Sisteme de Operare

- Note de curs -

Anton Puiu
Grupa 332
Seria CA
Facultatea de Automatică şi Calculatoare
Universitatea Politehnica Bucureşti
anton.puiu@email.com

Cuprins

1 Curs 1

Obiective

- Definirea noţiunii de **sistem de operare**, împreună cu legătura pe care acesta o are relativ la *sistemul de calcul*.
- Prezentarea anumitor **concepte importante în sisteme de operare**.
- Definirea **structurii unui sistem de operare**.

1.1 Sistem de calcul

Un sistem de calcul poate fi reprezentat printr-o stivă de nivele. Astfel, reprezentarea generală a astfel de sisteme este următoarea:

- **Dispozitive fizice** - acest nivel este definit de componentele din care este format un sistem de calcul.
- **Microarhitectură** - acest nivel reprezintă modulele formate utilizînd dispozitivele fizice din nivelul anterior.
- **Limbajul maşină** - la acest nivel este definită **interfaţa dintre hardware şi software**. Hardware-ul pune la dispoziţie un set de instrucţiuni care pot fi plasate în memorie şi pentru care acesta garantează execuţia corectă a acelor instrucţiuni. Totalitatea acestor instrucţiuni formează **limbajul maşină**.
- **Sistemul de operare** - reprezintă *cel mai apropiat nivel software* în contextul interacţiunii cu maşina fizică. Scopul acestui nivel este de a pune la dispoziţie aplicaţiilor de nivel superior un set de funcţii mai uşor de utilizat.
- **Aplicaţii utilizator** - orice program lansat în execuţie de către un utilizator cu scopul rezolvării unei probleme.

1.2 Sistem de operare

Un sistem de calcul pune la dispoziţia utilizatorului următoarele resurse: unul sau mai multe procesoare, memorii interne, memorii auxiliare, ceasuri, terminale, interfeţe de reţea, dispozitive de intrare/ieşire (imprimante, scannere etc). Dacă utilizatorul ar trebui să scrie el însuşi programele care gestionează şi utilizează corect aceste componente hardware ar rezulta programe foarte complicate. Utilizatorul ar trebui să facă programe ce conţin propriile rutine pentru crearea de blocuri pe disc, recuperarea din condiţii de eroare, tratarea unor întreruperi. Pentru a scuti utilizatorul de sarcina de a gestiona singur hardware-ul complex s-a adoptat soluţia de a plasa un strat de programe specializate deasupra maşinii fizice. Aceste programe trebuie să administreze toate resursele sistemului.

Prin **sistem de operare** se înţelege o colecţie organizată de programe pentru control şi serviciu, rezidente permanent într-o memorie principală sau auxiliară, specifice tipurilor de echipamente din componenţa sistemului, avînd ca sarcină optimizarea utilizării resurselor, minimizarea efortului uman de programare şi automatizarea operaţiilor manuale, în cît mai mare măsură, în toate fazele de pregătire şi exploatare a echipamentelor.

Resursele sistemului
Resursele pot fi **permenente** (procesoare, memorii, dispozitive i/e) sau **temporare** (mesaje) acestea din urmă sunt produse de un proces şi consumate de un altul. Resursele pot fi **dedicate** (folosesc numai procesului care le deţine) sau **partajabile**.

În ceea ce priveşte modul de recuperare a resurselor de la procesul care le deţine acestea pot fi cu **prelevare forţată** - de exemplu procesorul, memoria - (pot fi luate în orice moment de la procesul care le deţine) şi fără prelevare forţată - **neprelevabile** - (resursele pot fi recuperate numai cînd procesul care le stăpîneşte le cedează).

Sistemul de operare are ca scop să pună la dispoziţia utilizatorului o maşină extinsă care este mai uşor de programat decît hardware-ul. El pune la dispoziţia programatorului aşa numitele **apeluri de sistem** care pot fi considerate instrucţiuni ale maşinii extinse.

Programele adăugate maşinii fizice în vederea administrării resurselor sale sunt cunoscute sub numele de **Sisteme de Operare**. Acesta cuprinde acele proceduri manuale şi module program din interiorul unui sistem de calcul care asigură controlul sistemului de calcul (procesor, memorie internă, memorii externe, dispozitive I/E, fişiere) de către utilizatori, rezolvă conflictele de acces la resurse, încearcă să îmbunătăţească performanţele sistemului şi simplifică utilizarea acestuia. S.O. acţionează ca o interfaţă între programele utilizator şi hardware.

1.3 Organizarea sistemelor de operare

Pentru a putea lansa în execuţie un program, sunt necesare diferite componente software care să pregătească respectivul program pentru rulare. Astfel de componente software sunt în general parte din **sistemul de operare**. De asemenea, pentru a realiza *compilarea şi editarea surselor programelor*, sunt necesare programe precum compilatoare şi editoare text, acestea fiind incluse în **sistemul de operare**. Întrucît se observă două tipuri de **programe** oferite de **sistemul de operare**, organizarea acestuia se realizează după cum urmează:

- **Programe de control**, destinate a lucra în starea de lucru supervizor, avînd deci ca resurse întregul set de instrucţiuni (instrucţiunile normale şi instrucţiunile privilegiate).
- **Programe de serviciu**, destinate a lucra în starea de lucru program, pentru care instrucţiunile privilegiate sunt **interzise**.

1.3.1 Programe de control

Programele de control sunt organizate într-o ierarhie cu două niveluri:

- *Nivelul fizic*
- *Nivelul logic*

Nivelul fizic
Programele care fac parte din acest nivel primesc controlul prin intermediul sistemului de întreruperi. Aceste programe exercită controlul asupra funcţionării dispozitivelor şi au în componenţă următoarele categorii de rutine:

- Rutine care asigură intrarea şi ieşirea din sistemul de operare, adică trecerea controlului de la programul utilizatorului la sistemul de operare şi invers;

- Rutine pentru satisfacerea întreruperilor, adică cercetarea cauzelor care au generat comutarea de stare şi efectuarea acţiunilor corespunzătoare, în vederea reluării execuţiei programului întrerupt;
- Rutine pentru îndeplinirea de servicii ce necesită utilizarea unor instrucţiuni privilegiate, cunoscute sub numele de **servicii ale supervizorului**, care sunt accesibile programelor prin forţarea apariţiei unei întreruperi din clasa **chemare supervizor**.

Nivelul logic
Programele care fac parte din acest nivel sunt răspunzătoare de luarea deciziilor privind utilizarea resurselor sistemului şi au în componenţă următoarele categorii de rutine:

- Rutine pentru administrarea resurselor sistemului, care asigură alocarea acestora la lucrările în curs de execuţie;
- Rutine pentru planificarea lucrărilor la execuţie (se poate organiza planificarea pe mai multe niveluri: planificarea de lucrări, planificarea de sarcini, planificarea de timp, etc);
- Rutine pentru comunicarea şi interacţiunea cu operatorul sistemului.

Datorită apelării frecvente a rutinelor ce intră în componenţa sistemelor de operare, este important ca executarea repetată a acestora să se desfăşoare în condiţii de funcţionare asincronă. Este necesar pentru aceasta să se poată comuta controlul dintr-un punct oarecare al unei rutine de sistem, întreruptă de un proces prioritar, în alt punct al aceleiaşi rutine, fără a avea vreo relaţie deterministă între poziţiile celor două puncte şi păstrîndu-se posibilitatea de a se relua prelucrarea de la punctul anterior de întrerupere.

Se impune deci ca prin executarea unei rutine, textul acesteia să nu se modifice, astfel încît să fie posibilă rularea ulterioară a aceleiaşi rutine fără a o reîncărca în memoria principală, prin recopiere din biblioteca pe care rezidă sistemul de operare. Din acest motiv, majoritatea rutinelor sistemului de operare sunt **reentrante** (instrucţiunile care se referă la operanzi în memoria principală, efectuează în aceştia numai operaţii care implică citire din memorie) sau **reutilizabile în serie** (rutinele îşi autoiniţiază valorile din text care suferă modificări în timpul unei chemări anterioare la execuţie).

Volumul mare de memorie necesar pentru a păstra toate rutinele sistemului de operare, nu poate fi asigurat decît într-o memorie peeriferică, utilizînd un volum cu capacitate mare de memorare. Acele rutine care sunt apelate mai frecvent, sau care au un aport mai mare în determinarea consumului de timp, sunt grupate într-un **nucleu** care rezidă permanent în memoria principală. Celelalte rutine sunt chemate la execuţie prin încărcare într-o zonă a nucleului numită **arie de tranziţie**, situată în afara rutinelor permanente. Pentru a nu perturba funcţionarea sistemului, rutinele de tip fizic sunt apelate într-o arie de tranziţie fizică, iar rutinele de tip logic sunt apelate într-o arie de tranziţie logică. În acest fel, evident prin mărirea timpului de supraveghere, se poate opera cu sistemul de operare utilizînd numai o fracţiune din memoria principală a calculatoarelor respective.

1.3.2 Programe de serviciu

Aceste programe sunt destinate realizării acelor funcţiuni de asistenţă care nu condiţionează desfăşurarea lucrărilor în curs de execuţie, sunt apelate din programele de control şi încărcate în partiţiile de memorie destinate programelor utilizatorilor. Unele limbaje de control prevăd enunţuri speciale pentru apelarea programelor de serviciu, altele folosesc aceleaşi enunţuri imperative care determină executarea oricărui program catalogat în biblioteca sistemului. Sunt considerate drept componente ale sistemelor de operare, făcînd parte din categoria programelor de serviciu, următoarele tipuri de programe:

- compilatoare, destinate obţinerii de programe în codul intern al calculatoarelor, avînd ca sursă programele elaborate în limbajele de programare simbolice;
- programe pentru structurare, editare de legături între programe şi pregătire a diferitelor forme de programe executabile;
- programe pentru crearea, organizarea şi întreţinerea de biblioteci pentru programele sistemului şi ale utilizatorului;
- programe pentru servicii uzuale: randomizări, sortări, interclasări, transferări de date de la un periferic la altul, iniţializări de volume, reorganizări de colecţii de date, etc.

Pentru a proteja sistemul de operare contra alterării accidentale datorită unor operaţii de scriere iniţiate din programele utilizatorilor, nucleul sistemului şi partiţiile de memorie destinate programelor utilizatorilor sunt protejate prin chei de protecţie distincte. Măsuri speciale se iau şi în ce priveşte protejarea volumului pe care rezidă sistemul şi bibliotecile sale.

Sisteme de calcul

Orice aplicaţie scrisă ce are ca scop rularea utilizînd resursele unui sistem de calcul, este necesar să se cunoască detalii legate de aceste resurse. Aceste detalii trebuie în schimb mascată pentru dezvoltatorii de aplicaţii, întrucît aceste detalii sunt dependente de maşina fizică, iar o aplicaţie nu are ca scop definirea utilizării resurselor sistemului pe care rulează, acestea existînd **în maşina de calcul**, la un alt nivel. Aplicaţia are la dispoziţie un set de funcţii pe care le poate executa, astfel încît să utilizeze resursele sistemului. Totalitatea acestor funcţii reprezintă **sistemul de operare**.

Sistemele de operare au fost create nu doar pentru a **organiza** funcţiile menţionate anterior, dar în acelaşi timp să utilizeze resursele într-un mod **eficient** şi de asemenea, **securitatea** este un punct esenţial.

Folosirea eficientă a resurselor

Cu scopul creşterii eficienţei utilizării sistemelor de calcul, se încearcă **distribuirea resurselor** acelor *procese* care necesită respectivele resurse. Astfel, în defavoarea lansării secvenţiale a respectivelor procese, care utilizează resurse diferite, prin intermediul **sistemului de operare**, în timpul în care un **proces** utilizează o resursă, iar un alt proces necesită o altă resursă care este liberă, respectivul proces va utiliza resursa liberă, astfel crescînd eficienţa sistemului de calcul.

Securitatea sistemelor de operare

Avînd în vedere rularea *în acelaşi timp* a mai multor procese, trebuie asigurată **iluzia** faptului că o aplicaţie are *control complet asupra sistemului*. De asemenea, accesul la **anumite resurse** la un **anumit moment** ar trebui sa fie restricţionat, întrucît, spre exemplu, dacă două procese încearcă să comunice cu un dispozitiv periferic, acestea vor încerca scrierea sau citirea la aceste dispozitive, dar pentru că un proces are **iluzia** că are *control complet asupra sistemului*, aceasta **nu are cum să afle de existenţa unei alte aplicaţii care doreşte utilizarea aceleiaşi resurse în acelaşi timp**. Pentru a realiza **arbitrarea** accesului la respectiva resursă, **sistemul de operare** poate fi *apelat* prin intermediul unui **apel de sistem**, acesta realizînd arbitrarea.

2 Curs 2 - Sistemul de fişiere

Obiective

- Expunerea scopului sistemelor de fişiere.
- Descrierea interfeţelor sistemelor de fişiere.
- Discutarea aspectelor de proiectare a sistemelor de fişiere, compromisurilor făcute în proiectarea acestora, metodele de acces şi de distribuire a fişierelor, blocarea fişierelor şi structura directoarelor.
- Explorarea protecţiei sistemelor de fişiere.

Avînd în vedere faptul că totalitatea datelor stocate într-un sistem de calcul formează o bază de date, acea bază de date, în cazul sistemelor de operare actuale se utilizează **modelul ierarhic**, în care datele sunt organizate sub forma unui arbore, nodurile constînd din înregistrări, iar arcele referiri către alte noduri.

Sistemul de fişiere este reprezentat de o colecţie de funcţii care au ca scop implementarea unei astfel de baze de date, în care primitiva, din punctul de vedere al sistemului de operare, este reprezentată de **fişiere**. La nivelul dispozitivelor fizice, se utilizează primitiva de **bloc**. Toată logica dintre crearea unui fişier, din punctul de vedere al sistemului de operare, pînă la stocarea acestuia pe un dispozitiv periferic sau în memoria secundară, reprezintă **sistemul de fişiere**.

2.1 Conceptul de fişier

Un fişier reprezintă o colecţie de informaţii stocate într-o memorie secundară. Fişierele pot reprezenta programe sau date. Datele pot fi stocate în format text sau binar. În general, un fişier este reprezentat de o secvenţă de biţi, iar interpretarea conţinutului este definită de creatorul acestuia.

Avînd în vedere gradul de generalitate pe care un fişier îl oferă, practic printr-un fişier se pot stoca o serie de tipuri de informaţii: programe sursă, imagini, text, etc. În funcţie de tipul de informaţie stocat într-un fişier, informaţiile sunt reţinute într-o anumită structură, astfel încît acestea să poată fi interpretate de către utilizator prin intermediul unui program.

2.1.1 Metadate

Metadatele unui fişier reprezintă informaţii necesare sistemului de operare, pentru a putea pune la dispoziţie beneficii precum securitate, identificare şi eficienţă. Metadatele fişierelor variază de la un sistem de operare la altul, dar în general sunt utilizate următoarele:

- **Nume**: necesar utilizatorilor pentru a identifica un anume fişier.
- **Identificator**: reprezintă o valoare unică în sistemul de fişiere.
- **Tip**.
- **Dimensiune**: dimensiunea curentă a fişierului şi posibil dimensiunea maximă.
- **Drepturi de acces**: necesare în vederea securităţii vizualizării, scrierii şi executării fişierelor.
- **Timp, dată şi utilizatori**: informaţii legate de crearea, modificare şi accesul fişierelor.

Toate metadatele fişierelor sunt stocate în directorul din care fac parte. Practic, un director este un caz particular de fişier, care drept date conţine identificatorul fişierelor pe care îl conţine împreună cu metadatele acestora. Astfel, dimensiunea unui fişier este dată de suma tuturor metadatelor fişierelor pe care le conţine, împreună cu dimensiunea identificatoarelor acestora.

2.1.2 Operaţii pe fişiere

Un fişier reprezintă un **tip abstract de date**. Pentru a defini corect un fişier, trebuie stabilite operaţiile care se pot efectua asupra acestora, acestea fiind implementate în sistemul de operare prin intermediul **apelurilor de sistem**. Aceste operaţii sunt:

- **Crearea**: doi paşi sunt necesari pentru a crea un fişier. În primul rînd, trebuie să existe spaţiu în sistemul de fişiere pentru acesta. În al doilea rînd, trebuie creată o intrare pentru noul fişier în cadrul directorului din care face parte.

- **Scrierea**: pentru a scrie un fişier se efectuează un apel de sistem specificînd numele fişierului şi informaţiile ce trebuie scrise. Fiind dat numele fişierului, sistemul de fişiere determină locaţia fişierului în directorul curent. Sistemul de fişiere trebuie să reţină un pointer către locaţia în care o scriere ulterioară va avea loc, astfel pointer-ul trebuie mutat ori de cîte ori se efectuează o scriere.

- **Citirea**: pentru a citi dintr-un fişier se utilizează un apel de sistem ce specifică numele fişierului şi locaţia de memorie unde se vor stoca datele citite. Similar cu scrierea, se va efectua o căutare în directorul curent pentru fişierul specificat şi se reţine un pointer care indică ultimul octet citit. Întrucît un proces poate citi şi scrie într-un fişier, trebuie reţinute ultimele locaţii unde au fost efectuate operaţiile, dar un singur pointer este necesar.

- **Poziţionarea cursorului**: se efectuează o căutare în directorul curent, iar pointer-ul asociat fişierului este repoziţionat la locaţia specificată. Această operaţie nu necesită interacţiune cu memoria secundară.

- **Ştergerea**: pentru a şterge un fişier, se caută directorul curent pentru fişierul specificat. După identificarea fişierului, se eliberează din sistemul de fişiere spaţiul alocat acestuia şi se şterge intrarea şi metadatele acestuia din directorul curent.

- **Truncherea**: în cazul în care se doreşte păstrarea atributelor unui fişier, se poate şterge conţinutul acestuia prin trunchere.

Aceste şase operaţii de bază reprezintă un set minimal de acţiuni asupra fişierelor. Alte operaţii comune includ redenumirea unui fişier sau scrierea la finalul acestuia, în cazul acestora fiind utilizate primitivele menţionate anterior.

Pentru a înlătura căutarea efectuată în cazul primitivelor, se preferă utilizarea apelului de sistem **open**, care efectuează căutarea *o singură dată* înainte de utilizarea fişierului, acest apel de sistem oferind un **descriptor de fişier** care este folosit ulterior pentru orice operaţie asupra fişierului respectiv. Sistemul de operare păstrează o tabelă conţinînd fişierele deschise de către un proces. Descriptorii de fişier reprezintă indecşi în respectiva tabelă, iar astfel nu mai este necesară căutarea. Atunci cînd nu se mai efectuează operaţii asupra unui fişier, acesta este închis, iar intrarea acestuia în tabelă este ştearsă.

Apelul de sistem **open** acceptă de asemenea specificarea informaţiilor de acces. Aceste informaţii trebuie să fie compatibile cu permisiunile fişierului, sau în cazul creării unui nou fişier, cu permisiunile directorului curent.

Implementarea apelurilor de sistem **open** şi **close** devine complicată în momentul în care mai multe procese ar putea deschide un fişier în acelaşi timp. În general, sistemul de operare utilizează două tabele interne: o tabelă în care se reţin **toate** fişierele deschise în sistem şi o tabelă pentru fiecare proces.

Fiecare intrare în tabela unui proces reprezintă o imagine a tabelei sistemului. Astfel, tabela procesului conţine doar legături către tabela sistemului.

În general, următoarele informaţii sunt asociate cu un fişier deschis:

- **Pointer fişier**: în sistemele de operare ce nu includ deplasamentul faţă de începutul fişierului ca parte al apelurilor de sistem **read** şi **write**, sistemul trebuie să urmărească locaţiile în care s-a efectuat ultima citire, respectiv ultima scriere. Aceste locaţii sunt unice în cazul fiecărui proces în parte, pentru fiecare fişier deschis, iar astfel acestea trebuie reţinute separat de atributele fişierului.
- **Numărul de fişiere deschise**: pe măsură ce procesele închid fişierele deschise, sistemul de operare trebuie să reutilizeze intrările tabelei globale de fişiere. Întrucît orice proces poate deschide un fişier, sistemul de operare trebuie să aştepte pînă în momentul în care toate procesele au închis un anumit fişier înainte ca acesta să fie eliminat din tabela globală de fişiere.
- **Locaţia unui fişier pe disc**: Majoritatea operaţiilor pe fişiere presupun modificarea datelor dintr-un fişier. Astfel, informaţia necesară pentru a localiza un fişier pe disc este reţinută în memorie astfel încît să nu fie necesară o căutare pentru fiecare operaţie.
- **Drepturi de acces**: Fiecare proces deschide un fişier într-un mod de acces. Această informaţie este stocată pentru fiecare proces în parte astfel încît sistemul de operare să poată gestiona accesul fiecărui proces.

Anumite sisteme de operare pun la dispoziţie operaţia de blocare a unui fişier, iar astfel fişierul respectiv poate fi accesat doar de procesul care a deschis respectivul fişier. Această facilitate este utilă pentru fişiere partajate, în cazul în care mai multe procese doresc să scrie într-un astfel de fişier.

2.1.3 Structura internă a fişierelor

Identificarea unui deplasament în cadrul unui fişier este o sarcină destul de grea pentru un sistem de operare. Aşa cum a fost menţionat anterior, informaţiile se reţin în memoria secundară la nivel de **bloc**. În cazul hard-disk-urilor, un **bloc** reprezintă un grup de **sectoare** pe care sistemul de operare le poate adresa. Un sector, în cazul hard-disk-urilor are dimensiunea de 512 octeţi. Motivul pentru care sistemul de operare lucrează cu blocuri este dat de limita de adresare a blocurilor. Pentru ca un sistem de operare să poată adresa dispozitive de stocare de dimensiune mare, sectoarele sunt grupate în blocuri.

Versiunile curente de sisteme de operare verifică dimensiunea hard-disk-ului şi determină numărul minim de sectoare necesar pentru a forma un bloc pentru a utiliza întregul spaţiu de stocare. În cazul în care un bloc reprezintă 512 octeţi, atunci pentru un fişier de 1949 octeţi ar fi alocate patru blocuri, iar ultimii 99 octeţi ar fi risipiţi. Această risipă este datorată utilizării blocurilor şi se numeşte **fragmentare internă**. Toate sistemele de fişiere suferă de **fragmentare internă**. Cu cît dimensiunea blocului creşte, cu atat este mai mare fragmentarea internă.

2.2 Metode de acces

Fişierele reţin informaţie, iar atunci cînd sunt utilizate, această informaţie trebuie accesată şi reţinută în memoria principală a calculatorului. Informaţia din fişiere poate fi accesată în diferite moduri. Anumite sisteme pun la dispoziţie mai multe metode de acces, iar alegerea metodei potrivite pentru o aplicaţie particulară reprezintă o problemă de proiectare majoră.

2.2.1 Acces secvenţial

Această metodă de acces este cea mai simplă. Informaţia din fişier este procesată în ordine. Acest mod de acces este cel mai comun.

Scrierile şi citirile se realizează cum a fost menţionat anterior. Aceste fişiere pot fi trunchiate, iar în anumite sisteme, aplicaţiile pot scrie la deplasamente arbitrare faţă de începutul fişierului.

2.2.2 Acces direct

O altă metodă de acces este acces-ul direct. Un fişier este de dimensiune logică fixă, iar aplicaţiile pot citi şi scrie informaţii rapid în orice ordine. Pentru această metodă, fişierul este văzut ca o secvenţă numerotată de blocuri. Astfel, este posibilă citirea arbitrară a blocurilor.

Această metodă de acces este utilă în cazul în care este necesar accesul unui volum mare de informaţii. Bazele de date utilizează în general acest tip de acces. Atunci cînd o cerere este efectuată, este determinat blocul în care se regăseşte răspunsul cererii, urmînd citirea acestuia direct pentru a pune la dispoziţie informaţia căutată.

În cazul acestei metode de acces, apelurile de sistem **read** şi **write** trebuie să includă numărul blocului drept parametru. Astfel, în defavoarea citirii/scrierii a n ulteriori octeţi, se va citi/scrie blocul n. De asemenea, este necesar un apel de sistem pentru poziţionarea cursorului la blocul n.

Numărul blocului specificat în apelul de sistem este, în mod normal, **număr de bloc relativ**. Un astfel de număr reprezintă un index relativ la începutul fişierului. Numerele relative încep de la zero, indiferent de locaţia lor pe disc. Utilizarea acestor numere permite sistemului de operare să decidă unde va fi stocat fişierul respectiv.

2.2.3 Alte metode de acces

Se pot crea diferite metode peste metoda de acces directă. Aceste metode în general implică construirea unui index pentru fişiere, în care se vor stoca pointeri către anumite blocuri. Pentru a căuta înregistrările unui fişier, se vor căuta în index pointer-ul asociat acestuia, urmînd să fie efectuată căutarea pentru înregistrarea respectivă.

În momentul în care dimensiunea fişierelor creşte, index-ul poate deveni prea mare pentru a fi reţinut în memorie. O soluţie este de a crea un index pentru index. Primul index conţine pointeri pentru al doilea index, care ar reţine datele efective.

2.3 Structura directoarelor pe disc

2.3.1 Structura de stocare

2.3.2 Tipuri de directoare

2.4 Montarea sistemelor de fişiere

2.5 Protecţia

2.5.1 Tipuri de acces

2.5.2 Controlul accesului

2.5.3 Alte abordări ale protecţiei

2.6 Structura sistemului de fişiere

Sistemele de fişiere au două caracteristici pentru care reprezintă un mediu convenabil pentru stocarea fişierelor:

- Un disc poate fi rescris.
- Un disc poate accesa direct orice bloc de informaţie. Astfel, accesarea oricărui fişier reprezintă o operaţie simplă, iar citirea a două fişiere arbitrare necesită doar mutarea capului de citire, împreună cu timpul de aşteptare necesar discului să efectueze o rotaţie.

Pentru a spori eficienţa transferurilor de intrare/ieşire, datele dintre memoria principală şi secundară sunt transmise în blocuri. Fiecare bloc are unul sau mai multe sectoare.

Sistemele de fişiere pun la dispoziţie o metodă de acces convenabilă şi eficientă pentru stocarea şi accesarea datelor. Un sistem de fişiere rezolvă două probleme de proiectare. Prima problemă este definirea interfeţei cu utilizatorul. Această sarcină implică definirea unui fişier şi a atributelor sale, operaţiile permise asupra fişierelor, împreună cu structura directoarelor pentru organizarea fişierelor. A doua problemă este reprezentată de definirea algoritmilor şi structurilor de date pentru a mapa sistemul de fişiere logic pe dispozitivele fizice.

Sistemul de fişiere este, în general, compus din mai multe nivele:

- Control I/O
- Sistem de fişiere de bază
- Modulul de organizare a fişierelor
- Sistemul logic de fişiere

Fiecare nivel utilizează funcţii ale nivelelor anterioare pentru a pune la dispoziţie noi funcţionalităţi utile nivelelor superioare.

Primul nivel, *Control-ul I/O*, este reprezentat de **driver-ul de dispozitiv** şi handler-e de întrerupere pentru a transfera informaţia de la memoria principală la dispozitivul din memoria secundară. Un astfel de driver poate fi privit drept translator. Intrarea sa este reprezentată de comenzi de nivel înalt cum ar fi citirea unui bloc. Ieşirea acestuia conţine instrucţiuni specifice dispozitivului pe care îl controlează, trimise către controller-ul dispozitivului pentru a realiza respectiva operaţie.

Sistemul de fişiere de bază trebuie doar să transmită comenzi generice driver-ului de dispozitiv pentru a citi sau scrie blocuri fizice pe disc. Acest nivel este responsabil de zonele tampon ce reţin diferite sisteme

de fişiere, directoare şi blocuri de date. Un bloc de din zona tampon este alocat înainte de a fi iniţiat transferul către disc. Atunci cînd zona tampon este plină, aceasta trebuie eliberată pentru a permite completarea operaţiei de intrare/ieşire.

Modulul de organizare a fişierelor este punctul în care se cunosc blocurile logice şi blocurile fizice. La acest nivel se efectuează translatarea dintre cele două tipuri de blocuri. Tot aici se efectuează eliberarea spaţiului liber prin urmărirea blocurilor nealocate.

Sistemul logic de fişiere administrează metadatele. Metadatele includ toate structurile sistemului de fişiere, mai puţin datele efective. De asemenea, la acest nivel se administrează structura directoarelor pentru a pune la dispoziţie nivelului inferior numele fişierelor. Un **bloc de control al fişierelor** (FCB, inode) conţine informaţii legate de fişier, cum ar fi deţinător, permisiuni şi locaţia datelor. La acest nivel se realizează protecţia şi securitatea.

3 Curs 3 - Procese

Obiective

- Introducerea noţiunii de proces.
- Descrierea proprietăţilor proceselor, cum ar fi planificarea, crearea şi terminarea.

Unul din cele mai importante concepte ale sistemului de operare este **procesul**: abstractizarea unui program în execuţie. În funcţie de modul de utilizare a resurselor sistemului, există următoarele tipuri de sisteme de operare:

- Sistemele în care resursele necesare proceselor sunt alocate acestora pe toată durata execuţiei, sunt cunoscute sub denumirea de **sisteme cu resurse alocate**. În cazul resurseor unice (procesor, memorie) se folosesc metode de utilizare în serie şi de partiţionare, care să permită menţinerea simultană în sistem a mai multor procese. Aceste sisteme execută **lucrări** (**jobs**).
- Sistemele în care resursele necesare lucrărilor sunt alocate acestora periodic pe durata unor cuante de timp, astfel încît fiecare proces dispune de resursele sistemului în intervalul de timp alocat, sunt cunoscute sub denumirea de **sisteme cu resurse distribuite**. Astfel, cînd resursa distribuită este **timpul unităţii centrale**, fiecărui proces i se atribuie consecutiv cîte o cuantă de timp controlul unităţii centrale. Aceste sisteme sunt cunoscute sub denumirea de **sisteme cu timp partajat** (time sharing). Cînd resursa distribuită este memoria principală, aceasta este divizată în partiţii egale denumite **pagini**, iar pe dispozitivele de memorare periferice se organizează spaţiul în partiţii de aceleaşi dimensiuni. Dacă sistemul de operare asigură trecerea automată a paginilor ddin memoria periferică în memoria principală pentru executarea de programe, sau invers pentru eliberarea de spaţiu operaţional, se consideră că sistemul dispune de o **memorie virtuală**, a cărei mărime este dată de suma capacităţilor de memorare ale memoriei principale şi dispozitivelor periferice de memorare. Aceste sisteme execută **programe utilizator** sau **sarcini** (**task-uri**).
- Sistemele care permit controlul executării lucrărilor în interiorul unui interval de timp specific denumit **timp de răspuns**, sunt cunoscute sub denumirea de **sisteme în timp real**.

3.1 Modelul de proces

În acest model, orice program dintr-un sistem de calcul este organizat într-un număr de **procese secvenţiale**. Diferenţa dintre un program şi un proces este că un program reprezintă o entitate **pasivă**, reprezentată de un fişier executabil stocat în memoria secundară, iar un proces o entitate **activă**, care poate exista doar după ce fişierul executabil este încărcat în memoria principală. Astfel, toate informaţiile necesare rulării unui program sunt încapsulate în conceptul de **proces**. Chiar dacă două procese pot fi asociate cu acelaşi program, acestea sunt considerate două unităţi de execuţie separate.

Un proces conţine următoarele informaţii:

- Secţiunea text, în care se regăseşte codul programului.
- Numărătorul de program şi conţinutul registrelor procesorului.
- Stiva procesului, unde sunt păstrate valorile temporare ale programului.
- Secţiunea de date, unde se regăsesc variabilele globale, împreună cu zona de memorie alocată dinamic, sau **heap**-ul.

3.1.1 Stările proceselor

Starea unui proces este definită de activitatea curentă a acestuia. Un proces poate avea una din următoarele stări:

- Nou: procesul a fost creat.
- În rulare: procesul execută instrucţiuni. *Un singur proces poate fi în această stare pentru un procesor.*
- Aşteptare: procesul aşteaptă un eveniment.
- Pregătit: procesul aşteaptă planificarea pe procesor.
- Terminat: procesul si-a încheiat execuţia.

3.1.2 Bloc de control al proceselor

Fiecare proces este reprezentat în sistemul de operare de un bloc de control (PCB). Un astfel de bloc reţine următoarele elemente:

- Starea procesului.
- Contorul de program.
- Registrele procesor.
- Informaţii legate de planificarea procesului, cum ar fi prioritate, pointeri către cozi de planificare, etc.
- Informaţii legate de administrarea memoriei, cum ar fi dimensiunea paginilor, adresele de memorie fizică alocate, etc.
- Informaţii de rulare, cum ar fi numărul de procese create, timpul petrecut pe procesor, etc.
- Informaţii de intrare/ieşire, cum ar fi dispozitivele periferice accesate de către proces, fişiere deschise, etc.

3.2 Operaţii pe procese

Procesele pot fi create şi şterse în mod dinamic. Astfel, sistemele cu timp partajat trebuie să pună la dispoziţie un mecanism pentru crearea şi terminarea proceselor.

3.2.1 Crearea proceselor

Există trei evenimente principale pentru care un proces poate fi creat:

- Iniţializarea sistemului: În momentul în care sistemul este pornit, se crează primele procese. Unele dintre acestea interacţionează cu utilizatorul, altele interacţionează între ele, având anumite funcţii speciale. De exemplu, un proces poate fi activ pentru a prelua email-uri sau pagini web. Astfel de procese se numesc **demoni**.

- Efectuarea unui apel de sistem ce are drept consecinţă crearea unui nou proces, de către un proces: Deseori un proces poate efectua apeluri de sistem pentru a crea noi procese cu scopul de a-şi îndeplini scopul. Un motiv pentru care un proces ar efectua un astfel de apel de sistem este pentru paralelizare. Pe un sistem multiprocesor, o astfel de tehnică va micşora timpul de execuţie.
- Solicitarea unui utilizator de a crea un nou proces: În sistemele de operare interactive, utilizatorii pot porni procese prin intermediul comenzilor.

În oricare dintre cazurile menţionate anterior, un nou proces este creat prin intermediul unui apel de sistem. După crearea procesului copil, părintele şi copilul au spaţii de adresă diferite, iar astfel cele două procese sunt separate.

Majoritatea sistemelor de operare identifică procesele printr-un identificator de proces (pid), care este în general un număr întreg. Acest identificator poate fi utilizat drept index pentru a accesa diferite atribute ale unui proces în momentul execuţiei unui apel de sistem.

În general, cînd un proces părinte crează un proces copil, copil-ul necesită anumite resurse (timp pe procesor, memorie, fişiere, dispozitive periferice) pentru aşi îndeplini sarcinile. Un astfel de proces ar putea obţine resursele necesare direct din partea sistemului de operare, sau ar putea fi constrîns la o submulţime a resurselor părintelui. Părintele ar fi obligat să împartă resursele sale cu procesele copil, sau să le împartă. Restricţionînd un proces copil la o submulţime a resurselor părintelui previne supraîncărcarea sistemului prin crearea prea multor procese copil.

În plus faţă de resursele fizice şi logice, părintele poate trimite date de iniţializare către procesul copil. Alternativ, sistemul de operare ar putea trimite aceste resurse către copil.

Atunci cînd un proces crează un nou proces, există două posibilităţi de execuţie:

- Procesul părinte îşi conţinuă execuţia.
- Procesul părinte aşteaptă toate procesele copil să termine execuţia.

Există de asemenea două spaţii de adresă posibile pentru procesele copil:

- Procesul copil este un duplicat al procesului părinte.
- Procesul copil are un nou program încarcat.

3.2.2 Terminarea proceselor

După ce un proces este creat, acesta începe să îşi execute sarcinile. Atunci cînd un proces îşi termină execuţia, acesta poate fi încheiat din următoarele motive:

- Ieşire normală (voluntar).
- Ieşire cu eroare (voluntar).
- Eroare (involuntar).
- Terminat de către un alt proces printr-un semnal (involuntar).

Un proces părinte ar putea termina execuţia unui proces copil în una din următoarele situaţii:

- Procesul copil a atins limita resurselor alocate (pentru a determina acest lucru, părintele trebuie să aibă un mecanism de inspactare al stării proceselor copil).
- Sarcina alocată procesului copil nu mai este necesară.

- Părintele îşi termină execuţia, iar sistemul de operare nu permite unui proces copil să continue dacă parintele său nu mai este activ.

Unele sisteme nu permit execuţia unui proces copil dacă părintele a terminat. În astfel de sisteme, dacă un proces termină, atunci se termină şi procesele copil. Acest fenomen este denumit **terminare în cascadă**.

Un proces părinte trebuie să aştepte terminarea proceselor copil prin intermediul apelului de sistem **wait**. Atunci cînd un proces îşi termină execuţia, resursele alocate pentru acesta sunt eliberate de către sistemul de operare, dar intrarea în tabela de procese a sistemului de operare asociată procesului va rămîne validă pînă în momentul în care procesul părinte apelează **wait** pentru a citi starea finală a procesului copil. Un proces care a terminat, dar pentru care părintele nu a apelat **wait** se numeşte **proces zombie**.

În cazul în care un proces părinte nu apelează **wait** şi îşi termină execuţia înainte ca procesul copil să termine, atunci procesul copil devine un **proces orfan**.

3.3 Ierarhii de procese

În unele sisteme de operare, în momentul în care un proces părinte crează un proces copil, acestea continuă să fie asociate în anumite feluri. Procesul copil poate crea de asemenea alte procese, formîndu-se astfel o ierarhie de procese.

Diferenţa între sistemele de operare bazate pe ierarhii de procese (UNIX) şi sistemele care consideră toate procesele egale (WINDOWS) este reprezentată de faptul că în sistemele bazate pe ierarhii doar procesul părinte poate controla procesul copil. În cazul sistemelor fără ierarhii de procese, în momentul creării unui nou proces, sistemul de operare oferă un identificator de proces părintelui, iar acel identificator poate fi trimis oricărui alt proces pentru a controla procesul copil.

3.4 Schimbare de context

Întreruperile au ca efect schimbarea modului de lucru al procesorului din utilizator în privilegiat şi execuţia codului sistemului de operare. Aceste situaţii sunt des întîlnite în sistemele cu scop general. Atunci cînd soseşte o întrerupere din partea unui dispozitiv, periferic, ceas, sau orice alt dispozitiv din sistem, sistemul de operare trebuie să salveze **contextul curent** al procesului care rulează în momentul respectiv, astfel încît acesta să poată fi refăcut după execuţia rutinei de tratare a întreruperii. Contextul este reprezentat în PCB-ul procesului. Acesta include valoarea registrelor procesorului, starea procesului şi informaţii de administrare a memoriei. Teoretic, se efectuează o salvare a stării curente a procesorului, urmată de o restaurare a stării acestuia.

Tranziţia procesorului de la un proces la altul necesită salvarea stării procesorului în cadrul procesului curent şi restaurarea stării procesorului în cadrul procesoului pe care acesta urmează sa îl ruleze. Această sarcină este cunoscută sub numele de **schimbare de context**. Atunci cînd se petrece o schimbare de context, sistemul de operare salvează contextul procesului vechi în PCB-ul său şi încarcă contextul salvat al procesului planificat pentru rulare.

Timpul necesar schimbării de context reprezintă pur overhead, întrucît sistemul nu efectuează calcule utile cît timp îşi schimbă contextul. Viteza schimbării de context variază de la maşină la maşină, aceasta depinzînd de viteza memoriei, numărul de registre ce trebuie copiate şi existenţa instrucţiunilor speciale (cum ar fi o singură instrucţiune pentru a încărca sau salva toţi regiştrii procesorului).

4 Curs 4 - Planificarea execuţiei. Comunicarea între procese

Obiective

- Introducerea noţiunii de planificare.
- Descrierea algoritmilor de planificare.
- Discutarea criteriului de evaluare pentru selecţia unui algoritm de planificare potrivit, fiind dat un sistem particular.

4.1 Concepte generale

Într-un sistem de calcul ce dispune de un singur procesor, un singur proces poate rula la un moment dat. Obiectivul multiprogramării este de a rula cît mai multe procese în timp, pentru a maximiza utilizarea procesorului. Pentru a îndeplini acest scop, un proces este executat pînă în momentul în care acesta trebuie să aştepte, în general pentru finalizarea unei operaţii de intrare/ieşire.

Într-un sistem cu resurse alocate, cît timp se aşteaptă completarea operaţiei de intrare/ieşire, procesorul în acest timp doar aşteaptă, iar acest timp este risipit. Pe de altă parte, într-un sistem cu resurse distribuite, timpul în care un proces aşteaptă completarea unei operaţii de intrare/ieşire este alocat unui alt proces. Mai multe procese sunt reţinute în memoria principală la un moment dat.

Tranziţia de la un proces la altul se numeşte **planificare**, iar planificarea este o funcţie fundamentală a unui sistem de opeprare cu timp partajat. Majoritatea resurselor sistemului de calcul sunt planificate înainte de a fi utilizate.

4.1.1 Comportamentul proceselor

Succesul unei planificări de procese depinde de observarea următoarei proprietăţi a acestora: execuţia unui proces reprezintă o serie de cicluri procesor şi cereri de intrare/ieşire. Procesele alternează între aceste două stări. Execuţia proceselor începe cu o rafală de cicluri procesor. Aceasta este urmată de o rafală de intrare/ieşire, care este urmată de o rafală de cicluri de procesor şi aşa mai departe. În funcţie de durata totală a ciclurilor de procesor, comparată cu durata totală a ciclurilor intrare/ieşire, se poate alege algoritmul de planificare potrivit.

4.1.2 Planificarea preemptivă

Algoritmii de planificare pot fi împărţiţi în două categorii din punctul de vedere al comportamentului în legătură cu întreruperile de ceas:

- **Nepreemptive**: Aceşti algoritmi aleg un proces pentru rulare, acesta putînd rula pînă în momentul în care se blochează (eveniment i/o sau aşteptarea unui alt proces), sau pînă în momentul în care eliberează procesorul în mod voluntar. Practic nu se iau decizii de planificare pe durata întreruperilor de ceas. După executarea întreruperii, procesul îşi continuă execuţia.
- **Preemtive**: Aceşti algoritmi planifică un proces şi alocă acestuia un timp fix de rulare. Dacă acesta rulează la expirarea cuantei de timp, acesta este suspendat iar planificatorul alege un alt proces. Pentru astfel de planificatoare, sunt necesare întreruperi de ceas la finalul cuantei de timp pentru a transfera controlul de la proces la planificator.

4.1.3 Momentele de planificare

O problemă legată de planificare este reprezentată de momentul în care este necesară o decizie de planificare. Decizia de planificare poate fi luată în următoarele situaţii:

1. Atunci cînd un proces în starea de rulare trece în starea de aşteptare.
2. Atunci cînd un proces trece din starea de rulare în starea de pregătit (atunci cînd ii expiră cuanta).
3. Atunci cînd un proces trece din starea de aşteptare în starea de pregătit (atunci cînd operaţia de intrare/ieşire s-a terminat).
4. Atunci cînd un proces îşi termină execuţia.
5. Atunci cînd un nou proces este creat.

În primul rînd, atunci cînd un proces se blochează pentru aşteptarea unui eveniment de intrare/ieşire, semafor sau alt motiv, alt proces trebuie selectat pentru a fi rulat. Uneori motivul blocării reprezintă un parametru pentru decizia planificării următorului proces. De exemplu, dacă A este un proces important şi aşteaptă ca B să iasă dintr-o secţiune critică, planificînd B pentru a-l lăsa să iasă din secţiunea critică, A va putea continua execuţia.

În al doilea rînd, avînd în vedere faptul că algoritmii de planificare preemptivi alocă fiecărui proces o durată maximă de execuţie, numită **cuantă de timp**, orice proces poate rula pînă la expirarea cuantei. Momentul în care atenţia procesorului este transferată de la proces la planificator este datorat de o întrerupere de ceas.

În al treilea rînd, atunci cînd se petrece un eveniment de intrare/ieşire, o decizie de planificare trebuie luată. Dacă întreruperea vine din partea unui dispozitiv periferic care şi-a terminat execuţia, un proces care a fost blocat pentru aşteptarea acestuia ar putea rula. Este datoria planificatorului să decidă ce proces va rula: unul nou creat, cel care se afla în aşteptare sau un alt proces.

În al patrulea rînd, o decizie de planificare trebuie luată ori de cîte ori un proces îşi termină execuţia. În acest caz, fie se alege un proces dintre cele pregătite, fie sistemul devine inactiv.

În cincilea rînd, atunci cînd un proces este creat, trebuie luată o decizie în legătură cu procesul care va urma să fie executat: părintele sau copilul. Avînd în vedere faptul că ambele procese sunt în starea **pregătit**, o decizie normală de planificare ar putea alege oricare dintre cele două procese.

Atunci cînd planificarea este efectuată doar în primul sau al patrulea caz, schema de planificare se numeşte **nepreemptivă** sau **cooperativă**, altfel este **preemptivă**. De altfel, conceptul de **multiprogramare** se referă la planificarea nepreemptivă, iar sistemele de tip time sharing, la planificare preemptivă.

4.1.4 Dispatcher-ul

O altă componentă implicată în planificare este funcţia de dispatcher. Acest modul oferă controlul asupra procesorului procesului selectat de către planificator. Această funcţie implică următoarele:

- Schimbarea contextului.
- Schimbarea stării procesorului în mod utilizator.
- Execuţia procesului din momentul în care a fost întrerupt.

Acest modul trebuie să fie cat mai rapid, avînd în vedere faptul că este executat la fiecare schimbare de proces. Timpul necesar dispatcher-ului pentru a opri un proces şi a porni altul este cunoscut sub numele de **latenţa dispatcher-ului**.

4.2 Criteriul de planificare

Diferiţi algoritmi de planificare au diferite proprietăţi, iar alegerea unui algoritm de planificare particular într-o situaţie particulară poate favoriza anumite clase de procese. Pentru a putea alege un anumit algoritm de planificare într-o situaţie particulară, trebuie considerate proprietăţile diferiţilor algoritmi.

Multe criterii au fost sigerate pentru compararea algoritmilor de planificare. Criteriile includ următoarele:

- Utilizare procesor. Scopul este de a menţine procesorul cît mai ocupat. Conceptual, utilizarea procesorului poate ajunge de la 0% pînă la 100%. Într-un sistem real, acest parametru are valori cuprinse între 40% şi 90%.
- Debit. O măsură a muncii efectuate de procesor este numărul de procese terminate raportat la unitatea de imp, numit **debit**. Pentru procese de durată lungă, această rată poate fi un proces pe oră; pentru tranzacţii scurte, ar putea fi 10 procese per secundă.
- Timpul de completare. Din punctul de vedere al unui proces particular, un criteriu important este reprezentat de intervalul de timp cuprins între momentul în care procesul a fost lansat în execuţie pînă cînd se termină. Acest timp este suma următoarelor perioade: timp de încărcare în memorie, aşteptare de planificare, execuţie pe procesor şi operaţii intrare/ieşire.
- Timp de aşteptare. Algoritmii de planificare nu afectează timpul în care un proces rulează sau efectuează i/o; afectează doar timpul petrecut într-o coadă de aşteptare în starea **pregătit**. Timpul de aşteptare este suma tuturor perioadelor în care un proces se află în starea **pregătit**.
- Timpul de răspuns. Într-un sistem interactiv, timpul de completare nu este neapărat cel mai bun criteriu. Deseori, un proces poate produce rezultate pe măsură ce se execută. Astfel, o altă măsură este reprezentată de timpul măsurat de la momentul în care utilizatorul trimite o cerere către un proces pînă în momentul în care procesul începe să răspundă.

Obiectivul este de a creşte utilizarea procesorului şi debit-ul, împreună cu minimizarea timpului de aşteptare, de răspuns şi de completare. Totuşi, este mult mai important de minimizat variaţia timpului de răspuns decît minimizarea timpului de răspuns mediu.

4.3 Categorii de algoritmi de planificare

Parametrii ce trebuie optimizaţi de către planificator nu sunt aceeaşi pentru toate tipurile de sisteme. Se disting trei tipuri de sisteme:

1. Sisteme cu resurse alocate.
2. Sisteme cu resurse distribuite
3. Sisteme de timp real.

În sistemele cu resurse alocate, timpul de răspuns nu este un parametru critic. Astfel, algoritmii nepreemptivi sau algoritmii preemptivi ce alocă cuante mari de timp sunt acceptati. Această abordare minimizează schimbările de context, iar astfel împunătăţeşte performanţa.

Într-un mediu cu utilizatori interactivi, preemptarea este esenţială pentru a aloca resursele cît mai optim. Chiar dacă un proces rulează la nesfîrşit din cauza unei erori de proiectare, un alt proces poate opri alte procese. Preemptarea este necesară pentru a preveni un astfel de comportament.

În sisteme de timp real preemptarea nu este întotdeauna necesară, întrucît procesele sunt proiectate astfel încît să nu ruleze o perioadă lungă de timp, iar acestea îşi efectuează sarcinile şi se blochează rapid. Cît timp sistemele interactive sunt de scop general şi pot rula programe arbitrare ce nu sunt cooperative, în cazul sistemelor de timp real, toate procesele care rulează au un scop predefinit.

4.4 Scopurile algoritmilor de planificare

Există anumite scopuri care trebuie îndeplinite de algoritmii de planificare, indiferent de tipul de sistem pentru care aceştia sunt proiectaţi:

- Egalitate: oferirea oricărui proces o şansă egală la procesor.
- Balance: menţinerea tuturor părţilor sistemului ocupate.

În cazul sistemelor cu resurse alocate, trebuie îndeplinite următoarele:

- Debit: maximizarea proceselor per oră.
- Timp de completare: minimizarea timpului dintre submitere şi terminare.
- Utilizare procesor: menţinerea procesorului ocupat.

În cazul sistemelor cu resurse distribuite, trebuie îndeplinite următoarele:

- Timp de răspuns: minimizarea timpului de răspuns.
- Proporţionalitate: îndeplinirea aşteptărilor utilizatorului.

În cazul sistemelor de timp real, trebuie îndeplinite următoarele:

- Respectarea termenelor limită: evitarea situaţiilor în care se pierd date.
- Predictibilitate: evitarea degradării calităţii în sistemele multimedia.

În orice tip de sistem, egalitatea este importantă. Procese comparabile ar trebui să primească servicii comparabile. Alocînd unui proces mult mai mult procesor decît unui alt proces echivalent nu este corect. Desigur, diferite categorii de procese ar putea fi tratate diferit.

Un alt scop general este menţinerea tuturor părţilor sistemului ocupate. Dacă procesorul împreună cu toate dispozitivele periferice pot fi menţinute ocupate, eficienţa sistemului creşte. Avînd unele procese consumatoare de procesor împreună cu unele procese consumatoare de date în memorie este mai eficient decît rularea tuturor proceselor consumatoare de procesor, urmate de rularea proceselor consumatoare de date.

4.5 Algoritmi de planificare

4.5.1 Planificare în sisteme cu resurse alocate

FIFO

Cel mai scurt proces primul

Cel mai scurt timp rămas următorul

Trei nivele de planificare

4.5.2 Planificare în sisteme cu resurse distribuite

Round-Robin

Planificare cu priorităţi

Cozi multiple

Cel mai scurt proces următorul

Planificare garantată

Planificare aleatoare

Planificare distribuită egal

4.6 Evaluarea algoritmilor

4.7 Comunicare între procese

Procesele care se execută concurent într-un sistem de operare pot fi fie independente, fie cooperative. Un proces este **independent** dacă nu poate afecta sau poate fi afectat de alte procese care se execută în sistem. Orice proces care nu distribuie informaţii cu alt proces este independent. Un proces este **cooperativ** dacă poate afecta sau poate fi afectat de alte procese care se execută în sistem.

Există cîteva motive pentru a permite cooperarea între procese:

- **Distribuirea informaţiei**: Întrucît mai mulţi utilizatori pot fi interesaţi de aceeaşi informaţie, sistemul de operare trebuie să permită accesul concurent la o astfel de informaţie.
- **Accelerarea calculului**.
- **Modularitate**.
- **Comoditate**.

Procesele cooperative necesită un mecanism de comunicare între procese care le va permite schimbul de informaţie. Există două modele fundamentale a comunicaţiei interproces:

- Memorie partajată: o regiune de memorie este comună proceselor cooperative. Procesele pot interschimba informaţii prin scrierea şi citirea datelor în regiunea partajată.
- Transmitere de mesaje: comunicaţia este reprezentată de mesajele interschimbate între procesele cooperative.

Ambele modele sunt comune în sistemele de operare, iar majoritatea sistemelor de operare le implementează pe ambele. Pasarea de mesaje este utilă pentru comunicaţia de mesaje scurte şi este mai uşor de implementat. Memoria partajată poate fi mai rapidă, întrucît pasarea de mesaje este implementată prin intermediul apelurilor de sistem. În memoria partajată, singurele apeluri de sistem necesare sunt cele de stabilire a memoriei partajate. Din pricina coerenţei cache-ului, pasarea de mesaje este mai rapidă decît memoria partajată.

4.7.1 Memoria partajată

Comunicarea interproces utilizînd memoria partajată necesită procesele comunicatoare să stabilească o zonă de memorie partajată. Teoretic, o zină de memorie partajată este alocată în spaţiul de adresă al procesului care solicită această zonă. Orice alt proces ce doreşte comunicarea cu acel proces trebuie să ataşeze acea zonă de memorie în spatiul lor de adresă. În general, sistemul de operare încearcă să prevină accesul unui proces în zona de memorie a unui alt proces. Memoria partajată necesită ca două sau mai multe procese să înlăture această restricţie. Procesele sunt atunci responsabile să îşi stabilească reguli pentru a păstra consistenţa datelor.

4.7.2 Pasarea de mesaje

Pasarea de mesaje pune la dispoziţie un mecanism prin care permite proceselor să comunice şi să îşi sincronizeze acţiunile fără a avea acces la acelaşi spaţiu de adresă. În general, acest mecanism se găseşte a fi util în sistemele distribuite, unde procesele comunicatoare se regăsesc pe sisteme diferite concetate printr-o reţea.

Transmiterea de mesaje se realizează prin intermediul a două operaţii: **send** şi **receive**. Mesajele trimise de un proces pot fi de dimensiune fixă sau variabilă.

Dacă două procese doresc să comunice, acestea trebuie să trimită mesaje unul celuilalt: o legătură de comunicaţie trebuie să existe între acestea. Această legătură poate fi implementată în trei feluri:

- Comunicaţie directă sau indirectă.
- Comunicaţie sincronă sau asincronă.
- Tehnici de buffering automate sau explicite.

Prin comunicaţia directă, fiecare proces trebuie să numească direct procesul destinatar. O legătură de comunicaţie în acest caz are următoarele proprietăţi:

- Legătura este stabilită automat între fiecare pereche de procese ce doresc să comunice. Procesele trebuie să cunoască doar identitatea destinatarului direct.
- O legătură există doar între două procese.

Această schemă prezintă simetrie în adresare, astfel procesul expeditor şi procesul destinatar trebuie să cunoască adresele pentru a comunica. O altă variantă a acestei scheme implică asimetrie în

adresare. Aici, doar transmiţătorul numeşte destinatarul, destinatarul nefiind obligat să cunoască numele transmiţătorului.

Dezavantajele ambelor scheme este reprezentată de modularitatea limitată a proceselor. În cazul în care un proces îşi schimbă identificatorul, toate celelalte procese trebuie notificate pentru a cunoaşte noul identificator al procesului respectiv.

4.7.3 Pipe-uri

Două mecanisme primare de comunicaţie între procese sunt reprezentate de pipe-urile anonime şi cu nume. Pipe-urile anonime pot fi utilizate pentru redirectarea rezultatelor unui proces către alt proces, fiind bazate pe caractere şi half-duplex. Pipe-urile cu nume sunt mult mai puternice decît cele anonime. Acestea sunt full-duplex şi orientate pe mesaje, permiţînd comunicaţie de reţea.

Pipe-uri anonime

Acest tip de pipe-uri permit comunicaţia între două procese în maniera producător-consumator: producătorul scrie la un capăt al pipe-ului iar consumatorul citeşte de la celălalt capăt. Astfel, acest tip de pipe-uri este unidirecţional. Dacă se doreşte comunicaţie bidirecţională, trebuie utilizate două pipe-uri.

Un pipe anonim nu poate fi accesat din afara procesului în care a fost creat. Teoretic, un proces părinte crează un pipe şi îl utilizează să comunice cu un proces copil pe care îl crează prin apelul de sistem **fork**.

Pipe-uri cu nume

Un pipe anonim există atîta timp cît procesul care a creat pipe-ul rulează. Astfel, după încheierea execuţiei acestui proces, pipe-ul anonim nu mai există în sistem. Pipe-urile cu nume pun la dispoziţie o metodă mult mai puternică de comunicaţie. Comunicaţia este bidirecţională şi nu este necesară o relaţie tată-fiu. O dată ce un pipe cu nume este creat, mai multe procese îl pot utiliza pentru comunicaţie. În general, un pipe cu nume are mai mulţi producători, iar aceste pipe-uri există după terminarea execuţiei proceselor ce le utilizează.

Aceste pipe-uri sunt văzute în sistemul de operare drept fişiere. Chiar dacă aceste pipe-uri permit comunicaţie bidirecţională, doar transmisie half-duplex este permisă. Dacă este necesară o comunicaţie bidirecţională, atunci două pipe-uri trebuie utilizate, în cazul sistemelor unix.

5 Curs 5 - Gestiunea memoriei

Obiective

- Detalierea căilor de a organiza memoria în hardware.
- Discutarea tehnicilor de administrare a memoriei, incluzînd paginarea şi segmentarea.

5.1 Concepte generale

Memoria este reprezentată de şiruri de cuvinte de memorie sau octeţi, fiecare avînd propria sa adresă. Procesorul aduce instrucţiunile din memoria principală în funcţie de contorul de program. Aceste instrucţiuni pot avea drept consecinţă stocări sau citiri din memorie.

Unitatea de memorie primeşte ca argument adrese de memorie, împreună cu operaţia ce trebuie efectuată asupra acestora, iar în cazul stocărilor, valoarea ce trebuie reţinută la acea adresă.

5.1.1 Hardware de bază

Memoria principală împreună cu registrele procesorului sunt singurele unităţi de stocare a informaţiei pe care procesorul le poate accesa direct. Există instrucţiuni din ISA care iau ca argument adrese de memorie, dar nu există instrucţiuni care să accepte adrese ale disc-ului. Astfel, orice instrucţiune în execuţie şi orice informaţie utilizată de instrucţiuni trebuie să se regăsească într-unul din aceste dispozitive de stocare directe. Dacă datele nu sunt în memoria principală sau în registrele procesor, acestea trebuie mutate înainte ca procesorul să le poată accesa.

Registrele sunt parte a procesorului, iar astfel acestea sunt accesibile într-un singur ciclu de ceas. În cazul memoriei principale, acest lucru nu este valabil, întrucît accesul la memorie reprezintă o tranzacţie pe magistrala de memorie. Completarea unui acces la memorie poate dura multe cicluri de ceas. În astfel de cazuri, procesorul în general aşteaptă, întrucît nu are datele necesare pentru a completa o instrucţiune în execuţie. Această situaţie nu este tolerată din cauza timpului de acces la memorie. Un remediu este reprezentat de adăugarea unei memorii mai rapide între procesor şi memoria principală. O zonă tampon este utilizată pentru a reduce diferenţa de timp de acces, numit **cache**.

Pe lînga timpul de acces al memoriei, trebuie ţinut cont şi de aspecte de securitate, cum ar fi accesul unui utilizator neprivilegiat la codul sistemului de operare, sau accesul unui proces în spaţiul de adresa al altui proces. Această protecţie trebuie efectuată în hardware.

În primul rînd trebuie să asigurăm că fiecare proces are un spaţiu de adresă separat. Pentru a realiza această sarcină, trebuie determinat spaţiul de adresă pe care un proces îl poate accesa şi să asigurăm faptul că acel proces accesează doar acel spaţiu de adresă. Pentru a implementa aceste cerinţe, se pot utiliza doi regiştrii, unul de bază şi unul de limită. Registrul de bază reţine cea mai mică adresă fizică permisă; registrul de limită specifică dimensiunea intervalului.

Protecţia spaţiului de adresă poate fi îndeplinită prin suport hardware care să compare **toate** adresele generate în mod utilizator cu regiştrii menţionaţi anterior. Orice acces la o zonă de memorie care nu se află în intervalul specificat de cei doi regiştrii va cauza o excepţie, iar sistemul de operare tratează această excepţie ca eroare fatală.

Registrele de bază şi limită pot fi modificate doar de către sistemul de operare, care utilizează instrucţiuni privilegiate pentru acest scop. Întrucît instrucţiunile privilegiate pot fi executate doar în modul privilegiat, iar instrucţiunile în mod privilegiat pot fi executate doar de către sistemul de operare, doar sistemul de operare poate modifica aceşti regiştrii.

Astfel, sistemul de operare, executat în modul privilegiat, are acces nerestricţionat la memoria sistemului de operare şi memoria utilizatorului. Această proprietate permite sistemului de operare să încarce programe utilizator în memoria utilizatorului, să elibereze memoria în cazul unei erori, să modifice şi să acceseze parametrii apelurilor de sistem, etc.

5.1.2 Legarea adreselor

În general, legarea instrucţiunilor şi a datelor în adrese de memorie poate fi efectuată în următorii paşi:

- La compilare: Se generează cod absolut, dacă se cunoaşte în momentul compilării locaţia de memorie la care se va încărca executabilul, atunci codul obţinut la compilare va începe la respectiva adresă. În cazul sistemului de operare MS-DOS, programele sunt legate în momentul compilării.
- La încărcare: Dacă nu se cunoaşte la momentul compilării locaţia de memorie unde va fi încărcat executabilul, atunci compilatorul trebuie să genereze **cod relocabil**. În acest caz, legarea finală este întîrziată pînă la momentul încărcării în memorie. Dacă adresa de start se schimbă, trebuie reîncărcat codul utilizatorului pentru a incorpora valoarea schimbată.
- La execuţie: Dacă procesul poate fi mutat cît timp se află în execuţie de la un segment de memorie la altul, legarea trebuie mutată pînă la momentul rulării. Pentru această metodă este necesar suport hardware. Majoritatea sistemelor de operare de scop general utilizează această metodă.

5.1.3 Spaţiul de adresă logic vs virtual

O adresă generată de procesor reprezintă în general o **adresă logică**, în timp ce o adresă văzută de unitatea de memorie - mai exact, una încărcată în registrul de adrese al memoriei - reprezintă o **adresă fizică**.

Legarea la compilare şi încărcare a adreselor de memorie generează adrese logice şi fizice identice. Schema de legare la momentul execuţiei are ca rezultat adrese fizice şi logice diferite. În acest caz, adresele logice sunt referite ca **adrese virtuale**. Mulţimea tuturor adreselor logice generate de un program reprezintă **spaţiul logic de adresă**. Similar, mulţimea tuturor adreselor fizice asociate adreselor logice reprezintă **spaţiul fizic de adrese**.

Maparea adreselor din spaţiul virtual în spaţiul fizic de adrese este realizată de un dispozitiv hardware numit **unitate de administrare a memoriei**. Există multe metode de realizare a mapării ce vor fi discutate în subcapitole ulterioare. Schema de mapare a unei astfel de unităţi pentru modelul ce implică cei doi regiştri, bază şi limită sunt discutaţi în continuare.

Registrul bază este acum denumit **registru de relocare**. Valoarea din registrul de relocare este adăugată fiecărei adrese generate de un proces utilizator în momentul în care aceasta este trimisă către memorie. Sistemul de operare MS-DOS rulînd procesoare din familia Intel 80x86 utiliza patru registre de relocare atunci cînd rula şi încărca un proces.

Programul utilizator niciodată nu lucra cu adrese fizice reale, acesta lucrînd doar cu adrese logice. Dispozitivele hardware pentru mapare a memoriei convertesc adresele logice în adrese fizice. Locaţia finală a adresei de memorie nu este determinată pînă nu este efectuat un acces la memorie.

În momentul de faţă există două tipuri de adrese: logice (în intervalul [0, max]) şi fizice (în intervalul [R + 0, R + max], unde R este valoarea din registrul de relocare). Utilizatorul generează doar adrese logice şi are impresia că procesul accesează memoria în intervalul [0, max].

Conceptul spaţiului de adresă logic ce este legat de un spaţiu de adresă fizic separat este central pentru o administrare a memoriei corecte.

5.1.4 Încărcare dinamică

Pînă în momentul de faţă a fost necesar ca întregul program şi toate datele unui proces să se afle în memoria fizică pentru ca un proces să poată fi executat. Dimensiunea procesului este atunci limitată la dimensiunea fizică a memoriei principale. Pentru a obţine o utilizare mai eficientă a memoriei poate fi utilizată **încărcarea dinamică**. Cu încărcarea dinamică, o rutină nu este încărcată pînă cînd nu este apelată. Toate rutinele sunt păstrate pe disc în format relocabil. Programul principal este încărcat în memorie şi executat. Atunci cînd o rutină necesită apelarea unei alte rutine, rutina care efectuează apelul prima dată verifică dacă rutina pe care doreşte să o apeleze este încărcată în memorie. Dacă nu este încărcată, încărcătorul de legături relocabile este chemat să încarce rutina respectivă în memorie şi să modifice spaţiul de adresă virtual al procesului pentru a putea accesa respectiva rutină, oferind controlul înapoi procesului care se execută.

Avantajul încărcării dinamice este reprezentat de faptul că rutinele neapelate nu sunt încărcate în memorie. Această metodă este utilă în cazul în care bucăţi mari de cod sunt necesare pentru a trata cazuri petrecute cu frecvenţă redusă, cum ar fi rutine de eroare. În acest caz, chiar dacă dimensiunea totală a programului este mare, porţiunea utilizată poate fi mult mai mică.

Încărcarea dinamică nu necesită suport special din partea sistemului de operare. Este responsabilitatea programatorilor să utilizeze avantajele acestei metode. Sistemul de operare poate ajuta programatorul punînd la dispoziţie biblioteci pentru a implementa această metodă.

5.1.5 Legare dinamică şi biblioteci partajate

Unele sisteme de operare oferă suport doar pentru **legare statică de cod**, în care bibliotecile sunt tratate ca orice modul obiect şi sunt combinate de către loader în imaginea binară a programului. Legarea dinamică, pe de altă parte, este similară cu încărcarea dinamică. Aici legarea este întîrziată pînă la momentul execuţiei. Această caracteristică este utilizat în general cu biblioteci de sistem, cum ar fi subrutine ale limbajului de programare. Fără această facilitate, fiecare program din sistem ar trebui să includă o copie a bibliotecii limbajului în executabil. Aceste necesităţi risipesc spaţiu pe disc şi în memoria principală.

Cu legarea dinamică, este necesară includerea unei bucăţi mici de cod ce indică cum să fie alocate biblioteci rezidente în memorie, sau cum să fie încărcată o bibliotecă dacă o rutină a acesteia nu este disponibilă. Atunci cînd se execută acea bucată de cod, se verifică dacă există deja în memorie rutina necesară. Dacă nu există, programul încarcă rutina în memorie, astfel bucata iniţială de cod este înlocuită cu rutina necesară. Procedînd în acest fel, la un moment ulterior de timp cînd se reapelează rutina respectivă, rutina bibliotecii respective este executată direct, fără a plăti costul legării dinamice. Utilizînd această schemă, toate procesele ce utilizează o blbiotecă de limbaj execută doar o copie a codului bibliotecii.

Legarea dinamică poate fi extinsă la update-uri de biblioteci. O bibliotecă poate fi înlocuită cu o nouă versiune, iar toate programele care fac referire la respectiva bibliotecă vor utiliza automat noua versiune. Fără legare dinamică, toate programele care ar utiliza respectiva bibliotecă ar trebui recompilate pentru a avea acces la noua bibliotecă. Mai mult de o versiune a unei biblioteci poate fi încărcată în memorie, iar astfel orice program poate utiliza ce versiune doreşte a unei biblioteci date, dacă aceasta este valabilă în sistem. Versiunea bibliotecii utilizată de un program este definită la compilare, iar în momentul încărcării în memoria principală, loader-ul va încărca versiunea de bibliotecă cerută de executabil. Acest sistem este cunoscut sub numbele de **biblioteci partajate**.

Spre deosebire de încărcarea dinamică, legarea dinamică în general necesită ajutor din partea sistemului

de operare. Dacă un proces în memorie este protejat de un alt proces, atunci sistemul de operare este singura entitate care poate verifica dacă rutina necesară este prezentă în memoria unui alt proces sau dacă poate permite accesul mai multor procese către aceeaşi adresă de memorie.

5.2 Swapping

Un proces trebuie să se afle în memorie pentru a fi executat. De asemenea, un proces poate fi mutat temporar în afara memoriei principale cît timp acesta nu este planificat pentru rulare, urmînd să fie reintrodus în momentul planificării acestuia. În mod normal, un proces care a fost înlăturat din memoria principală cît timp nu era planificat, în momentul în care este replanificat va ocupa acelaşi spaţiu de adresă. Această restricţie este dictată de metoda de legare a adreselor. Dacă legarea este realizată la asamblare sau încărcare, atunci procesul nu poate fi mutat uşor într-o locaţie diferită. Dacă se utilizează legare la momentul execuţiei, atunci procesul poate fi mutat într-un alt spaţiu de adresă, întrucît adresele fiziice sunt calculate în momentul execuţiei instrucţiunilor.

Operaţia de swap necesită un spaţiu de stocare, reprezentat de un dispozitiv de memorie secundară rapid. Acesta trebuie să dispună de o capacitate suficient de mare astfel încît toate copiile imaginilor de memorie pentru toate procesele rulate în sistem să poată fi stocate şi să pună la dispoziţie acces direct la aceste imagini. Sistemul de operare reţine o **coadă de procese pregătite**, în care sunt reţinute toate procesele ale căror imagini de memorie se află pe dispozitivul de memorie secundară şi sunt în starea **pregătit**. În momentul în care planificatorul decide să execute un proces, acesta apelează dispatcher-ul. Dispatcher-ul verifică dacă următorul proces din coadă este în memorie. Dacă nu este şi nu există memorie suficientă pentru încărcarea acestuia, atunci dispatcher-ul efectuează o operaţie de swap a unui proces din memoria principală, pentru a putea muta procesul ce urmează să fie planificat în memoria principală, urmînd să refacă contextul procesului planificat.

O parte semnificativă a timpului de swap este reprezentată de timpul de transfer. Timpul total de transfer este direct proporţional cu cantitatea de memorie pentru care se efectuează operaţia de swap. O optimizare ar putea fi făcută dacă am putea afla cîtă memorie utilizează un proces, nu cît de multă memorie ar putea utiliza. Astfel, am putea încărca în memorie doar procedurile care sunt necesare execuţiei procesului, reducînd timpul de swap.

Operaţia de swap poate fi efectuată **doar în cazul proceselor care se află în starea pregătit**. Dacă dorim mutarea unui proces din memoria principală, dar acesta este blocat pentru o operaţie i/o, atunci dacă dispozitivul periferic accesează memoria procesului asincron pentru zone de tampon, atunci procesul nu poate fi evacuat din memorie. Există două soluţii la această problemă: nu se efectuează swap pentru procesele blocate, sau execuţia operaţiilor I/O doar în zonele tampon ale sistemului de operare, astfel transferurile între zonele tampon ale sistemului de operare şi memoria procesului se petrec doar în momentul în care procesul se află în memorie.

5.3 Alocare continuă de memorie

Memoria principală trebuie să reţină sistemul de operare împreună cu procesele utilizator. Astfel, este necesară alocarea memoriei principale într-o manieră cît mai eficientă. O metodă comună utilizată pentru îndeplinirea acestui scop este alocarea continuă de memorie.

În general, memoria principală este împărţită în două porţiuni: sistemul de operare şi procesele utilizator. Sistemul de operare poate fi poziţionat fie la adrese mici(începutul spaţiului de adresă), fie la adrese mari(sfîrşitul spaţiului de adresă). Decizia este luată ţinînd cont de poziţionarea vectorului de întreruperi. Întrucît acest vector este în general localizat la adrese mici, programatorii poziţionează

sistemul de operare de asemenea la adrese mici.

În general este de preferat ca mai multe procese utilizator să se afle în memorie la un moment dat. Astfel, trebuie avut în vedere metoda de alocare a memoriei disponibile proceselor ce aşteaptă să fie introduse în memorie. În alocarea continuă de memorie, fiecare proces este reţinut într-o singură zonă continuă de memorie.

5.3.1 Maparea de memorie şi protecţia

Pentru a efectua maparea de memorie, se poate utiliza un registru de relocare împrună cu un registru de limită, aşa cum a fost discutat anterior. Cu aceşti doi regiştrii, fiecare adresă logică trebuie să fie mai mică decît valoarea din registrul de limită; unitatea de administrare a memoriei mapează adresele logice dinamic prin adunarea adresei logice cu valoarea registrului de relocare. Această adresă mapată este trimisă memoriei principale.

Atunci cînd planificatorul selectează un proces pentru execuţie, dispatcher-ul încarcă registrele de relocare şi limită cu valorile corecte drept parte a schimbării de context. Întrucît fiecare adresă generată de către procesor este verificată împotriva acestor regiştrii, putem proteja sistemul de operare şi celelalte programe utilizator de posibilele modificări efectuate de procesul curent.

5.3.2 Alocarea memoriei

Una dintre cele mai simple metode pentru alocarea memoriei este împărţirea memoriei în părţi de dimensiune fixă. Fiecare parte conţine exact un proces. Atunci cînd o parte a memoriei este liberă, un proces poate fi încărcat în aceasta. Atunci cînd procesul îşi termină execuţia, un nou proces poate utiliza spaţiul de memorie pe care vechiul proces îl ocupa.

O altă metodă este reprezentată de reţinerea unei tabele în care se regăsesc părţile de memorie libere şi ocupate. Iniţial, toată memoria este disponibilă pentru programele utilizator şi este considerată drept un bloc mare de memorie disponibilă. Pe măsură ce diferite procese ocupă, respectiv eliberează memoria principală, se formează mai multe blocuri libere de memorie, de dimensiuni variabile, întrucît procesele încărcate în memorie nu sunt, în general, de aceeaşi dimensiune.

Această procedură reprezintă un caz particular al problemei de alocare dinamică a memoriei, care se referă la modul de satiisfacere a unei cereri de mărime n dintr-o listă de părţi de memorie libere. Există multe soluţii pentru această problemă:

- **Prima potrivire**: Se alocă prima porţiune de memorie care este suficient de mare şi este cea mai mare. Căutarea poate începe fie a începutul mulţimii de porţiuni, fie de la locaţia la care să găsit ultima porţiune liberă.

- **Cel mai potrivit**: Se alocă prima porţiune de memorie care este suficient de mare şi este cea mai mică. Trebuie căutată întreaga listă de porţiuni, dacă aceasta nu este sortată după dimensiune. Strategia aceasta produce cele mai mici resturi de memorie, ceea ce înseamnă că iniţial se va utiliza memoria eficient, dar pe măsură ce procesele sunt încărcate, respectiv scoase din memorie, resturile de memorie liberă rămase sunt inaccesibile.

- **Cel mai nepotrivit**: Se alocă cea mai mare porţiune de memorie liberă. Această strategie produce cele mai mari resturi de memorie. Această strategie permite un număr mic de procese să fie încărcate a un moment dat în memorie, dar posibilitatea de a avea porţiuni de memorie liberă ce nu poate fi alocată scade, comparativ cu strategia *cel mai potrivit*.

Simulările au arătat că strategiile prima potrivire şi cel mai potrivit sunt mai eficiente decît cel mai nepotrivit în termenii utilizării de memorie şi scăderea timpului de execuţie. Prima potrivire este în general mai rapidă.

5.3.3 Fragmentarea

Prima potrivire şi cea mai bună potrivire suferă de **fragmentare externă**. Pe măsură ce procesele sunt încărcate şi scoase din memorie, spaţiul de memorie liberă este divizat în porţiuni de dimensiune mică. Fragmentarea externă există în situaţia în care cantitatea de memorie liberă totală este suficientă pentru a încărca un nou proces, dar această memorie nu este continuă. Spaţiul de stocare este fragmentat într-un număr mare de spaţii mici de memorie.

Soluţia fragmentării externe este reprezentată de alocarea în blocuri fixe de memorie. Cu această abordare, memoria alocată unui proces ar putea depăşi necesitatea procesului, astfel ajungînd la **fragmentare internă** - memorie neutilizată în interiorul unui bloc.

5.4 Paginare

Paginarea este o schemă de administrare a memoriei ce permite spaţiului de adresă fizic să nu fie continuu. Paginarea evită fragmentarea externă. Tradiţional, suportul pentru paginare era implementat în hardware. Totuşi, proiectările recente au implementat paginarea prin integrarea hardware-ului şi a sistemului de operare, în special pentru procesoare pe 64 de biţi.

5.4.1 Metoda de bază

Meotda de bază pentru implementarea paginării implică împărţirea memoriei fizice în blocuri de dimensiune fizică numite **cadre** şi împărţirea spaţiului de adresă logic în blocuri de aceeaşi dimensiune numite **pagini**. Atunci cînd un proces se află în execuţie, paginile sale sunt încărcate în orice cadre disponibile de memorie. Atunci cînd este utilizată o schemă de paginare, nu există fragmentare externă: orice cadru liber poate fi alocat unui proces. Totuşi, există fragmentare internă. Dacă memoria necesară unui proces nu coincide cu limitele unei pagini, atunci ultimul cadru alocat ar putea să nu fie complet ocupat.

Paginarea este tehnica de divizare a unui program în blocuri mai mici cu dimsniuni identice şi stocarea acestor bllocuri în memoria secundară sub forma unor *pagini*. Prin utilizarea avantajului localităţii referinţelor, aceste pagini pot fi încărcate apoi în memoria principală în blocuri de aceeaşi dimensiune numite **cadre de pagină** şi pot fi executate ca şi cum întregul proces ar fi încărcat în memorie. Pentru ca această metodă să funcţioneze corect, fiecare proces trebuie să păstreze o tabelă de adrese ale memoriei, numită **tabela de pagini**, în memoria principală. Fiecare adresă de pagină are o adresă corespunzătoare unui cadru de pagină.

Dimensiunea unei pagini este definită de către hardware. Această dimensiune este în general o puterea a lui 2, variind de la 512 octeţi pînă la 16MB, depinzînd de arhitectura sistemului de calcul. Alegînd astfel dimensiunea unei pagini uşurează procesul de translatare al adreselor fizice în număr de pagină şi deplasament de pagină uşor. Dacă dimensiunea unei adrese logice este 2^m iar dimensiunea unei pagini este 2^n, atunci cei mai semnificativi $m-n$ biţi reprezintă numărul de pagină, iar restul de n biţi deplasamentul de pagină.

Fiecare adresă generată de procesor este împărţită în două părţi: un **număr de pagină** şi un **deplasament de pagină**. Numărul de pagină este utilizat drept index într-o **tabelă de pagini**. Tabela de pagini conţine adresa de bază pentru fiecare pagină în memoria fizică. Această adresă de bază combinată cu deplasamentu paginii definesc adresa de memorie fizică ce este transmisă unităţii de memorie. Practic,

un număr de biţi din adresa de memorie utilizată de procesor reprezintă numărul paginii din tabela de pagini a procesului. După ce se extrage adresa cadrului asociat numărului de pagini din tabela de pagini, se utilizează restul de biţi din adresa utilizată de procesor pentru a forma adresa de memorie fizică care va fi transmisă unităţii de memorie.

O adresă logică constă din două părţi: o adresă de bază şi un deplasament. Fiecare proces are un registru de bază, care conţine adresa de început a tabelei de pagini a procesului. Tabelele de pagini au cîte o intrare pentru fiecare pagină pe care o conţine procesul. Aceste intrări conţin de obicei un cîmp de **prezenţă** de un bit, un cîmp de **acces** şi un cîmp de **adresă**.

Cîmpul de prezenţă specifică dacă pagina este prezentă în memoria principală. Cîmpul de acces specifică tipul operaţiilor care pot fi executate asupra paginii. Acest cîmp determină dacă pagina poate fi doar citită, citită şi scrisă, sau doar executată. Cîmpul de adresă specifică numărul cadrului în care se încarcă pagina. Adresa de început a paginii în memoria principală este evaluată înmulţind numărul cadrului cu dimensiunea cadrului. Dacă o pagină nu a fost încărcată în memorie, în cîmpul de adresă se păstrează adresa paginii în memoria secundară.

Atunci cînd se întîlneşte un acces la o variabilă sau o instrucţiune care nu este încarcată în memorie, apare o **lipsă de pagină** (page fault), iar pagina care conţine variabila sau instrucţiunea necesară este încărcată în memorie. Pagina este depusă într-un cadru liber, dacă un asemenea cadru există. Dacă nu există un cadru liber, trebuie selectată una din pacinile procesului, iar noua pagină va fi memorată în locul acesteia. Criteriul pentru selectarea paginii care se va înlocui constituie **strategia de înlocuire** sau **algoritmul de înlocuire**.

Întrucît sistemul de operare administrează memoria fizică, acesta trebuie să cunoască detaliile de alocare ale acesteia - ce cadre sunt alocate, ce cadre sunt disponibile, cîte cadre există şi aşa mai departe. Această informaţie este în general reţinută într-o structură de date numită **tabela de cadre**. Aceasta are o intrare pentru fiecare cadru fizic, indicînd dacă este alocat sau liber, iar dacă este alocat, pagina în care este alocat, al procesului sau proceselor care îl utilizează.

De asemenea, sistemul de operare trebuie să ţină cont de faptul că programele utilizator operează în spaţiul utilizator, iar toate adresele logice trebuie mapate pentru a produce adrese fizice. Dacă un program utilizator efectuează un apel de sistem şi pune la dispoziţie o adresă drept parametru, acea adresă trebuie să fie mapată pentru a putea accesa adresa fizică corect. Sistemul de operare reţine o copie a tabelei de pagini pentru fiecare proces pentru a putea determina dacă o adresă logică este mapată şi pentru a translata adresa logică în adresă fizică, dacă aceasta este mapată. Este de asemenea utilizată de către dispatcher pentru a defini tabela de pagini hardware atunci cînd un proces este planificat pe procesor. Astfel, paginarea creşte timpul de schimbare de context.

5.4.2 Suport hardware

Fiecare sistem de operare are propriile metode pentru a stoca tabelele de pagini. Majoritatea alocă cîte o tabelă de pagini pentru fiecare proces. Un pointer la tabela de pagini este stocat împreună cu valorile altor regiştrii (cum ar fi contor instrucţiune) în PCB. Atunci cînd dispatcher-ul porneşte un proces, acesta trebuie să reîncarce registrele procesului şi să definească corect valorile tabelei de pagini din hardware din tabela de pagini stocată a procesului.

Implementarea în hardware a tabelei de pagini poate fi făcută în cîteva moduri. În cel mai simplu caz, tabela de pagini este implementată ca un set de registre dedicate. Aceste registre trebuie construite astfel încît să fie foarte rapide, pentru a oferi o translatare pagină-adresă eficientă. Fiecare acces la memorie presupune accesul acestor registre. Dispatcher-ul reîncarcă aceste registre în aceeaşi manieră în care reîncarcă restul registrelor. Instrucţiunile pentru a încărca valori în aceste registre sunt privilegiate,

deci doar sistemul de operare poate schimba maparea memoriei.

Utilizarea registrelor pentru tabela de pagini este o soluţie satisfăcătoare dacă tabela de pagini este mică (256 de intrări). Soluţia standard pentru această problemă este de a utiliza o memorie cache în hardware, numită buffer de translatare TLB (translation look-aside buffer). TLB este o memorie asociativă de mare viteză. Fiecare intrare în TLB are două părţi: o cheie şi o valoare. Atunci cînd memoriei asociative îi este prezentată o cheie, acea cheie este comparată cu toate cheile simultan. Dacă cheia este găsită, valoarea asociată este întoarsă. Căutarea este rapidă, dar hardware-ul este scump. În general, numărul de intrări într-un TLB este mic (64 pînă la 1024 intrări).

TLB-ul este utilizat împreună cu tabela de pagini în următorul fel. Atunci cînd o adresă logică este generată de către procesor, numărul de pagini al acesteia este prezentat către TLB. Dacă numărul de pagină este regăsit în TLB, numărul de cadru al acesteia este disponibil imediat şi utilizat pentru accesul la memorie. Dacă numărul de pagină nu este în TLB, trebuie efectuată o referinţă în memorie către tabela de pagini a procesului. Atunci cînd numărul de cadru este obţinut, îl putem folosi pentru a accesa memoria. În plus, se adaugă numărul de pagină şi numărul de cadru în TLB, astfel încît acestea vor fi regăsite rapid la o ulterioară referinţă. Dacă TLLB-ul este deja plin, sistemul de operare trebuie să selecteze o intrare pentru înlocuire. Politica de înlocuire variază de la LRU la aleatoriu. Anumite TLB-uri permit anumite intrări să fie legate permanent, în general pentru codul sistemului de operare.

Unele TLB-uri stochează în fiecare intrare a acestora un identificator de spaţiu de adresă (ASID). Acest identificatoor este asociat unui proces şi utilizat pentru a oferi protecţie a spaţiului de adresă pentru acel proces. Atunci cînd TLB-ul încearcă să rezolve un număr de pagină, se asigură că ASID-ul pentru procesul care rulează este acelaşi cu ASID-ul asociat cu numărul de pagină reţinut. Dacă ASID-urile nu coincid, încercarea este tratată drept TLB miss. În afară de protecţie, ASID-ul permite TLB-ului să reţină intrări al mai multor procese simultan. Dacă TLB-ul nu suportă ASID-uri separate, atunci de fiecare dată cînd o tabelă nouă este selectată (de exemplu, la schimbare de context), TLB-ul trebuie şters pentru a asigura faptul că următorul proces nu utilizează informaţiile de translatare ale unui alt proces.

Eficienţa unui sistem de memorie virtuală depinde de minimizarea numărului lipsurilor de pagină. Deoarece timpul de acces al memoriei secundare este mult mai ridicat decît timpul de acces al memoriei principale, un număr excesiv al lipsurilor de pagină poate încetini sistemul în mod semnificativ. Atunci cînd apare o lipsă de pagină, trebuie identificată o pagină în memoria principală care nu este necesară în momentul respectiv, astfel încît aceasta poate fi scrisă în memoria secundară. Apoi, pagina cerută poate fi încărcată în acest cadru eliberat din memoria principală.

Pentru a măsura eficienţa unui sistem de memorie care utilizează paginarea, se consideră *frecvenţa lispurilor de pagină* Această frecvenţă reprezintă raportul dintre numărul lipsurilor de pagină care apar în timpul execuţiei unui proces şi numărul total de accesuri la memorie, care este suma dintre numărul lipsurilor de pagină şi numărul de accesuri cu succes. Această frecvenţă trebuie să fie cît mai redusă pentru a minimiza numărul accesurilor la disc. Aceasta este afectată de dimensiunea paginii şi de numărul cadrelor de memorie.

Paginarea creşte în mod substanţial timpul de prelucrare necesar unui proces, deoarece vor fi necesare două accesuri la disc, pe lîngă execuţia unui algoritm de înlocuire. Există însă o alternativă care, în unele cazuri, poate reduce numărul acceselor la disc la unul singur. Această reducere este obţinută prin adăugarea unui bit suplimentar la fiecare pagină (**bit de modificare** sau **bit de inconsistenţă**). Dacă o anumită pagină a fost modificată, bitul corespunzător de modificare este setat la 1. Dacă bitul de modificare al unei pagini este 1 şi aceaastă pagină a fost selectată pentru a fi eliminată din memorie, atunci vor fi necesare două accesuri la disc. Dacă bitul de modificare este 0 (aceasta însemnînd că nu au fost modificări ale acestei pagini din momentul în care a fost încărcată ultima dată), nu este necesară scrierea

paginii pe disc. Deoarece starea originală acestei pagini se află pe disc şi nu s-au efectuat modificări ale paginii, aceasta poate fi rescrisă de noua pagină solicitată.

5.5 Paginare ierarhică

Majoritatea sistemelor de calcul moderne pun la dispoziţie un spaţiu de adresă suficient de mare (2^{32} sau 2^{64}). Într-un astfel de mediu, tabela de pagini devine excesiv de mare. De exemplu, pentru un sistem cu adrese logice de 32 de biţi, cu dimensiunea unei pagini de 4KB (2^{12}), se obţine o tabelă de pagini de pînă la 1 milion de intrări ($2^{32}/2^{12}$). Presupunînd că fiecare intrare conţine 4 octeţi, fiecare proces ar avea nevoie de pînă la 4MB de spaţiu de adrese fizice doar pentru tabela de pagini. O simplă soluţie la această problemă este împărţirea tabelei de pagini în bucăţi mai mici.

O soluţie pentru această problemă este reprezentată de algoritmul de paginare pe două nivele, în care tabela de pagini este de asemenea paginată. Spre exemplu, pentru un sistem cu adrese logice de 32 de biţi şi dimensiunea unei pagini de 4KB, în care adresa logică este împărţită în numărul de pagină de 20 de biţi şi deplasamentul de pagină 12 biţi. Întrucît reţinem o pagină în tabela de pagini, numărul de pagină este împărţit într-un număr de pagină de 10 biţi şi un deplasament de pagină de 10 biţi.

5.6 Segmentare

O altă metodă de implementare a memoriei virtuale este numită **segmentare**. În acest caz, un program este împărţit în secţiuni de lungime variabilă numite **segmente**. Un segment poate corespunde unei entităţi logice cum ar fi un set de date sau o funcţie în cadrul unui program. Fiecare proces păstrează o tabelă de segmente în memoria principală, tabelă care conţine în principiu aceleaşi informaţii ca şi tabela de pagini. Însă, spre deosebire de pagini, segmentele au lungimi diferite şi ele pot începe în orice zonă din memorie. De aceea, eliminarea unui segment din memoria principală nu asigură întotdeauna spaţiu suficient pentru un alt segment.

Un segment este un set de cuvinte continue, asociate logic. Un cuvînt dintr-un segment este referit specificînd o adresă de bază, numită **adresă de segment**, şi un deplasament în cadrul segmentului. Un program şi datele sale pot fi considerate ca o colecţie de segmente înlănţuite. Legăturile provin din faptul că un segment de program utilizează sau apelează alte segmente.

Avantajul principal al segmentării este că limitele segmentelor corespund limitelor programului şi ale datelor. În consecinţă, informaţiile care sunt partajate între diferiţi utilizatori sunt organizate adesea în segmente. Din cauza independenţei logice între segmente, un segment de program poate fi modificat şi recompilat în orice moment fără a afecta alte segmente. Anumite proprietăţi necesită ca accesurile la segmente să fie verificate pentru a preveni utilizarea lor neautorizată. Această protecţie este implementată cel mai simplu atunci cînd unităţile de alocare sunt segmente. Anumite tipuri de segmente, cum sunt segmentele de stivă, variază în lungime în timpul execuţiei programului. Segmentarea modifică dimensiunea zonei de memorie asignată unui asemenea segment, utilizînd astfel în mod eficient spaţiul de memorie disponibil. Pe de altă parte, faptul că segmentele pot avea lungimi diferite necesită o metodă de alocare relativ complexă pentru a evita fragmentarea excesivă a memoriei. Această problemă este simplificată prin combinarea segmentării cu paginarea.

Comparînd paginarea şi segmentarea, paginarea necesită un sistem mai simplu de alocare a memoriei decît segmentarea, deoarece paginile au aceeaşi dimensiune. Pe de altă parte, paginile nu au semnificaţie logică, deoarece ele nu reprezintă elemente de program. Paginarea şi segmentarea pot fi comparate de asemenea din punctul de vedere al fragmentării memoriei. La sistemele care utilizează segmentarea, blocurile de dimensiuni diferite tind să prolifereze în memoria principală; ele pot fi eliminate prin pro-

cesul de compactare a memoriei. Existenţa unui spaţiu inutilizabil între zonele ocupate este numită **fragmentare externă**. Deoarece cadrele din pagină sunt coninue, fragmentarea externă nu apare la sistemele care utilizează paginare. Totuşi, dacă un bloc de k cuvinte este împărţit în p pagini de cîte n cuvinte, iar k nu este multiplu de n, ultimul cadru din pagină căruia îi este asignat blocul nu va fi ocupat complet. Existenţa unui spaţiu inutilizabil în interiorul unui cadru de pagină ocupat parţial este numită **fragmentare internă**.

6 Curs 6 - Memoria virtuală

Obiective

- Discutarea beneficiilor sistemului de memorie virtuală.
- Explicarea conceptului de paginare la nevoie, algoritmii de înlocuire a paginilor şi alocarea paginilor.
- Discutarea principiilor modelului set de lucru.

Memoria virtuală permite utilizarea unei memorii cu o dimensiune mult mai mare decît memoria reală fizică. Într-un sistem de memorie virtuală, memoria principală şi cea secundară se prezintă pentru un program al utilizatorului ca o memorie unică, de dimensiuni mari şi adresabilă direct.

Înaintea apariţiei memoriei virtuale, dacă spaţiul de adrese al unui program depăşea dimensiunea memoriei disponibile, programatorul era responsabil pentru împărţirea programului în fragmente mai mici, astfel încît fiecare fragment să poată fi încărcat în memoria principală. Toate aceste fragmente erau păstrate în memoria secundară, fiind încărcate în memoria principală pe măsură ce erau necesare. Acest proces necesita cunoaşterea locului în care fragmentele trebuiau stocate pe disc, cunoaşterea operaţiilor de intrare/ieşire necesare pentru accesul fragmentelor şi păstrarea evidenţei întregului proces de fragmentare. Acesta reprezenta un proces foarte complex, ceea ce complica şi mai mult programarea unui calculator.

Conceptul memoriei virtuale a fost creat în principal pentru a elibera programatorul de această sarcină. Memoria virtuală permite utilizatorului să scrie programe care depăşesc limitele memoriei fizice. De asemenea, memoria virtuală permite multiprogramarea, prin care memoria principală este partajată între mai mulţi utilizatori într-un mod dinamic. În cazul multiprogramării, porţiuni ale mai multor programe sunt plasate în memoria principală în acelaşi timp, iar procesorul îşi împarte timpul de execuţie între aceste programe. Procesorul execută un program pentru o perioadă scurtă de timp (numită o cuantă sau o diviziune de timp), iar apoi comută la un alt program; acest proces continuă pînă cînd fiecare program este terminat.

Atunci cînd se utilizează memoria virtuală, sistemul de memorie este adresat printr-un set V de adrese **logice** sau **virtuale**, fiind numite astfel deoarece ele sunt translatate în adrese ale memoriei fizice şi de aceea nu se referă la acelaşi spaţiu din memoria fizică la diferitele execuţii ale unui program. Un set de adrese fizice sau reale R identifică locaţiile de memorare fizică din fiecare unitate de memorie. Adresele virtuale sunt generate de obicei în timpul compilării şi sunt translatate de procesor în adrese fizice în timpul execuţiei. Un mecanism eficient pentru implementarea translatării adreselor de forma $f : V \to R$, este esenţial pentru un sistem de memorie virtuală.

Mecanismul de memorie virtuală oferă suport pentru distribuirea fişierelor şi zonelor de memorie în cazul mai multor procese, permiţînd acestora accesarea unei pagini mapate în memorie şi prin maparea fişierelor în memorie. Aceste proprietăţi conduc la următoarele beneficii:

- Librăriile de sistem pot fi încărcate o singură dată în memoria principală şi utilizate de mai multe procese. Chhiar dacă fiecare proces consideră că bibliotecile partajate sunt o parte din spaţiul propriu de adresă virtuală, cadrele de memorie în care bibliotecile se regăsesc în memoria principală sunt distribuite tuturor proceselor. Practic, bibliotecile sunt mapate doar cu drepturi de scriere într-un spaţiu în care fiecare proces are posibilitate de acces.
- Similar, mecanismul de memorie virtuală oferă posibilitatea proceselor de a împărţi memorie. Acest mecanism permite unui proces să creeze o regiune de memorie pe care o poate distribui celorlalte

procese. De asemenea paginile de memorie comune cu mai multe procese pot fi specificate în momentul apelului de sistem **fork**, astfel optimizînd timpul de creare al procesului.

6.1 Demand paging

Sistemul de demand paging este similar cu sistemul de paginare cu swapping unde procesele rezidă în memoria secundară. Atunci cînd se doreşte executarea unui proces, acesta este încărcat în memorie. În loc de a încărca întreg procesul în memorie se utilizează un incărcător leneş. Acesta nu va încărca o pagină în memorie pînă în momentul în care acea pagină este accesată.

Atunci cînd un proces este încărcat în memorie, dispatcher-ul alege anumite pagini pe care să le încarce în memorie.Astfel, evită încărcarea în memorie a paginilor ce au o probabilitate mică de execuţie, scazînd astfel timpul de swap şi cantitatea de memorie fizică necesară.

Cu această schemă, este necesar un suport hardware pentru a diferenţia paginile care sunt în memorie şi paginile care se află pe disc. Pentru aceasta se poate utiliza bit-ul de valid/invalid din cadrul unei intrări din TLB. Astfel, atunci cînd bit-ul este valid, pagina asociată se află în memorie. Cînd bit-ul este invalid, pagina nu se afla în memoria principală sau nu se află în spaţiul de adresă logic al procesului. Dacă o pagină are bit-ul invalid şi se află în spaţiul de adresă virtual al procesului, atunci adresa regăsită în TLB reprezintă adresa paginii de pe disc.

Dacă procesul încearcă accesarea unei pagini care are bit-ul de validitate invalid, se va genera un **page fault**. Hardware-ul pentru paginare, în momentul accesării paginii respective, va observa faptul că bit-ul de validitate este invalid, generînd o întrerupere către sistemul de operare. Acea întrerupere reprezintă eşuarea sistemului de operare de a aduce pagina dorită în memorie. Procedura pentru rezolvarea acestui page fault este următoarea:

- Se verifică tabela de pagini a procesului pentru a determina dacă referinţa a fost validă sau nu.
- Dacă referinţa a fost invalidă, se **termină procesul**. Dacă a fost validă, dar pagina nu a fost disponibilă în memorie, se continuă.
- Se caută un cadru de memorie liber.
- Se planifică o operaţie cu memoria secundară pentru citirea respectivei pagini în cadrul nou alocat.
- După ce s-a finalizat citirea din memoria secundară, se modifică tabela de pagini a procesului astfel încît pagina să fie disponibilă.
- Se reîncearcă instrucţiunea care a generat întreruperea. Procesul poate acum accesa pagina.

Se poate de asemenea executa un proces fără a avea nicio pagină mapată în memorie. Atunci cînd sistemul de operare setează pointer-ul instrucţiune la prima instrucţiune din proces, care se află într-o pagină în afara memoriei principale, procesul generează primul page fault. După ce pagina a fost adusă în memorie, procesul se execută şi generează page fault-uri pînă în momentul în care toate paginile au fost aduse în memorie. În acel moment se poate executa fără page fault-uri. Această schemă se numeşte **pure demand paging**: niciodată nu se aduce o pagină în memorie pînă cînd nu este necesară.

Suportul hardware necesar pentru demand paging este la fel ca cel pentru paginare şi swapping:

- Tabelă de pagini. Această tabelă trebuie să poată marca o intrare ca invalidă printr-un bit valid/invalid sau o valoare specială pentru biţii de protecţie.
- Memorie secundară. Această memorie reţine acele pagini care nu sunt prezente în memoria principală.

6.2 Copy-on-Write

Apelul de sistem **fork** crează o copie identică a procesului părinte. Astfel, se copiază şi spaţiul de adresă pentru copil, acesta avînd paginile procesului părinte. Totuşi, considerînd faptul că majoritatea proceselor copil efectuează apelul de sistem **exec** după creare, copierea spaţiului de adresă al părintelui ar putea fi inutilă. Pentru a evita acest lucru, se poate utiliza tehnica **copy-on-write**, care permite părintelui şi copilului să utilizeze aceleaşi pagini. Aceste pagini partajate sunt marcate drept pagini copy-on-write, însemnînd că dacă unul dintre procese scrie într-o pagină partajată, o copie a paginii partajate este creată.

Atunci cînd este determinat faptul că o pagină va fi duplicată folosind copy-on-write, este important de observat locaţia la care noua pagină va fi alocată. Aceste pagini noi sunt în general alocate atunci cînd stack-ul sau heap-ul pentru un proces trebuie să se extindă sau cînd există pagini copy-on-write de administrat. Sistemele de operare în general alocă aceste pagini utilizînd o tehnică cunoscută ssub numele de **zero-fill-on-demand**. Aceste pagini sunt zeroizate înainte de a fi alocate, astfel ştergînd conţinutul precedent.

6.3 Algoritmi de înlocuire a paginilor

6.3.1 Algoritmul optim de înlocuire a paginilor

6.3.2 Nefolosit recent

6.3.3 FIFO

6.3.4 A doua şansă

6.3.5 Clock page

6.3.6 LRU

6.3.7 Working set

6.3.8 WSClock

6.3.9 Concluzii

6.4 Fişiere mapate în memorie

În mod tradiţional, pentru citirea unui fişier din memoria secundară sunt necesare apeluri de sistem precum **open, read, write**, iar astfel fiecare acces în fişier necesită un apel de sistem şi un acces la disc. Alternativ, putem folosi tehnicile memoriei virtuale pentru a trata un fişier drept acces la memorie. Această abordare, cunoscută sub numele de **mapare în memorie**, permite o parte din spaţiul virtual de adresă să fie asociat cu un fişier. Utilizarea acestei abordări conduce la creşteri semnificative de performanţă.

Această tehnică este realizată prin maparea unui bloc de pe disc într-o pagină în memorie. Accesul iniţial în fişier reprezintă un clasic page fault, generat de mecanismul de demand paging. Astfel, dimensiunea unei pagini va fi citită din fişier şi introdusă în memorie. Pe măsură ce se realizează accesuri în memorie, se alocă noi pagini conţinînd datele din fişier. Scrierile şi citirile pot fi abstractizate sub forma accesurilor la memorie, înlăturînd astfel costul apelurilor de sistem necesare pentru operaţiile de **read** şi **write**.

Scrierile într-un fişier mapat în memorie nu sunt neapărat sincrone. Unele sisteme ar putea alege să efectueze scrierile în momentul în care sistemul de operare verifică dacă pagina din memorie a fost

modificată. Atunci cînd fişierul este închis, toate paginile cu date mapate în memorie sunt scrise pe disc şi eliminate din memoria virtuală a procesului.

Sistemul de operare poate permite mai multor procese să mapeze acelaşi fişier în mod concurent, pentru a permite distribuirea datelor. Scrierile efectuate de orice proces modificînd datele din memoria virtuală pot fi văzute de orice proces ce mapează aceeaşi secţiune a fişierului. Astfel, pagina mapată în memoria fizică este valabilă tuturor proceselor, acestea avînd un pointer la pagina respectivă.

6.5 Alocarea memoriei nucleului

7 Curs 7 - Analiza executabilelor şi proceselor

8 Curs 8 - Securitatea memoriei

9 Curs 9 - Fire de execuţie

Obiective

- Introducerea noţiunii de thread.
- Discutarea API-urilor pthread şi win32.
- Examinarea problemelor legate de programarea multithread.

Un thread reprezintă unitatea de bază de utilizare a procesorului. Acesta este reprezentat de un identificator de thread, un contor de program, un set de registre şi o stivă. Secţiunile de cod şi de date, împreună cu resursele sistemului de operare, cum ar fi fişiere deschise şi semnale sunt distribuite tuturor thread-urilor din cadrul aceluiaşi proces. În general, un proces are un singur thread. Dacă acesta diispune de mai mult de un thread, atunci poate realiza mai mult de o sarcină la un moment dat.

Avantajele programării multithread pot fi împărţite în patru categorii:

- **Receptivitate**
- **Distribuirea resurselor**
- **Economie**
- **Scalabilitate**

9.1 Modele multithreading

9.2 Biblioteci de thread-uri

9.3 Probleme de thread-uri

10 Curs 10 - Sincronizare

Obiective

- Introducerea problemei de secţiune critică, a cărei soluţie poate fi utilizată pentru a păstra consistenţa datelor partajate.
- Prezentarea soluţiilor hardware şi software pentru secţiuni critice.
- Explorarea problemelor clasice în sincronizarea proceselor.

10.1 Secţiune critică

10.2 Soluţia lui Peterson

10.3 Sincronizare în hardware

10.4 Semafoare

10.5 Probleme clasice de sincronizare

10.6 Monitoare

10.7 Deadlock-uri

11 Curs 11 - Dispozitive de intrare/ieşire

Obiective

- Descrierea structurii fizice a dispozitivelor de stocare şi efectele pe care aceasta le implică în utilizarea acestora.
- Explicarea caracteristicilor de performanţă a dispozitivelor de stocare.
- Explorarea structurii sistemelor de operare a subsistemului de intrare/ieşire.
- Discutarea principiilor şi complexităţii hardware.
- Explicarea aspectelor de performanţă a subsistemului de intrare ieşire hardware şi software.

11.1 Structura discului

11.2 Planificarea discului

11.3 Structuri RAID

11.4 Hardware pentru intrare/ieşire

11.5 Interfaţa aplicaţie intrare/ieşire

11.6 Subsistemul intrare/ieşire al nucleului

11.7 Stream-uri

11.8 Performanţă

12 Curs 12 - Implementarea sistemelor de fişiere

Obiective

- Descrierea detaliilor de implementare a sistemelor de fişiere şi structurii directoarelor.
- Discutarea algoritmilor şi compromisurilor de alocare şi eliberare de blocuri.

12.1 Structura sistemului de fişiere

12.2 Implementarea sistemului de fişiere

12.3 Implementarea directoarelor

12.4 Metode de alocare

12.5 Metode de eliberare

12.6 Eficienţă şi performanţă

12.7 Recuperare

13 Curs 13 - Networking în sistemul de operare

14 Curs 14 - Analiza performanţei