdílení datových struktur**: Co se stane, když vlákno zavře soubor a další ho stále používá?
          3. **Signály**: Některé signály jsou logicky svázány s vláknem, zatímco jiné nikoliv.
  + **2.2, 2.2.1, 2.2.2, IPC, souběh (race condition), kritická sekce, podmínky dobrého řešení.**
    1. **2.2 Mezi-procesní komunikace (IPC)**
       1. Procesy často potřebují komunikovat s dalšími procesy. Například v shellu se musí výstup jednoho procesu předávat do druhého procesu. Tato komunikace by měla být dobře strukturovaná a bez použití přerušení. Existují tři základní problémy IPC:
          1. Jak může proces poslat informace jinému procesu?
          2. Jak zajistit, aby si procesy vzájemně nepřekážely při kritických aktivitách?
          3. Jak zajistit správné pořadí při vzájemných závislostech procesů?
    2. **2.2.1 Souběh (Race Conditions)**
       1. V operačních systémech mohou procesy sdílet společnou paměť, kde mohou číst i zapisovat. Sdílená paměť může být hlavní paměť nebo sdílený soubor. Příklad souběhu je tiskový server, kde procesy vkládají jména souborů do tiskového adresáře. Pokud dva procesy současně čtou a zapisují do sdílené proměnné, může dojít k souběhu, což vede k nekonzistentním výsledkům. Tento problém nastává, když finální výsledek závisí na tom, který proces běžel kdy.
       2. **Příklad souběhu:**
          1. Představme si tiskový adresář s pozicemi číslovanými 0, 1, 2, …, každá schopná nést jméno souboru. Dvě sdílené proměnné out a in ukazují na následující soubor pro tisk a na následující volné místo v adresáři. Pokud proces A přečte in a uloží hodnotu 7 do lokální proměnné nextslot, může dojít k přerušení a přepnutí na proces B, který také přečte in a uloží 7 do nextslot. Proces B pak uloží jméno svého souboru na pozici 7 a aktualizuje in na 8. Když se proces A znovu spustí, uloží jméno svého souboru na pozici 7, čímž přepíše hodnotu uloženou procesem B, a nastaví in na 8. Tím je tiskový adresář konzistentní, ale proces B přijde o svůj výstup.
    3. **2.2.2 Kritická sekce (Critical Section)**
       1. Abychom se vyhnuli souběhu, musíme zajistit, aby více než jeden proces nemohl současně číst a modifikovat sdílená data. To se nazývá vzájemné vyloučení (mutual exclusion). Kritická sekce je část programu, kde se přistupuje ke sdílené paměti. Musíme zajistit, aby procesy nebyly nikdy ve stejné kritické sekci současně.
       2. **Podmínky dobrého řešení**
          1. **Žádné dva procesy nesmí být současně uvnitř stejné kritické sekce.**
          2. **Na řešení nesmí mít vliv počet a rychlost CPU.**
          3. **Žádný proces mimo kritickou sekci nesmí blokovat jiný proces.**
          4. **Žádný proces nesmí zůstat čekat nekonečně dlouho na kritickou sekci.**
  + **2.2.3, Vzájemné vyloučení aktivním čekáním (busy waiting): zákaz přerušení, zamykací proměnná, TLS.**
    1. **Zákaz přerušení**
       1. Nejjednodušší řešení pro dosažení vzájemného vyloučení je zákaz všech přerušení procesem, který právě vstoupil do své kritické sekce, a jejich znovu-povolení před opuštěním kritické sekce. Když jsou přerušení zakázána, nemůže dojít k přerušení od časovače, což znamená, že CPU nebude přepnuto na jiný proces. Proces tak může bezpečně aktualizovat sdílenou paměť bez rizika zásahu jiného procesu.
       2. Tato metoda však není vhodná pro uživatelské procesy, protože pokud by uživatelský proces zakázal přerušení a nikdy je znovu nepovolil, mohlo by to vést k zablokování celého systému. Navíc v multiprocesorových systémech zákaz přerušení ovlivní pouze procesor, který instrukci vykonal, zatímco ostatní procesory mohou nadále přistupovat ke sdílené paměti.
    2. **Zamykací proměnné (Lock Variables)**
       1. Další metodou je použití zamykacích proměnných. Sdílená zamykací proměnná je inicializována na 0. Když proces chce vstoupit do své kritické sekce, nejprve otestuje zámek. Pokud je zámek 0, proces jej nastaví na 1 a vstoupí do kritické sekce. Pokud je zámek již 1, proces čeká, dokud se zámek neuvolní.
       2. Tato metoda však trpí stejným problémem jako předchozí příklad s tiskovým adresářem. Pokud dva procesy současně otestují a nastaví zámek, mohou oba vstoupit do kritické sekce současně, což vede k souběhu.
    3. **TSL instrukce (Test and Set Lock)**
       1. TSL (Test and Set Lock) je speciální instrukce, která atomicky testuje a nastavuje zámek. Instrukce TSL přečte hodnotu proměnné do registru a do této proměnné uloží nenulovou hodnotu. Operace čtení a uložení je atomická, což znamená, že žádný jiný procesor nemůže přistupovat do paměti, dokud instrukce neskončí.
       2. Pro použití TSL instrukce se používá sdílená proměnná lock. Když je lock 0, proces může nastavit lock na 1 pomocí TSL instrukce a vstoupit do kritické sekce. Po dokončení práce v kritické sekci proces nastaví lock zpět na 0.
  + **2.2.4, 2.2.5, Uspání a probuzení, semafor, monitor, fronta zpráv.**
    1. **2.2.4 Uspání a probuzení (Sleep and Wakeup)**
       1. Obě řešení, Petersenovo i s použitím TSL, jsou správná, ale jejich nedostatkem je aktivní čekání. Podstatou těchto řešení je, že pokud proces potřebuje vstoupit do kritické sekce, zkontroluje, zda je vstup povolen. Pokud ne, zůstane proces ve smyčce čekat na povolení. To nejen plýtvá procesorovým časem, ale může mít i neočekávané efekty, jako je problém převrácené priority, kdy proces s vysokou prioritou aktivně čeká a blokuje proces s nízkou prioritou, který je v kritické sekci.
       2. Jedním ze základních mezi-procesních mechanismů pro blokování je dvojice systémových volání **SLEEP** a **WAKEUP**. SLEEP uspí volající proces, dokud jej jiný proces neprobudí voláním WAKEUP. Tento mechanismus se používá například v problému výrobce–spotřebitel, kde výrobce vkládá položky do bufferu a spotřebitel je odebírá. Pokud je buffer plný, výrobce se uspí, dokud spotřebitel neodebere položku a neprobudí jej. Podobně, pokud je buffer prázdný, spotřebitel se uspí, dokud výrobce nevloží novou položku a neprobudí jej.
    2. **2.2.5 Semafory (Semaphores)**
       1. Dijkstra navrhl použití celočíselné proměnné, nazývané **semafor**, která může mít hodnotu 0 nebo kladné číslo, indikující počet nezpracovaných probuzení. Semafor má dvě operace: **DOWN** (snižuje hodnotu semaforu a uspí proces, pokud je hodnota 0) a **UP** (zvyšuje hodnotu semaforu a probudí jeden uspáný proces, pokud nějaký čeká). Tyto operace jsou nedělitelné, což znamená, že jednou započatá operace se semaforem nedovolí žádnému jinému procesu k semaforu přistupovat, dokud se operace neukončí.
       2. Semafory řeší problém ztraceného signálu, který se může vyskytnout u SLEEP a WAKEUP. Použití semaforů v problému výrobce–spotřebitel zahrnuje tři semafory: **full** (počítá obsazené položky), **empty** (počítá volné pozice) a **mutex** (zajišťuje vzájemné vyloučení při přístupu k bufferu). Full začíná na 0, empty na maximální velikosti bufferu a mutex na 1. Výrobce volá DOWN na empty a mutex před vložením položky do bufferu a UP na mutex a full po vložení. Spotřebitel volá DOWN na full a mutex před odebráním položky a UP na mutex a empty po odebrání.
    3. **Monitory**
       1. Monitory jsou vyšší úroveň synchronizačního prostředku, navržená Hoarem a Brinchem Hansenem. Monitor je skupina podprogramů, proměnných a struktur, které jsou zabaleny do speciálního typu modulu. Proces může volat podprogramy z monitoru, ale nemůže přímo přistupovat k vnitřním datovým strukturám. Monitor zajišťuje, že v jakékoliv situaci smí být aktivní pouze jeden proces.
       2. Monitory používají podmíněné proměnné se dvěma operacemi: **WAIT** (pozastaví volající proces) a **SIGNAL** (probudí proces čekající na podmíněnou proměnnou). Pokud podprogram monitoru zjistí, že nemůže pokračovat, zavolá WAIT. Jiný proces probudí svého spícího partnera voláním SIGNAL. Hoare navrhl, aby SIGNAL spustil probuzený proces a pozastavil druhý, zatímco Brinch Hansen požadoval, aby SIGNAL byl posledním příkazem podprogramu.
    4. **Fronta zpráv (Message Queue)**
       1. Předávání zpráv je metoda IPC, která používá dvě operace: **SEND** a **RECEIVE**. SEND zašle zprávu do daného cíle a RECEIVE ji přijme. Pokud není žádná zpráva k dispozici, příjemce se zastaví, dokud nějaká nepřijde. Předávání zpráv se používá například v problému výrobce–spotřebitel, kde výrobce posílá zprávy s daty spotřebiteli a ten posílá prázdné zprávy zpět výrobci. Tento systém zajišťuje, že výrobce a spotřebitel mohou efektivně komunikovat bez sdílené paměti.
  + **2.3 Klasické IPC problémy: výroba a spotřeba, spící holič, večeřící filozofové, čtenáři a spisovatelé.**
    1. **2.3.4 Problém výroby a spotřeby (Producer-Consumer Problem)**
       1. Problém výroby a spotřeby, známý také jako problém omezeného bufferu (bounded buffer problem), je klasický synchronizační problém, který ilustruje potřebu správné mezi-procesní komunikace a synchronizace. Tento problém zahrnuje dva typy procesů: výrobce a spotřebitele, kteří sdílejí společný omezený buffer.
       2. **Popis problému**
          1. **Výrobce**: Produkuje data a vkládá je do bufferu.
          2. **Spotřebitel**: Odebírá data z bufferu a zpracovává je.
       3. Buffer má omezenou kapacitu, což znamená, že výrobce musí čekat, pokud je buffer plný, a spotřebitel musí čekat, pokud je buffer prázdný.
       4. **Synchronizační mechanismus**
       5. Pro řešení tohoto problému se často používají semafory, které zajišťují vzájemné vyloučení a synchronizaci mezi výrobci a spotřebiteli. Tři hlavní semafory jsou:
          1. **Mutex**: Zajišťuje vzájemné vyloučení při přístupu k bufferu.
          2. **Empty**: Počítá volné pozice v bufferu.
          3. **Full**: Počítá obsazené pozice v bufferu.
       6. **Výrobce**: Po vyrobení položky čeká, pokud je buffer plný, a poté vstoupí do kritické sekce, kde vloží položku do bufferu. Po opuštění kritické sekce zvýší počet obsazených míst.
       7. **Spotřebitel**: Čeká, pokud je buffer prázdný, a poté vstoupí do kritické sekce, kde odebere položku z bufferu. Po opuštění kritické sekce zvýší počet volných míst.
    2. **Spící holič (Sleeping Barber Problem)**
       1. V holičství je jeden holič, jedno holičské křeslo a několik židlí pro čekající zákazníky. Pokud nejsou zákazníci, holič spí. Když přijde zákazník, probudí holiče. Pokud holič stříhá a přijde další zákazník, posadí se, pokud je volná židle, jinak odejde. Tento problém zahrnuje synchronizaci mezi zákazníky a holičem, aby nedošlo k souběhu.
       2. Řešení používá tři semafory: **customers** pro počítání čekajících zákazníků, **barbers** pro počítání volných holičů a **mutex** pro vzájemné vyloučení. Proměnná **waiting** počítá zákazníky a je kopií **customers**. Když zákazník přijde, zkontroluje, zda je volná židle. Pokud ano, zvýší **waiting** a probudí holiče. Pokud ne, odejde. Holič po probuzení zkontroluje **waiting** a pokud jsou zákazníci, začne stříhat. Po dokončení stříhání se holič vrátí k čekání na dalšího zákazníka.
    3. **Večeřící filozofové (Dining Philosophers Problem)**
       1. Pět filozofů sedí kolem kulatého stolu, každý má před sebou talíř se špagetami a mezi každou dvojicí talířů je jedna vidlička. Filozof potřebuje dvě vidličky k jídlu. Problém je napsat program, který zajistí, že filozofové neuvíznou při pokusu získat vidličky a nebudou hladovět, pokud budou čekat na vidličky.
       2. Jednoduché řešení, kde filozofové berou vidličky současně, může vést k zablokování, pokud všichni vezmou levou vidličku současně. Alternativní řešení, kde filozofové čekají náhodný čas, může vést k hladovění. Správné řešení používá semafory a pole **state** pro sledování stavu filozofů (jedení, přemýšlení, čekání). Filozof může začít jíst pouze tehdy, když žádný jeho soused nejí. Toto řešení zajišťuje maximální paralelismus a zabraňuje zablokování a hladovění.
    4. **Čtenáři a spisovatelé (Readers and Writers Problem)**
       1. Tento problém modeluje přístup k databázi, kde více procesů může číst současně, ale pouze jeden proces může zapisovat. Pokud proces zapisuje, žádný jiný proces nesmí přistupovat k databázi. Řešení zahrnuje synchronizaci, aby čtenáři mohli číst současně, ale spisovatelé měli výhradní přístup při zápisu.
       2. První čtenář získá přístup do databáze a zablokuje semafor **db**. Ostatní čtenáři jen inkrementují čítač **numr**. Poslední čtenář po dokončení čtení odblokuje semafor **db**. Problém nastává, když čtenáři neustále přicházejí a spisovatelé zůstávají zablokováni. Řešení zahrnuje blokování nových čtenářů, pokud čeká spisovatel, což zajišťuje, že spisovatelé nebudou nekonečně čekat.
  + **2.4 Plánování procesů.**
    1. **2.4 Plánování procesů**
       1. V příkladech předchozích kapitol jsme často měli situaci, kdy jeden nebo více procesů bylo připraveno ke spuštění. Když je připraveno více procesů ke spuštění, operační systém musí rozhodnout, který spustí jako první. Ta část operačního systému, která toto rozhodování provádí, se nazývá **plánovač (scheduler)** a použitý algoritmus rozhodování je **plánovací algoritmus (scheduling algorithm)**.
    2. **Cíle plánovače**
       1. Než se podíváme na konkrétní plánovací algoritmy, měli bychom se zamyslet, čeho se vlastně plánovač snaží dosáhnout. Některé možnosti jsou:
          1. **Férovost**: Dát všem procesům rovnocenné šance sdílení CPU.
          2. **Efektivnost**: Udržet procesor trvale vytížený na 100%.
          3. **Doba odezvy**: Minimalizovat odezvu pro interaktivní uživatele.
          4. **Doba běhu**: Minimalizovat čas, po který musí uživatel čekat na vykonání svého zadání.
          5. **Průchodnost**: Maximalizovat počet procesů, které budou vykonány během časové jednotky.
    3. **Preemptivní vs. nepreemptivní plánování**
       1. Plánování může být **preemptivní** nebo **nepreemptivní**. Preemptivní plánování umožňuje operačnímu systému přerušit běžící proces a přepnout na jiný, zatímco nepreemptivní plánování umožňuje procesu běžet, dokud se sám neukončí nebo nezablokuje.
    4. **Plánovací algoritmy**
       1. **Round Robin**: Každý proces má přiřazen časový interval (kvantum), po který smí běžet. Pokud je na konci svého kvanta proces stále běžící, je přepnut CPU na jiný proces. Tento algoritmus je jednoduchý a spravedlivý, ale efektivita závisí na délce kvanta.
       2. **Prioritní plánování**: Každý proces má přiřazenu prioritu a spuštěn je proces s nejvyšší prioritou. Priority mohou být statické nebo dynamické. Dynamické priority mohou být upravovány na základě chování procesů, například procesy vázané na V/V operace mohou mít vyšší prioritu.
       3. **Více front**: Procesy jsou rozděleny do několika prioritních tříd. Procesy s nejvyšší prioritou mají nejkratší časové kvantum a jsou spouštěny před procesy s nižší prioritou. Pokud proces vyčerpá své kvantum, je přesunut do nižší třídy.
       4. **Nejkratší dávka jako první (Shortest Job First)**: Tento algoritmus je určen pro dávkové systémy, kde je doba běhu předem známá. Procesy s nejkratší dobou běhu jsou spouštěny jako první, což minimalizuje průměrnou dobu odezvy.
       5. **Zaručené plánování (Guaranteed Scheduling)**: Tento přístup zajišťuje, že každý uživatel nebo proces dostane spravedlivý podíl procesorového času. Plánovač sleduje, kolik času každý proces spotřeboval, a zajišťuje, že žádný proces nepřekročí svůj podíl.
       6. **Plánování losováním (Lottery Scheduling)**: Procesy dostávají losy a při každém plánování je náhodně vybrán jeden los. Proces s vybraným losem získá CPU. Tento algoritmus umožňuje snadné nastavení priorit přidělením více losů důležitějším procesům.
       7. **Plánování Real-Time**: Real-time systémy vyžadují, aby procesy reagovaly na události v určitých časových intervalech. Existují různé algoritmy pro plánování v reálném čase, jako je **poměrný monotonní algoritmus (rate monotonic algorithm)** a **algoritmus nejbližšího limitu jako první (earliest deadline first)**.
       8. **Dvouúrovňové plánování**: Tento přístup se používá, když není dostatek paměti pro všechny procesy. Některé procesy jsou uloženy na disku a plánovač vyšší priority rozhoduje, které procesy budou přesunuty mezi pamětí a diskem.
    5. **Princip vs. mechanismus**
       1. Je důležité oddělit plánovací mechanismus od plánovacího principu. Mechanismus je implementován v jádře systému, zatímco principy mohou být určeny uživatelskými procesy. To umožňuje uživatelům ovlivňovat plánování svých procesů, aniž by přímo zasahovali do plánovače.
       2. Tímto způsobem plánovač zajišťuje efektivní a spravedlivé využití procesorového času, přičemž bere v úvahu různé potřeby a priority procesů.

# Kapitola 3

3.1, 3.1.1 Principy V/V

3.2 Principy V/V, rozdělení do vrstev, úkoly jednotlivých vrstev.

3.3 Deadlock, co to je, jak může vzniknout, podmínky zablokování, dá se řešit?

* + **3.1, 3.1.1 Principy V/V**
    1. **Principy V/V**
       1. Jedna z hlavních funkcí operačního systému je řízení všech vstupních a výstupních zařízení počítače. Musí vysílat příkazy do zařízení, zachytávat přerušení a reagovat na chyby. Dále se stará o rozhraní mezi zařízeními a zbytkem systému, které by mělo být jednoduché a snadné pro používání. Rozhraní by mělo být stejné pro všechna zařízení (device independence). V/V kód představuje významnou část z celého operačního systému.
    2. **3.1 Principy V/V hardware**
       1. Různí lidé mají na V/V hardware rozdílný pohled. Inženýr elektrotechnik se na ně dívá jako na čipy, dráty, napájení, motor a všechny další komponenty tvořící hardware. Programátory zajímá rozhraní pro programování – příkazy, které zařízení přijímá, funkce, které vykonává, a chyby, které může vrátit zpět. Zde se budeme zabývat programováním V/V zařízení, ne jejich návrhem, výrobou a údržbou. Náš zájem se omezí na to, jak se hardware programuje, ne na jeho vnitřní činnost. Přesto programování mnoha V/V zařízení je úzce spojeno s jejich vnitřními operacemi.
    3. **3.1.1 V/V zařízení**
       1. V/V zařízení se dělí zhruba na dvě kategorie: bloková zařízení (block device) a znaková zařízení (character device).
          1. **Bloková zařízení**: Ukládají informace v blocích stejné velikosti a každý blok má vlastní adresu. Velikost bloku je obvykle od 512 do 32768 bytů. Základní vlastností blokových zařízení je schopnost číst a zapisovat každý blok nezávisle na všech ostatních. Nejobvyklejším blokovým zařízením je disk. Hranice mezi blokově adresovatelným a neadresovatelným zařízením není přesně definována. Například 8 mm pásková DAT mechanika pro zálohování disku může být použita jako blokové zařízení s náhodným přístupem, ale je to zdlouhavé a obvykle se takto nepoužívá.
          2. **Znaková zařízení**: Předávají nebo přijímají proud znaků bez jakékoliv blokové struktury. Nejsou adresovatelná a neznají operaci vyhledávání (seek). Příklady zahrnují tiskárny, sítě, myši, sériové porty a mnoho dalších zařízení, která nejsou podobná diskům.
       2. Toto schéma dělení není dokonalé. Například hodiny nejsou adresovatelné blokově a nemohou generovat nebo přijímat proud dat. Paměťově mapovaná obrazovka tomuto modelu také nevyhovuje. Přesto je model znakových a blokových zařízení nejobecnější a používá se jako základ pro tvorbu software operačního systému komunikujícího s V/V zařízeními. Například souborový systém komunikuje s abstraktním blokovým zařízením a nechává část závislou na zařízení nižší úrovni software, nazývanou ovladač zařízení (device driver).
  + **3.2 Principy V/V, rozdělení do vrstev, úkoly jednotlivých vrstev.**
    1. **Principy V/V software**
       1. Hlavním cílem V/V software je vytvořit systém, který je nezávislý na konkrétním zařízení a poskytuje jednotné, jednoduché a přehledné rozhraní pro uživatele. Tento cíl se dosahuje strukturováním V/V software do několika vrstev, kde každá vrstva má své specifické úkoly.
    2. **Rozdělení do vrstev**
       1. V/V software je rozdělen do čtyř hlavních vrstev:
          1. **Ovladač přerušení**
          2. **Ovladač zařízení**
          3. **Část operačního systému nezávislá na zařízení**
          4. **Uživatelské programy**
    3. **Úkoly jednotlivých vrstev**
       1. **1. Ovladač přerušení**
          1. **Úkoly**:

Zpracovává přerušení generovaná hardwarem.

Pozastavuje procesy, které spustily blokové V/V operace, dokud nejsou dokončeny.

Po dokončení operace odblokuje procesy, které operaci spustily.

* + - * 1. Ovladač přerušení je klíčovou součástí operačního systému, která zajišťuje, že systém může efektivně reagovat na události generované hardwarem. Když zařízení dokončí operaci, generuje přerušení, které je zachyceno ovladačem přerušení. Tento ovladač pak provede potřebné kroky k obsluze přerušení, jako je aktualizace stavových informací a odblokování procesů čekajících na dokončení operace.
      1. **2. Ovladač zařízení**
         1. **Úkoly**:

Zajišťuje komunikaci mezi operačním systémem a hardwarem.

Přijímá abstraktní požadavky z vyšších vrstev a překládá je do konkrétních příkazů pro hardware.

Spravuje frontu nevyřízených požadavků a zajišťuje jejich postupné zpracování.

Kontroluje správnost provedení příkazů a zpracovává chyby.

* + - * 1. Ovladač zařízení je specifický pro každý typ zařízení a obsahuje kód, který ví, jak komunikovat s konkrétním hardwarem. Například ovladač disků ví, jaké příkazy posílat řadiči disku, jak interpretovat stavové informace a jak zpracovávat chyby. Ovladač zařízení také spravuje frontu požadavků, což znamená, že pokud je zařízení zaneprázdněné, nové požadavky jsou zařazeny do fronty a zpracovány později.
      1. **3. Část operačního systému nezávislá na zařízení**
         1. **Úkoly**:

Poskytuje jednotné rozhraní pro ovladače zařízení.

Zajišťuje jednotné pojmenovávání a ochranu zařízení.

Provádí bufferování a alokaci úložného prostoru na blokových zařízeních.

Spravuje přidělení a uvolnění vyhrazených zařízení.

Zpracovává informace o chybách.

* + - * 1. Tato vrstva je zodpovědná za poskytování jednotného rozhraní pro všechny ovladače zařízení, což umožňuje vyšším vrstvám operačního systému a uživatelským programům komunikovat s různými zařízeními stejným způsobem. Tato vrstva také zajišťuje, že zařízení jsou správně pojmenována a chráněna, a provádí operace jako bufferování dat a alokaci úložného prostoru.
      1. **4. Uživatelské programy**
         1. **Úkoly**:

Provádějí systémová volání pro V/V operace.

Používají knihovny pro formátování vstupu a výstupu.

Mohou zahrnovat speciální procesy, jako jsou démoni pro spooling, které spravují vyhrazená zařízení v multiprogramovém prostředí.

* + - * 1. Uživatelské programy interagují s V/V systémem prostřednictvím systémových volání, která jsou implementována v knihovnách. Tyto knihovny poskytují funkce pro formátování vstupu a výstupu a umožňují programům provádět V/V operace bez nutnosti přímé komunikace s hardwarem. Speciální procesy, jako jsou démoni pro spooling, zajišťují efektivní správu vyhrazených zařízení, jako jsou tiskárny, v prostředí s více uživateli.
        2. Toto rozdělení do vrstev umožňuje efektivní správu V/V operací a zajišťuje, že software může být snadno rozšiřován a upravován bez nutnosti zásahů do všech částí systému. Každá vrstva má jasně definované úkoly a odpovědnosti, což usnadňuje údržbu a rozvoj operačního systému.
  + **3.3 Deadlock, co to je, jak může vzniknout, podmínky zablokování, dá se řešit?**
    1. **Deadlock**
       1. Deadlock, nebo také zablokování, je situace, kdy skupina procesů čeká na událost, kterou může vyvolat pouze jiný proces z této skupiny. V důsledku toho žádný z procesů nemůže pokračovat, protože všechny čekají na uvolnění prostředků, které jsou vlastněny jinými procesy ve skupině.
    2. **Jak může vzniknout**
       1. Deadlock může vzniknout, když procesy požadují výhradní přístup k více prostředkům. Pokud například proces A vlastní prostředek X a požaduje prostředek Y, zatímco proces B vlastní prostředek Y a požaduje prostředek X, oba procesy se zablokují, protože každý čeká na uvolnění prostředku, který vlastní ten druhý.
    3. **Podmínky zablokování**
       1. Aby mohlo dojít k deadlocku, musí být splněny následující čtyři podmínky:
       2. **Vzájemné vyloučení (Mutual Exclusion)**: Každý prostředek je buď právě vlastněn jedním procesem, nebo je dosažitelný.
       3. **Podmínka postupného přidělování prostředků (Hold and Wait)**: Proces, který již vlastní některé prostředky, může požádat o další.
       4. **Podmínka neodebrání (No Preemption)**: Dříve přidělené prostředky nemohou být procesu násilím odebrány. Musí být uvolněny procesem, který je vlastní.
       5. **Podmínka čekání v kruhu (Circular Wait)**: Musí existovat kruh dvou nebo více procesů, kde každý čeká na prostředek vlastněný následujícím procesem v kruhu.
    4. **Řešení deadlocku**
       1. Existují čtyři hlavní strategie pro řešení deadlocku:
          1. **Ignorování problému**: Tento přístup, známý jako “pštrosí algoritmus”, předpokládá, že deadlock je vzácný a není třeba se jím zabývat.
          2. **Detekce a zotavení**: Systém pravidelně kontroluje, zda nedošlo k deadlocku, a pokud ano, ukončí jeden nebo více procesů, aby deadlock odstranil.
          3. **Prevence deadlocku**: Zabránění deadlocku tím, že se zajistí, aby alespoň jedna z podmínek pro vznik deadlocku nebyla nikdy splněna. Například procesy mohou být nuceny požadovat všechny potřebné prostředky najednou, nebo mohou být prostředky očíslovány a procesy mohou požadovat prostředky pouze v určitém pořadí.
          4. **Zamezení deadlocku**: Dynamická analýza požadavků na prostředky a jejich přidělování tak, aby systém nikdy nevstoupil do nebezpečného stavu, který by mohl vést k deadlocku. Příkladem je bankéřův algoritmus, který zajišťuje, že systém zůstane v bezpečném stavu.

# Kapitola 4

4.2.1, 4.2.2, Implementace správy paměti.

4.3 Virtuální paměť (32 nebo 64 bitů), stránkování, jedno a víceúrovňové tabulky stránek.

4.4.x Výměna stránek.

4.6 Segmentace, účel, návaznost na virtuální paměť.

* + **Neni v otazkach ale asi dobre mist alespon trochu uvod**
    1. **4 Správa paměti**
       1. Paměť je klíčový systémový prostředek, který musí být efektivně spravován. I když dnešní počítače mají mnohem větší paměť než dříve, velikost programů roste stejným tempem. Parkinsův zákon říká: „Programy se zvětšují tak, aby zaplnily paměť jim danou.“ Operační systémy musí koordinovat použití různých typů pamětí: rychlé cache, hlavní paměti (RAM) a pomalé diskové paměti.
       2. Správce paměti sleduje využití paměti, alokuje ji procesům, dealokuje ji při ukončení a spravuje odkládání mezi hlavní pamětí a diskem.
    2. **4.1 Základní správa paměti**
       1. Systémy správy paměti se dělí na dvě třídy:
          1. **Systémy bez odkládání a stránkování**: Jednodušší, procesy nejsou přesouvány mezi pamětí a diskem.
          2. **Systémy s odkládáním a stránkováním**: Komplexnější, procesy se přesouvají mezi pamětí a diskem.
    3. **4.1.1 Jednoprocesové programování bez odkládání a stránkování**
       1. **Jednoduché schéma**: Spuštění jednoho programu v jednom okamžiku.
       2. **Organizace paměti**: Operační systém v dolní části RAM nebo v ROM.
       3. **Proces**: Po zadání příkazu se program zkopíruje z disku do paměti a spustí. Po skončení procesu se zobrazí příkazový řádek.
    4. **4.1.2 Víceprocesové programování s pevnými oddíly**
       1. **Více procesů najednou**: Paměť rozdělena na několik oddílů.
       2. **Oddíly**: Mohou být stejně nebo různě velké.
       3. **Přemístění a ochrana paměti**: Řešeno pomocí bázového a mezního registru.
       4. **Příklad**: Používáno v OS/360 na velkých sálových počítačích IBM.
    5. **4.2 Odkládání**
       1. Odkládání (swapping) je technika, kdy jsou procesy kompletně přeneseny do paměti, spuštěny po určitou dobu a následně odloženy zpět na disk. Tato metoda je jednoduchá a efektivní v dávkových systémech, kde je dostatek úloh k udržení CPU vytíženého.
          1. **V systémech sdílejících čas**: Není vždy dostatek paměti pro všechny aktivní procesy, takže nadbytečné procesy jsou uchovávány na disku a dynamicky spouštěny.
          2. **Proměnné oddíly**: Počet, umístění a velikost oddílů se mění dynamicky podle potřeby procesů.
          3. **Setřásání paměti (memory compaction)**: Spojení mezer v paměti do jedné velké oblasti přesunutím všech procesů na začátek paměti. Tato technika je náročná na čas a obvykle se neprovádí.
       2. Odkládání umožňuje efektivní správu paměti v systémech, kde není dostatek paměti pro všechny procesy současně.
  + **4.2.1, 4.2.2, Implementace správy paměti.**
    1. **4.2.1 Správa paměti bitmapami (Memory Management with Bit Maps)**
       1. Při dynamickém přidělování paměti musí operační systém sledovat její využití. Existují dva hlavní způsoby, jak toho dosáhnout: pomocí bitmap (bit maps) a propojených seznamů (linked lists).
       2. **Bit Mapy:**
          1. **Rozdělení paměti**: Paměť je rozdělena na alokační jednotky (allocation units), které mohou mít velikost od několika slov až po několik kilobajtů. Každé alokační jednotce odpovídá jeden bit v bitmapě. Bitová hodnota 0 znamená, že jednotka je volná, a hodnota 1, že je obsazená (nebo naopak).
          2. **Velikost bitmapy**: Velikost alokační jednotky ovlivňuje velikost bitmapy. Menší alokační jednotky znamenají větší bitmapu. Například s alokační jednotkou o velikosti 4 bajty potřebuje 32 bitů paměti jeden bit v bitmapě.
          3. **Prohledávání bitmapy**: Hledání souvislého úseku nul délky k v bitmapě může být pomalé, protože sekvence může být rozložena mezi slovy v bitmapě.
       3. Bit mapy poskytují jednoduchý způsob, jak sledovat využití paměti, protože velikost bitmapy závisí pouze na velikosti paměti a velikosti alokační jednotky. Hlavním problémem je, že prohledávání bitmapy na úseky dané délky je pomalé, což je hlavní argument proti používání bitmap.
    2. **4.2.2 Správa paměti s propojenými seznamy (Memory Management with Linked Lists)**
       1. Další metodou správy paměti je použití propojených seznamů alokovaných a volných paměťových segmentů.
       2. **Propojené seznamy:**
          1. **Struktura seznamu**: Paměť je reprezentována jako propojený seznam segmentů, kde každý segment je buď proces nebo mezera mezi procesy. Každý záznam v seznamu specifikuje mezeru (hole - H) nebo proces (P), počáteční adresu, délku a ukazatel na další záznam.
          2. **Aktualizace seznamu**: Když proces skončí nebo je odložen, aktualizace seznamu je přímá. Segment může mít dva sousedy, což vede ke čtyřem možným kombinacím:

Nahrazení procesu mezerou.

Spojení dvou sousedních mezer.

Spojení mezery a sousedního procesu.

Spojení tří sousedních segmentů do jedné mezery.

* + - 1. Propojené seznamy umožňují efektivní správu paměti a minimalizují fragmentaci. Existuje několik algoritmů pro alokaci paměti:
         1. **První vhodný (First Fit)**: Prochází seznam a najde první dostatečně velkou mezeru. Tento algoritmus je rychlý, protože prohledává jen do prvního úspěchu.
         2. **Další vhodný (Next Fit)**: Pokračuje hledáním od místa, kde naposled skončil. Tento algoritmus dává o trošku horší výsledky než první vhodný.
         3. **Nejlepší vhodný (Best Fit)**: Prohledává celý seznam a vybírá nejmenší vhodnou mezeru. Tento algoritmus je pomalejší než první vhodný a vede k většímu plýtvání pamětí, protože má sklon rozmělnit paměť na malé mezery.
         4. **Nejhorší vhodný (Worst Fit)**: Vybírá největší mezeru, která bude po rozdělení stále použitelná. Tento algoritmus také není nejlepším řešením.
         5. **Rychlý vhodný (Quick Fit)**: Udržuje oddělené seznamy pro často používané velikosti. Tento algoritmus je rychlý, ale může vést k rychlé fragmentaci paměti na malé mezery.
  + **4.3 Virtuální paměť (32 nebo 64 bitů), stránkování, jedno a víceúrovňové tabulky stránek.**
    1. **4.3 Virtuální paměť**
       1. Virtuální paměť umožňuje, aby celková velikost programů, dat a zásobníků překročila velikost dostupné fyzické paměti. Operační systém uchovává v hlavní paměti jen ty části, které jsou právě používány, a zbytek má na disku. Například 16MB program může běžet na stroji s 4MB paměti díky tomu, že se podle potřeby pečlivě vybírá, které části programu uchovávat v paměti a které odkládat mezi pamětí a diskem.
    2. **4.3.1 Stránkování (Paging)**
       1. Většina systémů virtuální paměti používá techniku zvanou stránkování. Virtuální adresy, které programy vytvářejí, se nazývají virtuálními adresami a vytvářejí virtuální adresový prostor. Na počítačích bez virtuální paměti se virtuální adresy posílají přímo na paměťovou sběrnici. Používá-li se virtuální paměť, virtuální adresa jde do jednotky správy paměti (MMU), která mapuje virtuální adresy na fyzické adresy.
       2. **Mapování adres**
          1. Virtuální adresní prostor se dělí na stránky (pages) a fyzická paměť na rámce stránek (page frames). Stránky a rámce stránek jsou vždy stejné velikosti. Přenosy mezi pamětí a diskem jsou vždy realizovány po stránkách. Když se program pokusí o přístup na adresu, virtuální adresa se pošle do MMU, která ji přemapuje na fyzickou adresu.
       3. **Výpadek stránky (Page Fault)**
          1. Pokud se program pokusí použít nemapovanou stránku, MMU oznámí, že stránka není namapovaná, a CPU toto předá operačnímu systému. Tento odchyt se nazývá výpadek stránky. Operační systém vezme nejméně používaný rámec stránky a zapíše jeho obsah zpátky na disk. Poté přenese právě odkazovanou stránku do právě uvolněného rámce stránky, změní mapu a spustí znovu odchycenou instrukci.
    3. **4.3.2 Tabulky stránek (Page Tables)**
       1. Tabulka stránek mapuje virtuální stránky na rámce stránek. Virtuální adresa je rozdělena na číslo virtuální stránky a offset. Číslo virtuální stránky se použije jako ukazatel do tabulky stránek, aby se nalezlo číslo rámce stránky. Číslo rámce stránky je připojeno za offset, což vytvoří fyzickou adresu.
       2. **Víceúrovňové tabulky stránek (Multilevel Page Tables)**
          1. Aby se předešlo nutnosti držet velké tabulky stránek v paměti, používá mnoho počítačů víceúrovňové tabulky stránek. Například 32bitové virtuální adresy mohou být rozděleny na 10 bitů pro první úroveň, 10 bitů pro druhou úroveň a 12 bitů pro offset. Tabulka stránek nejvyšší úrovně obsahuje ukazatele na tabulky stránek druhé úrovně, které obsahují ukazatele na rámce stránek.
       3. **Překlad s nahlédnutím do bufferu (TLB - Translation Lookaside Buffers)**
          1. Většina stránkovacích mechanismů používá TLB, což je malá hardwarová paměť, která mapuje virtuální adresy na fyzické adresy bez potřeby procházet tabulku stránek. Když se objeví výpadek TLB, MMU hledá potřebný odkaz v tabulce stránek, odstraní jeden záznam z TLB a nahradí ho novým.
       4. **Převrácené tabulky stránek (Inverted Page Tables)**
          1. Převrácené tabulky stránek mají jeden záznam na každý rámec skutečné paměti, na rozdíl od jednoho záznamu pro každý virtuální adresní prostor. Přestože ušetří obrovské množství paměti, překlad virtuální na fyzickou adresu je složitější a vyžaduje prohledání celé tabulky při každém odkazu do paměti. TLB může tento proces urychlit.
    4. **Rozdíly mezi 32bitovou a 64bitovou virtuální pamětí**
       1. **32bitová virtuální paměť**
          1. **Adresní prostor**: 32bitové systémy mají maximální adresní prostor 4 GB (2^32 bajtů).
          2. **Tabulky stránek**: Typicky používají dvouúrovňové tabulky stránek, kde první úroveň obsahuje ukazatele na tabulky druhé úrovně.
          3. **Velikost tabulek stránek**: S menším adresním prostorem jsou tabulky stránek menší a jednodušší na správu.
          4. **Výkon**: Menší adresní prostor může vést k rychlejšímu překladu adres, ale omezuje množství paměti, kterou může systém adresovat.
       2. **64bitová virtuální paměť**
          1. **Adresní prostor**: 64bitové systémy mají teoretický maximální adresní prostor 16 exabajtů (2^64 bajtů), což je mnohem více než současné potřeby.
          2. **Tabulky stránek**: Používají víceúrovňové tabulky stránek, často až čtyřúrovňové, aby efektivně spravovaly obrovský adresní prostor.
          3. **Velikost tabulek stránek**: S větším adresním prostorem jsou tabulky stránek mnohem větší, což může zvýšit režii správy paměti.
          4. **Výkon**: Větší adresní prostor umožňuje adresovat mnohem více paměti, což je výhodné pro aplikace vyžadující velké množství paměti, ale může také zvýšit složitost a čas potřebný pro překlad adres.
       3. **Shrnutí**
          1. Virtuální paměť je klíčová technologie umožňující efektivní využití fyzické paměti. Stránkování a tabulky stránek jsou základními mechanismy pro správu virtuální paměti. Rozdíly mezi 32bitovými a 64bitovými systémy se projevují hlavně ve velikosti adresního prostoru a složitosti tabulek stránek. 64bitové systémy nabízejí mnohem větší adresní prostor, což umožňuje adresovat více paměti, ale také zvyšuje složitost správy paměti.
  + **4.4.x Výměna stránek.**
    1. **4.4 Algoritmy výměny stránek**
       1. Když dojde k výpadku stránky, operační systém musí vybrat stránku k vyjmutí z paměti, aby vytvořil prostor pro stránku, která musí být vložena. Pokud byla stránka během pobytu v paměti upravena, musí být přepsána na disku, aby byla disková kopie aktuální. Pokud stránka nebyla pozměněna (například stránka obsahuje kód programu), disková kopie je aktuální a žádný přepis není nutný. Načítaná stránka pouze přepisuje stránku odkládanou.
       2. Přestože je možné vybrat náhodnou stránku k náhradě při každém výpadku stránky, výkon systému je lepší, pokud je vybrána stránka nepříliš používaná. Pokud je vyjmuta hodně používaná stránka, bude pravděpodobně potřeba ji rychle vrátit zpět, což způsobí vysokou režii. Bylo vyvinuto mnoho algoritmů náhrady stránky, teoretických i experimentálních. Níže jsou popsány některé z nejdůležitějších.
    2. **4.4.1 Optimální algoritmus náhrady stránky**
       1. Optimální algoritmus náhrady stránky je snadné popsat, ale nemožné implementovat. Funguje tak, že při výpadku stránky je vyjmuta stránka, která nebude použita nejdéle. Tento algoritmus je teoreticky nejlepší, protože minimalizuje počet výpadků stránek, ale v praxi je nerealizovatelný, protože operační systém nemůže předpovědět budoucí přístupy k paměti.
    3. **4.4.2 Náhrada dříve nepoužité stránky (Not Recently Used - NRU)**
       1. NRU algoritmus využívá dva bity: R (referenced) a M (modified). Stránky jsou rozděleny do čtyř kategorií:
          1. **Třída 0**: neodkazovány, nepozměněny
          2. **Třída 1**: neodkazovány, pozměněny
          3. **Třída 2**: odkazovány, nepozměněny
          4. **Třída 3**: odkazovány, pozměněny
       2. NRU algoritmus vyjímá stránku náhodně z neprázdné třídy s nejnižším číslem. Tento algoritmus je jednoduchý a efektivní, ale ne vždy optimální.
    4. **4.4.3 FIFO (First-In, First-Out) algoritmus náhrady stránky**
       1. FIFO algoritmus udržuje seznam všech stránek v paměti v pořadí, v jakém byly načteny. Při výpadku stránky je vyjmuta nejstarší stránka. Tento algoritmus je jednoduchý, ale může vyjmout často používané stránky, což snižuje výkon.
    5. **4.4.4 Náhrada stránky s druhou šancí (Second Chance)**
       1. Tento algoritmus je modifikací FIFO. Při výpadku stránky se kontroluje R bit nejstarší stránky. Pokud je 0, stránka je vyjmuta. Pokud je 1, R bit je vymazán a stránka je přesunuta na konec seznamu. Tento algoritmus zlepšuje výkon tím, že nevyjímá často používané stránky.
    6. **4.4.5 Hodinový algoritmus náhrady stránky (Clock)**
       1. Hodinový algoritmus je implementační varianta algoritmu druhé šance. Stránky jsou uspořádány do kruhového seznamu a ručička ukazuje na nejstarší stránku. Při výpadku stránky se kontroluje R bit stránky, na kterou ukazuje ručička. Pokud je 0, stránka je vyjmuta. Pokud je 1, R bit je vymazán a ručička se posune na další stránku. Tento algoritmus je efektivní a snadno implementovatelný.
    7. **4.4.6 Náhrada nejdéle nepoužívané stránky (Least Recently Used - LRU)**
       1. LRU algoritmus vyjímá stránku, která nebyla nejdéle používána. Tento algoritmus je blízký optimálnímu, ale je obtížné jej implementovat, protože vyžaduje udržování seznamu všech stránek v paměti a jejich aktualizaci při každém přístupu.
    8. **4.4.7 Programová simulace LRU**
       1. Jednou z možností simulace LRU je algoritmus stárnutí (aging). Každá stránka má čítač, který se při každém hodinovém přerušení posune doprava a přičte se R bit. Stránka s nejnižším čítačem je vyjmuta. Tento algoritmus je jednodušší než LRU, ale poskytuje podobné výsledky.
    9. **Shrnutí**
       1. Algoritmy výměny stránek jsou klíčové pro efektivní správu paměti. Každý algoritmus má své výhody a nevýhody, a jejich výběr závisí na konkrétních požadavcích a charakteristikách systému. Optimální algoritmus je teoreticky nejlepší, ale nerealizovatelný. Praktické algoritmy jako NRU, FIFO, Second Chance, Clock a LRU poskytují různé kompromisy mezi jednoduchostí a výkonem.
  + **4.6 Segmentace, účel, návaznost na virtuální paměť.**
    1. **4.6 Segmentace**
       1. Segmentace je technika správy paměti, která umožňuje rozdělit programy a data do logických segmentů. Každý segment představuje samostatný adresový prostor, což usnadňuje správu paměti a zvyšuje flexibilitu při práci s datovými strukturami.
       2. **Účel segmentace**
          1. **Logické členění**: Segmentace umožňuje rozdělit programy a data do logických částí, jako jsou kód, data, zásobník, tabulky symbolů, konstanty a další. Každý segment může růst a zmenšovat se nezávisle na ostatních, což usnadňuje správu paměti.
          2. **Ochrana paměti**: Každý segment může mít specifická přístupová práva (např. pouze pro čtení, zápis nebo vykonávání), což zvyšuje bezpečnost a stabilitu systému.
          3. **Sdílení paměti**: Segmentace umožňuje snadné sdílení kódu a dat mezi více procesy. Například knihovny mohou být sdíleny mezi různými programy, což šetří paměť a zvyšuje efektivitu.
       3. **Návaznost na virtuální paměť**
       4. Segmentace je často kombinována s virtuální pamětí, aby se dosáhlo ještě větší flexibility a efektivity. Virtuální paměť umožňuje, aby celková velikost programů a dat překročila velikost fyzické paměti tím, že se části programů a dat dynamicky načítají z disku do paměti podle potřeby.
          1. **Segmentace a stránkování**: V některých systémech, jako je Intel Pentium, se segmentace kombinuje se stránkováním. Virtuální adresy jsou rozděleny na segmenty a každý segment je dále rozdělen na stránky. Tím se dosahuje výhod obou technik – logického členění a ochrany segmentace a efektivního využití paměti stránkováním.
          2. **Správa paměti**: Segmentace umožňuje operačnímu systému efektivně spravovat paměť tím, že sleduje, které segmenty jsou aktivní a které mohou být odloženy na disk. To minimalizuje množství paměti potřebné pro běh programů a zvyšuje celkový výkon systému.
    2. **Implementace segmentace**
       1. **Čistá segmentace**
          1. Implementace segmentace se od stránkování liší již v základu: zatímco stránky mají pevnou velikost, segmenty nikoliv. Například fyzická paměť může obsahovat několik segmentů různé velikosti. Když se segment odstraní a nahradí menším segmentem, vznikají mezery (fragmentace). Tento jev, zvaný šachovnicový (checkerboarding) nebo externí fragmentace, plýtvá pamětí na mezery. Řešením je komprimace paměti, která spojuje mezery do jedné velké oblasti.
       2. **Segmentace se stránkováním**
          1. V některých systémech, jako je Intel Pentium, se segmentace kombinuje se stránkováním. Pentium má 16 000 nezávislých segmentů, přičemž každý může mít až 1 miliardu 32bitových slov. Každý program má vlastní tabulku lokálních deskriptorů (LDT) a globální tabulku deskriptorů (GDT), kterou sdílejí všechny programy. LDT popisuje lokální segmenty každého programu, zatímco GDT popisuje systémové segmenty.
          2. Při přístupu k segmentu program načte selektor do segmentového registru procesoru. Selektor určuje, zda se jedná o segment lokální nebo globální, a obsahuje číslo položky v LDT nebo GDT. Deskriptor segmentu obsahuje základní adresu segmentu, velikost a další informace. Pokud je stránkování zapnuto, lineární adresa je namapována pomocí tabulek stránek na fyzickou adresu.
    3. **Výhody segmentace**
       1. **Flexibilita**: Segmentace umožňuje růst a zmenšování segmentů nezávisle na ostatních, což usnadňuje správu paměti.
       2. **Ochrana**: Každý segment může mít specifická přístupová práva, což zvyšuje bezpečnost systému.
       3. **Sdílení**: Segmentace umožňuje snadné sdílení kódu a dat mezi procesy, což šetří paměť a zvyšuje efektivitu.
       4. **Kompatibilita**: Systémy jako Intel Pentium umožňují kombinaci segmentace a stránkování, což poskytuje výhody obou technik.