Laborator #6 : bibliografie suplimentară (1)

Sumar:

[Prezentare informativă despre unitatea de alocare și/sau de transfer a informației și despre cache-urile sistemului de fișiere]

i) [Despre *granularitatea informației* de la diferite nivele dintr-un sistem de calcul]

ii) [Despre *file-system cache*-ul gestionat de sistemul de operare]

iii) [Despre *cache*-urile gestionate de biblioteca standard I/O din C]

Prezentare informativă despre unitatea de alocare și/sau de transfer a informației și despre cache-urile sistemului de fișiere

i) Despre *granularitatea informației* (i.e., unitatea de alocare și/sau de transfer) de la diferite nivele dintr-un sistem de calcul

Hide / Show the details

- ***Granularitatea informației* la nivelul hardware al perifericelor de stocare**:

  Unitatea de alocare a informației (i.e. de stocare) pe un periferic de stocare se numește **sector**; o denumire alternativă utilizată este cea de **bloc fizic de disc**, prescurtată uneori doar prin **bloc de disc**.
  De asemenea, tot sectorul este și unitatea de transfer hardware (prin tehnica DMA, explicată la curs), între periferic și memorie! Cu alte cuvinte, nu se poate transfera hardware doar un singur octet, sau doar o secvență de câțiva octeți, ci se transferă fie exact un sector, fie un multiplu de sectoare (transferate "atomic", printr-un singur transfer hardware).

  Dimensiunea sectorului are o valoare constantă, standardizată pentru toate modelele de periferice de stocare dintr-o anumită categorie!
  Astfel, pentru categoria mediilor de stocare de tipul CD-urilor și DVD-urilor, dimensiunea sectorului era de 2048 octeți.
  Pentru categoria discurilor magnetice clasice (aka HDD-uri), dimensiunea sectorului era de 512 octeți, valoare standardizată folosită de toți producătorii de modele de HDD-uri vreme de cca. 3-4 decenii, până în anul 2005, când a apărut primul model de HDD având dimensiunea sectorului de 4096 octeți. În anii care au urmat, toți producătorii au migrat treptat la noua valoare de 4096 octeți/sector, astfel că în prezent putem considera că dimensiunea standardizată a sectorului este de 4096 octeți, pentru modelele actuale de HDD-uri.
  (*Notă*: migrarea de la HDD-uri cu sectorul de 512 octeți la cele cu sectorul de 4096 octeți nu a fost ușoară, datorită nevoii de păstrare a compatibilității pentru nivelul superior, software, printr-o tehnică de emulare a vechii dimensiuni -- în acest sens, vă recomand să lecturați articolul de aici.)
  Pentru categoria discurilor bazate pe memorie *flash* (aka SDD-uri), acestea reprezintă, încă, o tehnologie "tânără", i.e. în curs de dezvoltare. Adică, nu este încă standardizată, fiecare producător major de SDD-uri folosind propria tehnologie proprietară. Chiar dacă există diferențe între tehnologiile de fabricație a SSD-urilor utilizate de producătorii majori de pe această piață (inclusiv, aceste tehnologii proprietare este posibil să folosească intern unități de alocare de dimensiuni diferite), totuși toți fabricanții au inclus la nivelul firmware-ului de pe SSD-uri, un nivel de emulare prin software a interfeței hardware standardizate de la categoria HDD-urilor,

modalitate adoptată la introducerea acestei noi categorii pentru păstrarea compatibilității cu software-ul existent (atât SO-uri, cât și aplicații specializate care accesează discul în manieră directă). Astfel, practic, SSD-urile sunt "văzute" la nivel software ca și dispozitive de stocare cu unitatea de alocare și stocare (i.e., sectorul) similară cu cea standardizată de la categoria HDD-urilor.

- ***Granularitatea informației*** **la nivelul sistemelor de fișiere**:

  Unitatea de alocare a informației (i.e. de stocare) în cadrul unui fișier se numește ***cluster*** ; uneori, o denumire alternativă utilizată este cea de **bloc logic**. Cu alte cuvinte, spațiul alocat pe disc pentru stocarea conținutului unui fișier este un multiplu de dimensiunea *cluster*-ului (și anume, acel număr reprezentând partea întreagă superioară din raportul dintre dimensiunea în octeți a conținutul propriu-zis stocat în acel fișier și dimensiunea *cluster*-ului).
  Mai precis, structurile de date specifice fiecărui tip de sistem de fișiere (e.g., tabela MFT folosită de sistemul de fișiere NTFS din Windows, sau *inod*-ul, structura de date folosită de sistemele de fișiere native din Linux), utilizate pentru gestiunea fișierelor (și directoarelor) stocate în cadrul acelui sistem de fișiere, folosesc *cluster*-ul ca și unitate de alocare pe disc pentru păstrarea conținuturilor fișierelor stocate în cadrul acelui sistem de fișiere.

  Dimensiunea *cluster*-ului NU are o valoare constantă, standardizată pentru toate modelele de periferice de stocare dintr-o anumită categorie! Ea depinde de tipul de sistem de fișiere în care păstrăm acel fișier, și mai precis depinde de valoarea atribuită de utilizator, în manieră implicită sau explicită (i.e., prin interfața programului/comenzii de formatare specifice SO-ului și acelui tip de sistem de fișiere), la momentul creării acelui sistem de fișiere (prin operațiunea de formatare, cu acel sistem de fișiere, a unei partiții de pe un dispozitiv de stocare).
  În practică, valoarea aceea configurabilă de utilizator la momentul formatării unei partiții, este întotdeauna selectabilă dintr-o listă de multipli **puteri a lui 2** de dimensiunea sectorului, corespunzătoare dispozitivului de stocare pe care se află acea partiție.
  Spre exemplu, dacă la formatarea unei partiții cu un anumit sistem de fișiere, se alege dimensiunea *cluster*-ului să fie de 16384 octeți, și presupunând că partiția respectivă se află pe un dispozitiv de stocare (e.g., un HDD sau un SDD) care are sectorul de dimensiune 4096 octeți, înseamnă că fiecare *cluster* al fiecărui fișier va ocupa pe disc câte 4 sectoare, consecutive (adică, adresele de disc ale celor 4 sectoare sunt numere consecutive).

  Notă: când se citește în memorie de pe disc, respectiv când se scrie din memorie pe disc, o parte din conținutul unui fișier, folosind fie apelurile I/O specifice API-ului acelui SO, fie funcțiile de I/O din biblioteca stdio (sau din alte biblioteci cu rol asemănător), **unitatea de transfer** nu este nici *cluster*-ul, nici secvența exactă de octeți specificată în parametrii apelului I/O folosit. Ci, unitatea de transfer este cea descrisă la categoria care urmează mai jos:

- ***Granularitatea informației*** **la nivelul memoriei și al *file-system cache*-ului din memorie**:

  Pentru SO-urile comerciale moderne, care administrează memoria RAM folosind tehnica de paginare la cerere, bazată pe suportul pentru paginare hardware oferit de arhitectura hardware a sistemului de calcul folosit, unitatea de alocare a informației (i.e. de stocare) în memoria RAM este **pagina** (fizică și virtuală), a cărei dimensiune este o constantă ce depinde de arhitectura hardware.
  Spre exemplu, simplificând discuția de la cursul teoretic, dimensiunea paginii pentru arhitectura x86/x64 este de 4096 octeți/pagină.

  Orice transfer de informații între memoria RAM și un periferic de stocare, transfer comandat de SO (și, prin intermediul SO-ului, de către orice aplicație software), se face la nivel de pagină, nu de octet! Cu alte cuvinte, nu se transferă octeți individuali, ci pagini întregi (cu o pseudo-excepție: când dimensiunea logică a unui fișier nu este multiplu de dimensiunea paginii -- a se vedea detalii mai jos).
  Deci, tot pagina este și unitatea de transfer software (i.e., transfer comandat de SO sau aplicații), între memorie și sistemul de fișiere stocat pe un periferic de stocare!

Comentariu: în practică, această afirmație nu intră în contradicție cu cele spuse mai sus, despre unitatea de transfer la nivel hardware, deoarece dimensiunea paginii este egală cu un multiplu al dimensiunii sectorului, întotdeauna (sau, cel puțin, eu nu am cunoștiință de vreo arhitectură hardware și de vreun tip de periferic de stocare folosit de vreun sistem de calcul bazat pe acea arhitectură hardware, care să încalce această regulă).

Observația #1: principala metodă folosită de aplicații pentru a comanda un transfer software, prin intermediul SO-ului, este, după cum spuneam și mai sus, prin intermediul apelurilor de sistem pentru operații I/O, specifice API-ului acelui SO. Sau, prin intermediul apelării funcțiilor de I/O din biblioteca stdio (sau din alte biblioteci cu rol asemănător), iar implementarea funcțiilor respective de bibliotecă folosește tot apelurile de sistem pentru operații I/O, specifice API-ului acelui SO.
În ambele situații (i.e., fie că folosim un apel de sistem, fie o funcție de bibliotecă), SO-ul optimizează operațiile cu discul utilizând conceptul de *file-system cache*, pe care l-am sumarizat la topicul ii) de mai jos. Astfel, indiferent de lungimea exactă a secvenței de octeți specificată în parametrii apelului I/O folosit de noi în aplicație pentru a accesa o parte din conținutul unui fișier (i.e., fie pentru a citi în memorie informații din fișierul de pe disc, fie pentru a scrie informații din memorie în fișierul de pe disc), **unitatea de transfer** nu este nici *cluster*-ul, nici secvența exactă de octeți specificată în parametrii apelului I/O folosit, ci este pagina, transferul efectiv între RAM și disc a paginilor ce conțin acea informație fiind gestionat de administratorul de memorie al SO-ului, care are și sarcina gestiunii *file-system cache*-ului din memorie.
Spre exemplu, să presupunem că în programul nostru facem un apel de citire care comandă citirea a 100 octeți, începând de la offset-ul (poziția) 10 dintr-un anumit fișier. Practic, SO-ul va determina *cluster*-ul acelui fișier care stochează primii 4096 de octeți ai conținutului fișierului respectiv, apoi va determina sectorul de disc (sau sectoarele de disc consecutive) ce stochează pe disc acel *cluster*, iar apoi va citi acel sector/acele sectoare de pe disc și le va salva într-o pagină fizică alocată pentru *file-system cache* (sau în mai multe pagini, dacă se folosește, de exemplu, și tehnica de *read-ahead*). În situația în care acel sector/acele sectoare este/sunt deja în *cache*, atunci nu se mai citește efectiv de pe disc. În ambele cazuri (i.e., *cache hit* vs. *cache miss*), se face apoi copierea intra-memorie, din *file-system cache* în *user-space*, exact a informației solicitate în apelul de citire făcut, conform celor explicate în topicul ii) de mai jos.
Și acum, am ajuns la explicarea pseudo-excepției amintite mai sus: să presupunem că fișierul din care vrem să citim cei 100 de octeți începând de la poziția 10, are lungimea logică de doar 4000 de octeți (adică conținutul propriu-zis al fișierului are numai 4000 de octeți). Ce se întâmplă în acest caz este următorul lucru: la fel ca mai sus, SO-ul va comanda transferul complet a unui sector (sau mai multe, consecutive), determinate ca mai sus, pentru 4096 octeți, dar după terminarea transferului de pe disc în pagina corespunzătoare din *file-system cache*-ul din memorie, va "inițializa" cu zero ultimii 96 de octeți din acea pagină, înainte de a continua apoi cu copierea intra-memorie, descrisă mai sus.
Iar dacă în loc de citire dintr-un fișier, vrem să facem o scriere într-un fișier, lucrurile decurg simetric, în sens invers, precum am explicat și în topicul ii) de mai jos, adaptând în mod corespunzător explicațiile suplimentare date aici pentru citire.

Observația #2: pe lângă interfața de tip I/O amintită la Observația #1 de mai sus, SO-urile moderne pun la dispoziția programatorilor de aplicații și o a doua interfață de acces la fișiere, o tehnică denumită "fișiere *mapate* în memorie", despre care vă voi vorbi mai în detaliu la lecția practică și laboratorul din săptămâna a 9-a.
Pe scurt, ideea în cazul acestei tehnici este că un fișier poate fi copiat în memorie, unitatea de alocare în memorie și de transfer fiind tot pagina. Și în acest caz, la copierea în memorie, dacă lungimea fișierului nu este multiplu de dimensiunea paginii, se face practic o copiere "incompletă" în sensul explicat mai sus (i.e., inițializarea cu zero-uri a restului din pagină, după poziția corespunzătoare EOF-ului din fișier). După copierea în memorie, programul accesează adresele de memorie din pagina (sau paginile) în care a fost mapat fișierul de pe disc în memorie, făcând citiri și scrieri după dorință la acele adrese de memorie. Iar modificările făcute în memorie se pot "salva" înapoi pe disc, după dorința programatorului.
Mai mult, zona de memorie reprezentată de pagina (sau paginile) în care a fost mapat fișierul de pe disc, poate fi și partajată între două sau mai multe procese (secvențiale, sau *multi-threaded*), pur și simplu prin maparea aceluiași fișier, în fiecare dintre cele două sau mai multe procese.

Consecință: astfel, veți putea experimenta practic șabloanele de cooperare și de sincronizare bazate pe memorie partajată prezentate la cursurile teoretice, folosind procese secvențiale care partajează memorie prin intermediul unui fișier mapat în memorie!

Notă: mai multe detalii despre aceste lucruri veți învăța ulterior, la cursurile teoretice despre "Administrarea memoriei", despre "Administrarea informațiilor -- sisteme de fișiere" și despre "Administrarea perifericelor de stocare".

---

ii) Despre *file-system cache*-ul gestionat de sistemul de operare

Hide / Show the details

1) **Conceptul de *file-system cache* din cadrul unui SO:**

La nivelul componentei de gestiune a sistemelor de fișiere din cadrul nucleului unui SO, se folosește o zonă de memorie internă din *kernel-space* ce implementează un *cache* pentru operațiile cu discul, i.e. se păstrează în memoria RAM conținutul celor mai recent accesate blocuri de disc.
Acest *cache* este denumit ***file-system cache*** în literatura de specialitate, iar el funcționează după aceleași reguli generale ale *cache*-urilor de orice fel: i) citiri repetate ale aceluiași bloc de disc, la intervale de timp foarte scurte, vor regăsi informația direct din *cache*-ul din memorie; ii) scrieri repetate ale aceluiași bloc de disc, la intervale de timp foarte scurte, vor actualiza informația direct în *cache*-ul din memorie, iar pe disc informația va fi actualizată o singură dată, la momentul operației de *cache-flushing*; iii) operațiile de invalidare/actualizare a informației din *cache*: ... ; ș.a.

Granularitatea acestui *cache* (i.e., **unitatea de alocare** în *cache*) nu este octetul, ci pagina (i.e., unitatea de administrare a memoriei virtuale), care are o dimensiune dependentă de arhitectura hardware (e.g., pentru arhitectura x86/x64 dimensiunea paginii este de 4096 octeți). Cu alte cuvinte, operațiile efective de I/O prin DMA între memorie și disc transferă blocuri de informație cu această dimensiune!

Acest *cache* este unic per sistem, i.e. există o singură instanță a sa, gestionată de SO și utilizată simultan (ca și "resursă partajată") de toate procesele ce se execută în sistem (cu unele excepții, procese care fac operații I/O conform celor descrise în cele două observații numerotate, de la finalul acestui topic).

Notă: mai multe detalii despre aceste lucruri veți afla într-un curs teoretic ulterior.

2) **Implicațiile existenței acestui *file-system cache* pentru programarea aplicațiilor folosind API-ul POSIX**:

Implicit, apelul `read()` implementează o citire realizată astfel: (1) doar dacă informația solicitată nu este deja în *file-system cache*, se execută un transfer efectiv, prin DMA, dinspre disc înspre pagina de memorie alocată în *file-system cache* pentru a stoca acea informație, folosind ca unitate de transfer pagina; (2) se copie doar informația solicitată dinspre *kernel-space* (i.e., pagina din *file-system cache* alocată pentru blocul respectiv de disc) înspre *user-space* (i.e., variabila din programul dvs. a cărei adresă o specificați în apelul respectiv, ca argumentul al doilea).

Implicit, apelul `write()` implementează o scriere realizată astfel: (1) se copie informația solicitată dinspre *user-space* (i.e., variabila din programul dvs. a cărei adresă o specificați în apelul respectiv, ca argumentul al doilea) înspre *kernel-space* (i.e., în poziția ei din pagina din *file-system cache* alocată pentru blocul respectiv de disc); (2) nu imediat, ci ulterior, la un moment de timp stabilit prin politica folosită pentru *flushing*-ul pe disc al actualizărilor din *cache*, se va executa un transfer efectiv, prin DMA, dinspre pagina de memorie alocată în *file-system cache* pentru acea informație, înspre disc, folosind ca unitate de transfer întreaga pagină.

Consecință: astfel se obtine un spor de performanță, prin faptul că mai multe operații de citiri și/sau scrieri succesive (sau, la momente de timp "foarte apropiate" între ele) din/în același bloc de disc, vor "economisi" transferuri efective cu discul, și doar copierile intra-memorie (i.e., între *user-space* și *kernel-space*) se vor executa pentru fiecare dintre acele citiri și/sau scrieri !!! Iar de această optimizare este responsabil SO-ul, programatorul care scrie o aplicație nu are nimic special de făcut pentru a beneficia de acest tip de optimizare a acceselor la disc (exceptând faptul că, totuși, programatorul poate influența modul de gestiune implicită a *file-system cache*-ului utilizat de către SO -- în acest sens, a se vedea cele două observații de mai jos).

Notă: pe de altă parte, se mai poate îmbunătăți performanța la execuție și prin minimizarea numărului de apeluri de sistem făcute pentru a citi sau scrie o pagină! Cu alte cuvinte, un singur apel `read/write(fd, &page, 4096);` este mai eficient, ca timp de execuție, decât bucla echivalentă: `for(int offset = 0; offset < 4096; offset++) read/write(fd, &page+offset, 1);` . Iar pentru acest tip de optimizare este direct responsabil programatorul care scrie o aplicație, SO-ul nu poate "interveni" pentru a "rescrie" programul executabil în sensul realizării unei astfel de optimizări.

Observația #1: comportamentul implicit descris mai sus s-ar putea dovedi ineficient pentru unele aplicații cu cerințe speciale (e.g., programe care accesează discul după anumite "tipare" neobișnuite de accese la disc, ce provoacă o rată mică de "succes în *cache*"), situație în care **dubla copiere** impusă de comportamentul implicit (i.e., o copiere între disc și memoria nucleului, și o altă copiere între memoria nucleului și memoria aplicației) s-ar putea să le afecteze performanța la execuție.
Pentru astfel de aplicații, API-ul POSIX pune la dispoziție flagul `O_DIRECT` ce poate fi specificat în apelul `open()` pentru a "scurtcircuita" dubla copiere impusă de comportamenul implicit al operațiilor `read()` și `write()`. Practic, nu se mai folosește *cache*-ul, ci fiecare apel `read()` sau `write()` va provoca un transfer efectiv, prin DMA, între disc și *user-space* (i.e., variabila din programul dvs. a cărei adresă o specificați în apelul respectiv, ca argumentul al doilea).
Folosirea acestui flag impune anumite restricții de *aligment* a adresei și dimensiunii variabilei respective, precum și anumite particularități legate de comportamentul său, astfel că programarea cu acest stil de I/O directe nu este ușoară. Pentru mai multe detalii, citiți despre flagul `O_DIRECT` în pagina de manual `man 2 open`.

Observația #2: o altă manieră de modificare a comportamentului implicit descris mai sus este folosirea flagului `O_SYNC`, ce poate fi specificat în apelul `open()` pentru a lucra cu apeluri de scriere "full-sincrone", i.e. un apel `write()` de acest tip nu va returna decât după ce informația scrisă a ajuns efectiv pe disc. Practic, pentru descriptorul respectiv, se modifică comportamentul implicit al *file-system cache*-ului legat de politica folosită pentru *flushing*-ul pe disc al actualizărilor din *cache*, în sensul că se va comporta ca un *cache* de tipul "write-through", pentru descriptorul respectiv.

Ca o remarcă finală pe marginea acestui subiect, SO-ul mai folosește și alte tehnici de optimizare a accesului la disc prin intermediul *file-system cache*-ului gestionat de el, precum ar fi tehnica de *read-ahead* și tehnica de *write-behind*. Despre aceste tehnici vă voi vorbi într-un curs teoretic ulterior.

---

iii) Despre *cache*-urile gestionate de biblioteca standard I/O din C

Hide / Show the details

Conceptul de *cache* gestionat de biblioteca stdio:

După cum spuneam în lecția practică despre lucrul cu fișiere, funcțiile din biblioteca standard I/O din C lucrează *buffer*izat, adică folosesc un *cache* pentru disc implementat la nivelul bibliotecii stdio, adică "deasupra" *file-system cache*-ului gestionat la nivelul nucleului SO-ului, prezentat la topicul ii) de mai sus.
Cu alte cuvinte, acesta este un *cache* al informațiilor din *file-system cache*, care la rândul său este un *cache* al informațiilor de pe disc.

În plus, acest *cache* gestionat de biblioteca stdio, este implementat în *user-space* (la fel ca și toate funcțiile bibliotecii), ceea ce înseamnă că este unic per proces și nu per sistem, adică nu există un singur *cache* al bibliotecii care să fie partajat de toate procesele ce utilizează apeluri ale bibliotecii.
În concluzie, rețineți faptul că acest *cache* gestionat de biblioteca stdio nu este unic per sistem, ca în cazul *file-system cache*-ului gestionat de SO, de aceea am folosit pluralul în titlul acestui topic.

Analogie: putem face o paralelă cu situația *cache*-urilor prezente într-un procesor x86/x64 multicore cu doar 2 nivele de cache (nu cu 3 nivele, cum sunt modelele mai performante). Și anume: *cache*-ul gestionat de biblioteca stdio este analogul *cache*-urilor L1 de pe procesor (există câte unul pentru fiecare core), *file-system cache*-ul gestionat de SO este analogul *cache*-ului L2 de pe procesor (există unul singur, partajat de toate core-urile), iar discul este analogul memoriei RAM, i.e. este "depozitul" principal de informație, peste care s-au construit cele două nivele de *cache*-uri.

***Observație finală*:** pentru a putea scrie programe de aplicație nu doar corecte, ci și **eficiente/performante** (!), este necesar să cunoașteți astfel de detalii legate de modul de execuție a apelurilor I/O și de optimizări folosite atât la nivelul SO-ului, cât și la nivelele superioare ale bibliotecilor și framework-urilor pe care le apelați în programele pe care le scrieți. Și, bineînțeles, nu doar să le cunoașteți, ci și să le aplicați în mod adecvat în programele pe care le scrieți!
