Alocarea memoriei în nucleul sistemului de operare

Mihai Budiu — mihaib@cs.cmu.edu
`http://www.cs.cmu.edu/~mihaib/`

17 decembrie 1998

Subiect: alocarea memoriei: feluriţi algoritmi

Cunoştinţe necesare: programare în C, structuri de date

Cuvinte cheie: alocator, memorie, colectare de deşeuri, performanţă, nucleu

Cuprins

"640 de kilooctеţi trebuie să fie îndeajuns pentru oricine". Aceasta este probabil cea mai faimoasă gafă a lui Bill Gates, pronunţată cîndva pe la începutul anilor '80. Enunţul este cu atît mai amuzant cu cît compania lui Gates, Microsoft, produce nişte programe uriaşe, pentru care resursele calculatoarelor contemporane, care au evoluat într-o rată greu de imaginat, nu sunt suficiente.

Dacă lucrurile ar fi stat aşa, atunci articolul de faţă nu ar mai fi fost scris. Dar rata de creştere a programelor este comparabilă cu cea a capacităţilor memoriilor; pe de altă parte *datele* prelucrate de programe devin din ce în ce mai mari; audio, video, multimedia sunt mari consumatoare de spaţiu.

Dacă lucrurile ar fi stat aşa cum prevedea Gates, atunci sistemele de operare însele ar fi dispărut, sau ar fi arătat substanţial diferit de cele din ziua de azi; aceasta este tot o ironie, pentru că Microsoft îşi menţine dominaţia în arena mondială a software-ului prin monopolul pe care îl are asupra sistemelor de operare pentru calculatoare personale. De ce spun că sistemele de operare ar fi dispărut? Pentru că rolul principal al unui sistem de operare este cel de management al resurselor unui calculator, în particular de management al memoriei. Dacă resursele ar fi prezente din abundenţă, atunci necesitatea unui manager ar fi mult mai puţin stringentă.

1 Alocarea memoriei

Ce este de fapt alocarea memoriei? Calculatorul posedă din fabricaţie o anumită cantitate de memorie (RAM). În memorie vor fi încărcate mai multe programe şi datele prelucrate de ele: nucleul sistemului de operare, datele acestuia, programele utilizatorilor şi datele asupra cărora acestea operează, datele citite de la dispozitivele periferice, pachetele de date care vin şi merg în reţeaua în care calculatorul este conectat, bibliotecile încărcate dinamic, etc. O singură bucată mare de memorie (RAM-ul) trebuie împărţită între toate aceste entităţi lacome, în aşa fel încît să nu se incomodeze unele pe altele. Entitatea care gestionează memoria, ţine contabilitatea zonelor ocupate şi a celor libere, care satisface cererile pentru noi zone şi care re-utilizează zonele eliberate este alocatorul de memorie.

Alocarea memoriei este de obicei o treabă ierarhică; la baza ierarhiei se află sistemul de operare, care are la dispoziţie întregul RAM. Sistemul de operare dă feluritelor programe ale utilizatorilor porţiuni de memorie. La rîndul lor, fiecare din programe gestionează bucăţica primită de la nucleu pentru nevoile sale interne.

În acest text ne vom concentra asupra alocatorului de memorie din nucleele sistemelor de operare de tip Unix, dar vom privi superficial şi asupra unor alte alocatoare. Un tratament excelent al subiectului puteţi găsi în capitolul 12 din cartea "Unix Internals", de Uresh Vahalia, publicată în anul 1996 de editura Prentice Hall. Am folosit unele dintre prezentările din acea carte în scrierea acestui articol.

2 Limbaje şi alocarea de memorie

O clasificare a limbajelor din punctul de vedere al alocării memoriei le împarte în trei categorii:

1. Limbaje care nu pot aloca dinamic memorie. Din această categorie fac parte cele mai ancestrale limbaje: Cobol, Fortran. În aceste limbaje (cel puţin în versiunile lor iniţiale), utilizatorul nu poate aloca dinamic memorie de loc în momentul execuţiei programului; toată memoria necesară trebuie să fie alocată înainte ca programul să pornească în execuţie.

2. Limbaje cu alocare şi dealocare explicită. Limbaje ca Pascal, C şi C++ îi permit utilizatorului să ceară pe parcursul execuţiei noi zone de memorie şi să returneze memoria folosită. Utilizatorul apelează pentru acest scop nişte *funcţii de bibliotecă*. Aceste funcţii au fost implementate de cel care a scris compilatorul pentru limbajul respectiv. Aceste funcţii cer de la sistemul de operare o bucată mare de memorie pe care apoi o împart după necesităţi; atunci cînd toată bucata este consumată cer o alta de la nucleu. În Pascal funcţiile cu pricina sunt `new` şi `free`, în C `malloc` şi `free` iar în C++ `new` şi `delete`. Ca funcţionare sunt extrem de similare; funcţiile din Pascal şi C++ folosesc tipul obiectelor alocate pentru a deduce cîtă memorie este necesară (de exemplu programatorul zice: "vreau memorie pentru un vector de 10 întregi"); programatorii în C trebuie să indice explicit de cîtă memorie au nevoie (ex.: "dă-mi şi mie 40 de octeţi").

3. Limbaje cu colectoare de gunoaie (*garbage collection*). Lisp şi Java folosesc un mecanism extrem de interesant, prin care utilizatorul nu specifică niciodată cînd vrea să elibereze o zonă de memorie (adică `free()` nu există); compilatorul şi un sistem de funcţii care se execută simultan cu programul (runtime system) deduc singure care dintre zone sunt nenecesare şi le recuperează. Lisp-ul aparent este un limbaj în care nu există nici măcar alocare dinamică (o funcţie de gen `new`); în realitate în Lisp fiecare obiect nou creat este automat alocat într-o zonă de memorie nouă, fără ca utilizatorul să trebuiască să specifice asta (de exemplu cînd utilizatorul concatenează două liste, atunci sistemul alocă automat spaţiu pentru lista rezultat).

   Din anumite puncte de vedere, tehnica colectării de gunoaie este cea mai preferabilă. Principalul ei avantaj este că scuteşte utilizatorul de pericolul de a folosi zone de memorie nealocate, prevenind astfel apariţia unor bug-uri extrem de greu de depanat. Avantajele ei nu se opresc aici: împreună cu o disciplină de tipuri strictă, colectarea deşeurilor face demonstrarea automată a corectitudinii programelor o sarcină mult mai simplă: un demonstrator de teoreme va fi întotdeauna sigur că o zonă de memorie folosită nu este dealocată.

   Pe de altă parte, colectarea de gunoaie are anumite dezavantaje: este impredictibilă ca timp consumat (adică nu e clar în ce moment al execuţiei programului se va petrece), şi este *conservativă*. Întrebarea dacă o anumită zonă de memorie va mai fi sau nu folosită de un program în viitor este în general o chestiune *nedecidabilă*; asta înseamnă că nu se poate scrie nici un algoritm care să răspundă la o astfel de întrebare, chiar dacă are informaţii complete despre programul analizat şi despre datele lui de intrare. Din cauza aceasta este posibil ca un program cu colectare automată să păstreze alocate zone de memorie care sunt în realitate inutile, pentru că sistemul nu are cum să demonstreze acest lucru.

În acest articol vom vorbi mai ales despre sisteme de tipul intermediar, cu alocare şi dealocare explicită. Motivele sunt multiple. În primul rînd majoritatea alocatoarelor din nucleele sistemelor de operare comerciale sunt de acest tip[1]. În al doilea rînd, chiar implementarea unui alocator cu colector va folosi idei de genul celor prezente în alocatoarele explicite. Şi în al treilea rînd, colectarea gunoaielor este un subiect ceva mai dificil.

2.1 Alocarea în limbajul C

Nucleele sistemelor de operare comerciale sunt toate scrise în C, inclusiv alocatoarele de memorie din biblioteci; din cauza aceasta vom privi mai îndeaproape funcţiile acestui limbaj.

Programatorul în C are la dispoziţie funcţiile `malloc` şi prietenii ei (`calloc, realloc`) pentru a aloca memorie, şi funcţia `free` pentru a o elibera. `calloc` şi `realloc` pot fi scrise folosind `malloc`, aşa că nu le vom da prea mare atenţie. Prototipurile acestor funcţii se află în fişierul header `<stdlib.h>` din biblioteca standard C, prezentă pe orice sistem.

Funcţia `malloc` are un singur argument: numărul de octeţi care trebuie alocaţi. Rezultatul funcţiei este un pointer generic (`void*`) spre zona alocată. Dacă `malloc` nu găseşte destulă memorie, atunci rezultatul apelului ei este pointerul cu valoarea 0 (`NULL`).

Funcţia `free()` are tot un singur argument, şi nici un rezultat. Argumentul ei este un pointer obţinut de la un `malloc` anterior; efectul executării ei este eliberarea zonei de memorie aflată în acel loc.

Biblioteca în care sunt implementate `malloc` şi `free` menţine o pleiadă de structuri de date indicînd unde se află zonele libere şi unde cele ocupate. Observaţi de pildă că `free` nu primeşte nici un fel de informaţii despre cît de mare este zona dealocată; biblioteca trebuie deci să ştie acest lucru.

În cazul sistemului de operare Unix aceste funcţii de bibliotecă sunt implementate folosind un apel de sistem[2] numit `sbrk`. Figura 1 ilustrează relaţia dintre aceste funcţii. Apelul de sistem are la ca argument cantitatea cerută de memorie (pozitivă sau negativă); efectul executării apelului este adăugarea *la sfîrşitul segmentului de date* al programului a unei cantităţi de memorie egale cu cea cerută.

În mod normal `malloc` execută un `sbrk` la început pentru a obţine o cantitate mare de memorie, şi apoi la fiecare apel returnează utilizatorului cîte o bucăţică din bucata mare. Dacă după o vreme întreaga bucată mare devine ocupată, atunci `malloc` execută un nou `sbrk`.

3 Caracteristici ale alocatoarelor

Există o sumedenie de feluri de alocatoare diferite; în ceea ce priveşte colectoarele şi în ziua de azi se desfăşoară cercetări care îmbunătăţesc performanţa. Iată aici cîteva criterii după care putem evalua calitatea unui alocator.

[1]Cu toate acestea, sistemele Mach şi Digital Unix (fost OSF/1), care este bazat pe Mach, au o formă primitivă de colector chiar în interiorul nucleului.

[2]Apelurile de sistem sunt funcţii puse la dispoziţia programelor de către nucleul sistemului de operare.

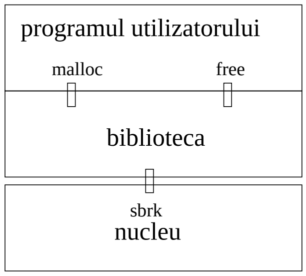

Figura 1: Funcţiile pentru alocarea memoriei. `malloc/free` sunt funcţii de bibliotecă folosite de utilizator. Ele obţin memorie de la nucleul sistemului de operare folosind funcţia `sbrk`.

3.1 Timp pe operaţie (cazul cel mai defavorabil)

Prima caracteristică este complexitatea unei operaţii de alocare/dealocare. În mod normal un alocator trebuie să execute un număr constant de operaţii pentru fiecare funcţiune. Dar vom vedea mai jos că adesea în cel mai rău caz alocatoarele trebuie să execute o sumedenie de instrucţiuni pentru a răspunde la o singură cerere. De exemplu alocatorul cu hartă de resurse, prezentat mai jos, ar putea să aibă nevoie să parcurgă întreaga listă de blocuri de memorie neocupate pentru a găsi unul de măsura potrivită. Asta înseamnă că, cu cît e mai multă memorie disponibilă, cu atît execuţia unui apel de alocare/dealocare poate fi mai costisitoare (pentru că lista va fi mai lungă).

3.2 Timp amortizat

O măsură interesantă a costului unei alocări este *timpul amortizat* pentru o operaţie de alocare. De exemplu dacă cele mai multe operaţii de alocare durează 1ms, dar la fiecare 500 de operaţii costul execuţiei este de 200ms, atunci costul amortizat este de $(499*1 + 1*200)/500 = 1.39$ms pe operaţie (costul în cazul cel mai rău ar fi 200).

Sistemele cu colectare de gunoaie au de obicei un timp amortizat mic pentru fiecare operaţie, dar din cînd în cînd (atunci cînd colectorul porneşte în execuţie), anumite operaţii durează foarte mult. Acesta este cazul sistemelor LISP obişnuite.

Pentru că nucleul unui sistem de operare trebuie să facă faţă unor cereri extrem de severe, în general alocatoarele din nuclee trebuie să aibă cazul cel mai defavorabil relativ mic. Exista alocatoare care primesc nişte indicaţii despre urgenţa operaţiei: dacă cel care are nevoie de memorie nu se grăbeşte, algoritmul îşi poate permite să petreacă mai mult timp; altfel trebuie să răspundă cît se poate de repede.

De exemplu nucleul trebuie să primească pachete de date din reţea; atunci cînd un pachet îşi face apariţia, nucleul nu poate să aştepte prea mult pentru a găsi un loc în care să-l depoziteze, pentru că nu are unde să pună pachetul între timp. Dacă nucleul nu reuşeşte să obţină un buffer pentru o stoca pachetul, atunci de obicei pachetul este pur şi simplu pierdut (asta se numeşte *buffer overrun*). Cel care a transmis pachetul iniţial va trebui să trimită o copie.

3.3 Fragmentare

Alocatoarele se mai confruntă cu o problemă neplăcută: aproape niciodată nu pot folosi întreaga memorie disponibilă, pentru că mici fragmente "se pierd" fără a putea fi folosite.

Aceste pierderi apar din cauză că consumatorii de memorie doresc bucăţi de memorie de mărimi variate, care sunt greu de pus cap la cap în memoria disponibilă. În funcţie de interfaţa alocatorului, există două tipuri de pierderi. Pentru că pierderile apar din cauza împărţirii spaţiului disponibil în fragmente, această problemă se numeşte *fragmentare*. Există două feluri de fragmentare: internă şi externă.

3.3.1 Fragmentarea externă

Tabela 1 arată un posibil scenariu de cereri de alocare/dealocare şi evoluţia memoriei în timp.

timp	cerere	starea memoriei				
		0	10	15	25	35
0		liber				
1	x = aloca(10)	XXXXXXXXX				
2	y = aloca(5)	XXXXXXXXX	XXXXX			
3	z = aloca(10)	XXXXXXXXX	XXXXX	XXXXXXXXXX		
4	dealoca(y)	XXXXXXXXX		XXXXXXXXXX		
5	w = aloca(10)	XXXXXXXXX		XXXXXXXXXX	XXXXXXXXX	

Tabelul 1: Fragmentarea externă apare datorită găurilor apărute între zone de memorie şi care sunt prea mici pentru a fi folosite.

Fenomenul se petrece la momentul 4: dealocăm un bloc de memorie de 5 octeţi cuprins între alte două blocuri. După aceea alocăm unul de zece; cei cinci octeţi nu sunt suficienţi, aşa că în acel loc rămîne o "gaură". Astfel de găuri apar din ce în ce mai multe, şi scad în dimensiune (de exemplu dacă am aloca apoi 3 octeţi în gaura de 5 am rămîne cu o gaură de 2 octeţi). După o vreme găurile practic nu mai pot fi folosite, dar consumă în total o sumedenie de spaţiu. Acest gen de fragmentare se numeşte "extern" pentru că risipa apare între blocurile alocate.

Se demonstrează că atunci cînd avem de-a face cu alocări de blocuri de mărimi diferite, fragmentarea externă este inevitabilă. Singurul mod în care putem repara ceva este de a muta blocuri dintr-o parte într-alta, *compactînd* spaţiul liber. Dar aceasta este o operaţie extrem de costisitoare ca timp, care în anumite condiţii este imposibilă (de exemplu atunci cînd muţi ceva din memorie trebuie să corectezi valorile tuturor pointerilor din program care punctează la acel obiect; pentru un limbaj ca C acest lucru este imposibil). LISP şi Java folosesc uneori compactarea spaţiului ocupat.

Tot compactare fac şi programele care defragmentează discurile pentru a creşte performanţa accesului la fişiere.

3.3.2 Fragmentarea internă

Acest tip de alocare este folosit adesea pentru alocarea spaţiului pe discurile magnetice, şi pentru alocarea memoriei virtuale pentru procese.

Singurul mod în care putem evita fragmentarea externă este de a aloca toate obiectele de exact aceeaşi mărime. În acest caz orice "gaură" între două obiecte poate fi folosită pentru un alt obiect. Cînd consumatorul vrea 10 sau 50 de octeţi, tot 100 îi dai (de exemplu). Asta duce la risipă în interiorul fiecărui "bloc" alocat; aceasta este fragmentarea internă.

Structurile de date şi algoritmii pentru alocarea blocurilor de mărimi egale sunt mult mai simple şi omogene (se pot chiar implementa parţial în hardware). Din cauza asta practic toate sistemele de operare moderne alocă memoria virtuală în termeni de *pagini*, care au în jur de 4 kiloocteţi fiecare.

3.4 Grad de utilizare

Pe lîngă faptul că spaţiul se fragmentează, alocatorul însuşi menţine propriile structuri de date pentru gestiune. Aceste structuri ocupă loc adiţional, reducînd utilizarea efectivă a memoriei. De pildă, nucleele au o tabelă a paginilor din RAM, în care memorează cum este utilizată fiecare pagină. Pentru fiecare pagină din RAM nucleul menţine la Pentium 4 octeţi suplimentari; asta înseamnă o risipă de 1 la mie.

3.5 Simplitate

Interfaţa `malloc/free` este foarte simplă; aproape că nu se poate concepe ceva mai simplu. Alte alocatoare pot avea interacţiuni mai complicate: de exemplu ar putea permite dealocarea doar a unui fragment dintr-o zonă alocată anterior, ar putea cere ca `free` să indice explicit cîtă memorie trebuie dealocată, ar putea avea tot felul de argumente suplimentare referitoare la urgenţa alocării, etc.

3.6 Concurenţă

În fine, o ultimă dimensiune după care putem compara alocatoarele de memorie este gradul de acces concurent pe care îl permit. Pe un calculator cu mai multe procesoare este imaginabil că două procesoare vor simultan să aloce/dealoce memorie. Pentru a preveni haosul, trebuie puse anumite restricţii asupra ordinii de execuţie. Zona de cod care alocă/dealocă este de obicei o *regiune critică*, în sensul că nu poate fi executată de mai multe procesoare simultan[3]. Un alocator bine scris va permite un grad mai ridicat de concurenţă, pentru a exploata mai bine resursele computaţionale.

4 Alocarea statică

Începînd cu această secţiune, voi trece în revistă felurite tipuri de alocatoare folosite în practică. În mijlocul acestei discuţii voi insera o secţiune despre constrîngerile puse asupra alocatoarelor. Discuţia este departe de a fi exhaustivă; este mai curînd ilustrativă.

Să începem cu cel mai simplu tip de alocator, cel care...nu există. În acest caz trebuie să alocăm toate zonele de la scrierea programului. Metoda aceasta era folosită în versiunile vechi de Unix, şi este în continuare folosită pentru anumite sisteme de fişiere (sistemul de fişiere din Unix, UFS, prealocă pentru fiecare fişier cîte un inod pe disc la momentul formatării discului; numărul acesta nu se poate apoi schimba). În Pascal matricile pot fi alocate numai static (adică dimensiunile lor trebuie să fie cunoscute la compilare).

Marele avantaj al metodei este simplitatea. Există însă două mari dezavantaje:

[3] Explicaţii mai ample despre conceptul de regiune critică am dat într-o serie de articole mai vechi din PC Report despre sistemele de operare; copiile articolelor sunt disponibile din pagina mea de web.

- Risipa. Pentru că nu ştim dinainte cîtă memorie o să folosim, trebuie să alocăm în mod conservativ mai multă decît orice valoare posibilă. Asta înseamnă risipă pentru majoritatea cazurilor.
- Depăşirea. Chiar dacă alocăm un milion de obiecte, programul care are nevoie de un milion unu o să eşueze.

5 Consumatori de memorie

Fiecare consumator de memorie are idiosincraziile lui; contextul în care memoria este folosită impune anumite constrîngeri asupra implementării alocatorului.

5.1 Programele utilizatorilor

Cel mai comun consumator de memorie sunt programele utilizatorilor. Acestea pun şi cele mai puţine restricţii asupra alocatorului; cele mai multe programe ale utilizatorilor nu au constrîngeri severe de viteză sau spaţiu, şi ca atare pot implementa scheme foarte complicate, cu cost mediu mai ridicat dar care au alte trăsături dezirabile.

5.1.1 Nişte recomandări

Înainte de a trece la celelalte tipuri de alocatoare, permiteţi-mi să fac nişte recomandări de stil în ceea ce priveşte alocarea memoriei. Recomandările sunt inspirate de indicaţiile proiectului GNU, care produce software de foarte bună calitate (despre proiectul GNU am scris pe larg într-un articol din PC Report consacrat fenomenului Free Software).

- Verificaţi întotdeauna rezultatul întors de funcţiile gen `malloc`. O eroare tipică este de a nu compara rezultatul cu 0. Cel mai recomandabil este să folosiţi o funcţie-"înveliş" (*wrapper*); numele tradiţional al unui wrapper pentru `malloc` este `xmalloc`, de la "checked malloc":

```
/* xmalloc este un wrapper pentru malloc care verifica
   rezultatele si opreste executia programului in caz
   de terminare a memoriei disponibile */

void * xmalloc(size_t size)
{
        void * r = malloc(size);
        if (!r) {
                fprintf(stderr, "Memoria virtuala este consumata\n");
                exit(1);
        }
        return r;
}
```

- Nu scrieţi niciodată într-un buffer de o anumită mărime lucruri care nu încap. Anumite funcţii de bibliotecă, cum ar fi `fgets` citesc de la un periferic un şir de caractere într-un buffer. Ori, nu există nici o limită pentru lungimea şirului tastat de utilizator; oricît de mare ar fi buffer-ul, utilizatorul poate tasta un şir mai lung. Majoritatea bug-urilor faimoase din Unix sunt de acest tip; celebrul "Internet Worm" din 1988, care a pus în genunchi cîteva zeci de mii de calculatoare, se baza în propagarea sa pe un astfel de bug în programul `finger`, care este folosit pentru a identifica utilizatori pe maşini aflate la distanţă. Folosiţi întotdeauna funcţii care limitează numărul de caractere citite, cum ar fi `fgets`.

- Nu folosiţi niciodată alocarea statică. Corectitudinea programelor este mai importantă decît viteza lor de execuţie. De fiecare dată cînd vreţi să scrieţi ceva într-un buffer ne-încăpător, realocaţi buffer-ul făcîndu-l mai mare.

  Funcţia de bibliotecă `realloc` face chiar acest lucru: are ca argument un pointer şi o nouă dimensiune. Rezultatul este un nou buffer, care are dimensiunea indicată, şi care conţine tot vechiul conţinut al buffer-ului iniţial.

- Dacă folosiţi `realloc` pentru a creşte în mod repetat un buffer, trebuie să aveţi grijă să nu plătiţi un cost amortizat prea mare pentru asta. Costul unui `realloc` este proporţional cu mărimea buffer-ului realocat, pentru că fiecare octet trebuie copiat. Din cauza asta se recomandă ca, de fiecare dată cînd realocaţi un buffer, să-i dublaţi lungimea, cum arată şi exemplul următor[4]

```
/* Folosirea lui 'realloc' pentru a mari un buffer pe masura
   datelor pe care trebuie sa le contina */

#include <stdlib.h>

#define DUBLEAZA 1

char *buf = 0;  /* bufferul */
int marime = 0; /* marimea curenta */
int ramas = 0;  /* spatiu liber ramas in buffer */
FILE *f = blabla; /* un fisier oarecare */

while (1) {
        int c = fgetc(f);   /* citim date dintr-un fisier intr-un buffer */
        if (c == EOF) break;
        if (!ramas) {       /* buffer-ul e plin: trebuie realocat */

                /* avem doua variante de realocare */
#if DUBLEAZA
                int marimenoua = marime ? (marime * 2) : 1;
```

[4]Anumite implementări mai vechi de C nu vor executa corect acest program, pentru că ştiu ce să facă atunci cînd funcţia `realloc` primeşte un argument 0, prima oară cînd e chemată. Conform standardului C însă, acest program este corect.

```
#else
                int marimenoua = marime + 1;
#endif

                buf = realloc(buf, marimenoua);
                if (!buf) exit(1);   /* eroare fatala */
                ramas = marimenoua - marime;
                marime = marimenoua;
        }
        buf[marime - ramas] = c;
        ramas--;
}
```

O simplă analiză arată că dacă dublăm lungimea buffer-ului, costul plătit pentru a stoca k caractere este $1 + 2 + 4 + \ldots + 2^{lk}$, unde $lk = \lfloor \log_2 k \rfloor$. (Plătim un cost de 1 pentru realloc(2), 2 pentru realloc(4), etc.). Asta este o serie geometrică, a cărei sumă este $< 2k$.

Pe de altă parte, dacă mărim buffer-ul cu o unitate la fiecare pas, plătim 1+2+3+...+(k-1) $\approx k^2$.

5.2 Nucleul: alocarea de memorie

Un alt alocator se află în nucleul sistemului de operare. Aşa cum am spus deja, acest alocator gestionează atît spaţiul folosit în structurile de date interne nucleului, cît şi spaţiu necesar proceselor care se execută. Interacţiunea dintre variatele alocatoare este înfăţişată în figura 2.

Subiectul acestui articol este cutiuţa etichetată "alocatorul intern al nucleului", deşi multe din principiile indicate se aplică şi celorlalte entităţi. Aceste alocatoare sunt mult mai constrînse decît alocatoarele din spaţiul utilizatorului; în particular trebuie să aibă timpi de răspuns foarte mici, pentru că pot fi chemate de părţi critice din nucleu.

Cele trei alocatoare interacţionează permanent: cînd nucleul nu mai are destule zone pentru propriile lui date cere noi pagini de la alocatorul de pagini. Cînd procesele utilizatorilor mor, paginile lor sunt preluate de alocatorul de pagini. Cînd alocatorul de pagini are puţine resurse disponibile, el poate cere alocatorului intern să returneze din memoria pe care nu o foloseşte în acel moment, etc..

5.3 Nucleul: alocarea de spaţiu pe disc

Un ultim mare subsistem al nucleului care se ocupă de alocarea spaţiului este sistemul care alocă spaţiu pe disc, atît pentru fişiere, cît şi pentru memoria virtuală (memoria proceselor care sunt oprite din execuţie este uneori salvată pe disc pentru a permite executarea altora; această tehnologie se numeşte "memorie virtuală").

Alocatoarele de spaţiu pe disc au alte atribute foarte specifice; de pildă alocatoarele care trebuie să parcurgă o listă întreagă pentru a găsi spaţiul necesar sunt complet nepractice, pentru că accesele la disc durează extrem de mult (milisecunde, ceea ce înseamnă milioane de cicli de procesor). Despre funcţionarea acestor alocatoare am vorbit pe scurt în alte articole din PC Report, consacrate sistemelor de fişiere.

```
        Memoria RAM
           ^
           |
        ___v_____________
        |    alocator   |
        |    de pagini  |
        |_______________|
        ^               ^
       /                 \
______v______      _______v______
| alocatorul|      | sistemul de |<--->procesele utilizatorilor
| intern al |      | paginare    |
| nucleului |      |_____________|<--->cache-ul de disc
|___________|
     ^
     |
     v
structuri de date interne ale nucleului
(buffere pentru retea, inoduri, tabela de procese, etc.)
```

Figura 2: Interacţiunea dintre feluritele alocatoare de memorie din nucleu. Alocatorul de pagini gestionează întreaga memorie fizică a maşinii. El generează spaţiu atît pentru procese (unitatea de bază fiind pagina), cît şi memorie pentru alocatorul intern al nucleului. Săgeţile indică schimb de spaţiu de memorie între componente.

6 Un alocator dinamic foarte simplu

Voi ilustra aici în scop pur didactic implementarea unui alocator extrem de ineficient şi risipitor, dar totodată complet. Aceasta este schema de bază de funcţionare a tuturor alocatoarelor din nucleu; variaţiunile constau în îmbunătăţiri. Acest alocator implementează funcţia `free()` fără a face nimic! Memoria eliberată este pur şi simplu pierdută. Alocatorul extrage zonele de memorie dintr-o memorie mare alocată static iniţial, care are 16 megaocteţi. Puteţi folosi acest alocator în programele dumneavoastră!

```
/* implementarea unui alocator dinamic foarte simplu; functia
   free() nu face absolut nimic. */

typedef unsigned int size_t;

#define MARIME_MEMORIE 16 * 1024 * 1024
char memorie[MARIME_MEMORIE];

int memorie_consumata = 0;
```

```
void * malloc(size_t marime)
{
        if (MARIME_MEMORIE < marime + memorie_consumata)
                return 0;
        memorie_consumata += marime;
        return (void*) &memorie[memorie_consumata - marime];
}

void free(void *)
{}
```

Acest alocator este extrem de rapid; probabil că nici nu poate fi scris un alocator mai rapid. Preţul plătit este o enormă risipă de memorie: tot ce este dealocat se pierde complet.

Oricum, aceste funcţii ilustrează cărămizile de bază cu care un alocator din nucleu trebuie să opereze. Alocatorul de pagini manipulează memoria fizică a maşinii ca un vector enorm de octeţi, din care extrage porţiuni aşa cum facem noi cu vectorul numit `memorie`.

7 Alocatorul cu hartă de resurse

Acesta este probabil cel mai simplu tip de alocator care implementează şi funcţia `free`. Acest alocator menţine un vector de structuri care descriu fiecare bloc liber. Figura 3 arată o posibilă reprezentare a situaţiei la un moment dat:

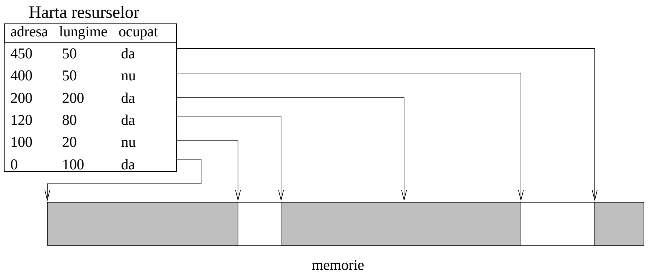

Figura 3: Alocatorul cu hartă de resurse are o "hartă" care indică fiecare regiune liberă şi ocupată.

Avem o mică problemă, pentru că alocatorul are nevoie el însuşi de spaţiu pentru a reprezenta tabela cu resurse. Putem face însă nişte mici trucuri pentru a compacta reprezentarea:

- Cînd `malloc` găseşte un bloc, foloseşte primii 4 octeţi[5] pentru a memora lungimea sa şi returnează adresa următorului octet. Această lungime este folosită de `free` pentru a şti cît de mult trebuie dealocat.

```
Forma unui bloc de memorie ocupat: fiecare bloc este
```

[5]Valoarea 4 este plauzibilă pentru maşini pe 32 de biţi; lungimea ar putea avea nevoie de mai mulţi biţi pe maşini care suportă mai mult de 4Go de spaţiu de adrese.

```
precedat de 4 octeţi care conţin lungimea blocului.

-------------------------------------------
|  lungimea  |                            |
-------------------------------------------
             ^
             \ adresa returnată de malloc
```

- Putem memora informaţia despre blocurile libere în chiar spaţiul ocupat de blocurile libere! Astfel, putem ţine blocurile libere într-o listă înlănţuită, ca în figura 4.

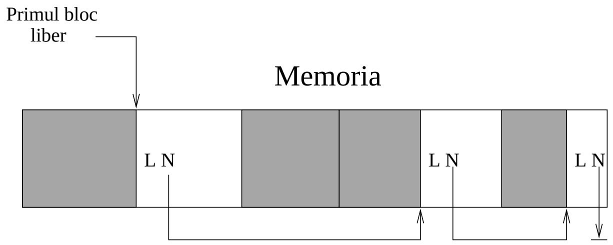

Figura 4: Putem implementa alocatorul cu hartă de memorie menţinînd în fiecare bloc liber informaţie despre lungimea proprie (L) şi adresa următorului bloc liber (N).

  Cînd alocatorul vrea să găsească un bloc liber pleacă de la adresa primului bloc liber şi parcurge lista de blocuri libere pînă găseşte unul de dimensiune corespunzătoare.

Acest alocator este relativ simplu, dar foarte ineficient. Din cauza fragmentării complexitatea operaţiilor este mare (după o vreme lista de blocuri libere devine populată cu o sumedenie de blocuri mici, a căror traversare este costisitoare şi inutilă). Schema este de asemenea uşor risipitoare: 4 octeţi sunt memoraţi în plus pentru fiecare bloc ocupat, iar faptul că un bloc liber trebuie să conţină două valori impune o lungime minimă de 8 octeţi pentru un bloc.

Eliberarea unui bloc îl inserează la începutul listei de blocuri libere. Dacă vrem să reducem fragmentarea, la eliberare putem să comasăm un bloc cu vecinul următor, în caz că acesta este liber şi el. Dacă vrem să putem comasa şi cu vecinul anterior, atunci în fiecare bloc memorăm şi în ultimii 4 octeţi lungimea sa; în acest fel, atunci cînd eliberăm un bloc, putem şti exact unde începe cel de dinainte. În orice caz, eliberarea cere un număr constant de instrucţiuni (care nu depinde de numărul de zone deja alocate).

8 Alocatorul cu puteri ale lui 2

Problema alocatorului de mai sus constă în faptul că trebuie să *caute* un bloc de dimensiune potrivită. Pentru a remedia acest lucru putem folosi o altă reprezentare: păstrăm o serie de liste de blocuri, toate blocurile de pe fiecare listă avînd o anumită dimensiune. Atunci cînd vrem un anumit bloc, returnăm un bloc de dimensiunea imediat superioară. Cele mai folosite dimensiuni sunt puteri ale lui 2, ca în figura 5.

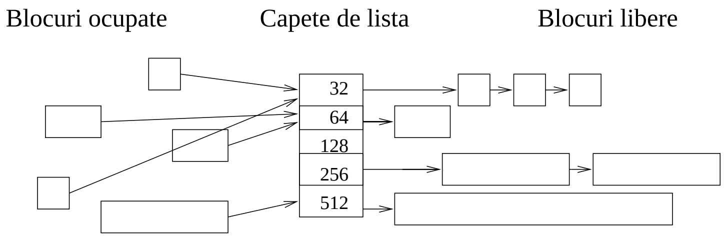

Figura 5: Structurile de date ale alocatorului cu puteri ale lui 2: cîte o listă de blocuri libere pentru fiecare dimensiune de bloc. Fiecare bloc ocupat are informaţii despre propria sa lungime.

Acest alocator suferă de fragmentare internă, pentru că alocă întotdeauna mai mult decît i se cere.

Alocarea şi dealocarea cer timp constant pe operaţiune (calculul listei plus extragerea primului bloc din listă). Putem reprezenta listele de blocuri libere exact ca în schema cu harta de resurse; comasarea este practic imposibil a (am putea comasa numai vecini de dimensiuni egale, ca să obţinem rezultate de mărime tot puteri ale lui doi). Pentru toate blocurile mai mici de o anumită limită şi mai mari de o alta, putem avea nişte liste separate, în care facem căutare exhaustivă. Dacă vrem să avem liste cu toate dimensiunile posibile de blocuri sunt suficiente 32 de liste, ceea ce e o valoare rezonabilă.

În momentul în care o listă dorită este complet depopulată, putem scoate un bloc mai mare de pe lista imediat superioară, pe care îl putem sparge în două bucăţi mai mici.

În interiorul nucleului există multe structuri de date care au ca dimensiuni exact puteri ale lui 2; pentru astfel de cereri se iroseşte o cantitate enormă de memorie, pentru că fiecare bloc alocat trebuie să conţină şi cei 4 octeţi suplimentari. Astfel, dacă vrem să alocam 256 de octeţi, trebuie de fapt 260, care înseamnă ca folosim un bloc de 512; risipa este de aproape 100%!

Alte dezavantaje ale metodei: în general blocurile mici nu se pot grupa laolaltă pentru a forma blocuri mari, şi sistemul de paginare nu poate obţine pagini nefolosite înapoi în caz de nevoie.

9 Alocatorul Karels-McKusick

În 1988 Kirk McKusick şi Michael Karels au construit o variantă îmbunătăţită a alocatorului cu puteri ale lui 2, cere este folosită acum în 4.4BSD Unix şi Digital Unix. Metoda lor elimină risipa pentru cazul blocurilor care au dimensiuni exact puteri ale lui 2.

Listele de blocuri libere sunt înlănţuite exact ca în alocatorul cu puteri ale lui 2. Diferenţa principală constă în modul în care blocurile ocupate îşi reprezintă lungimea; trucul folosit este foarte ingenios: alocatorul menţine un vector mare de numere, cîte unul pentru fiecare pagină. Elementul 5 din vector reprezintă mărimea blocurilor din pagina 5. Figura 6 arată un exemplu de structuri de date ale alocatorului.

O altă optimizare făcută de alocatorul acesta este modul în care se calculează rotunjirea în sus a unei valori la cea mai mică putere a lui 2; calculul este făcut cu un macro care conţine o expresie de genul:

```
#define LISTA(marime) \
 (marime) > 128 \
```

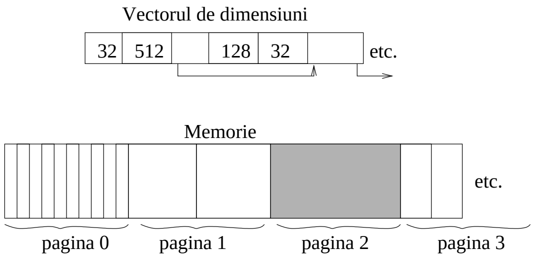

Figura 6: Vectorul care reprezintă mărimea blocurilor pentru fiecare pagină în alocatorul Karels-McKusick. Pagina 0 este divizată în blocuri de cîte 32 de octeţi, pagina 1 în blocuri de cîte 512, etc. Paginile încă nefolosite sunt înlănţuite într-o lista.

```
? (marime) > 256 ? 4 : 3 \
: (marime) > 64 \
        ? 2 \
        : (marime) > 32 ? 1 : 0
```

Avantajul unei astfel de expresii în locul unei bucle este că, atunci cînd mărimea este cunoscută la compilare, întreaga expresie devine o constantă care poate fi calculată de compilator, deci codul executat devine mult mai rapid.

Acest alocator este mai eficient decît cel cu puteri ale lui 2, şi risipeşte mult mai puţină memorie, pentru că toate cererile egale cu o putere a lui doi sunt satisfăcute cu un buffer de mărime potrivită. Celelalte dezavantaje rămîn însă prezente.

10 Alocatorul "slab"

O schemă mult mai sofisticată de alocare, inspirată din limbajele orientate pe obiecte a fost introdusă de Sun în sistemul de operare Solaris 2.4 şi este acum implementată şi în Linux. "Slab" înseamnă în engleză "lespede"; acest alocator are zone de memorie diferite pentru obiecte diferite, pe care le putem vedea ca nişte mozaicuri de forme diferite; de aici şi numele. Acest alocator tratează cu multă atenţie o serie de aspecte importante pentru performanţă care sunt complet ignorate de alte alocatoare:

- Alocatorul încearcă atunci cînd caută noi zone să nu atingă prea multe locaţii de memorie, pentru a nu polua cache-ul microprocesorului cu date care nu sunt utile; se spune că alocatorul are o "urmă" mică (*small footprint*);

- Alocatorul încearcă să aloce obiectele în memorie în aşa fel încît două obiecte diferite să nu pice în aceeaşi linie din cache-ul de date; asta maximizează utilizarea cache-ului cînd obiectele sunt folosite simultan. De exemplu obiectul care reprezintă în nucleu un fişier are cîteva cîmpuri care sunt foarte des folosite, aflate să zicem la începutul structurii, şi alte cîteva cîmpuri care sunt extrem de rar folosite. Să presupunem că obiectul "fişier" are 256 de octeţi; în acest caz toate cîmpurile des folosite din toate fişierele se vor afla la adrese multiplu de 256. Din cauza asta ele se vor lupta pentru un set restrîns de linii din cache, în vreme ce celelalte linii vor fi prost utilizate. Alocatorul slab va încerca să aloce fiecare obiect la o adresă care nu e multiplu de aceeaşi valoare.

- Alocatorul încearcă să reducă numărul de operaţii de iniţializare asupra noilor obiecte alocate. De exemplu multe obiecte din nucleu care pot fi folosite de mai multe procese au un cîmp care indică numărul de utilizatori (de pildă un fişier memorează numărul de procese care au deschis fişierul). Cînd acest număr ajunge 0, înseamnă că obiectul nu mai este utilizat, şi el poate fi dealocat. Asta înseamnă că avem certitudinea că orice fişier dealocat va avea valoarea 0 în acel cîmp. Atunci cînd alocăm un nou fişier, dacă folosim vechea zonă de memorie, ştim deci că nu mai trebuie să iniţializăm acest cîmp, pentru că are deja valoarea corectă! Practic nucleul menţine un cache cu obiecte de curînd dealocate, pe care le foloseşte atunci cînd i se cer noi alocări.

Alocatorul slab constă de fapt într-o rutină centrală (numită `kmem_cache_create`) care crează alocatoare pentru fiecare obiect. Rutina asta primeşte ca parametri numele obiectului, mărimea unui obiect, constrîngerile de aliniere (de ex. toate obiectele trebuie să fie la o adresă multiplu de 16) şi pointeri spre o funcţie constructor şi o funcţie destructor.

Iată un exemplu de utilizare:

```
alocator_de_fisiere = kmem_cache_create("fisier", sizeof(struct fisier),
                                        8, constructor_fisier,
                                        destructor_fisier);

alocator_de_inoduri = kmem_cache_create("inod", sizeof(struct inod),
                                        4, constructor_inod,
                                        destructor_inod);

struct fisier * fs = kmem_cache_alloc(alocator_de_fisiere, FLAGS);
struct inod * in = kmem_cache_alloc(alocator_de_inoduri, FLAGS);
```

Funcţia `kmem_cache_create` crează deci alocatoare pentru felurite obiecte. Fiecare alocator este apoi folosit pentru a construi obiecte de acel tip.

Fiecare alocator are propria lui zonă de memorie, în care menţine numai obiecte de acel tip; va exista astfel o zonă de pagini în care se alocă numai fişiere, o zonă în care se alocă numai inoduri, etc. În acest fel toate obiectele dintr-o zonă au aceeaşi dimensiune!

Fiecare alocator menţine de asemenea o listă cu obiectele care au fost de curînd dealocate, şi le refoloseşte atunci cînd i se cer noi obiecte. Pentru că obiectele au fost dealocate, ele nu mai trebuie sa fie iniţializate din nou.

Atunci cînd un alocator epuizează toată memoria pe care o are la dispoziţie, el cere de la alocatorul de pagini o nouă pagină pe care o umple cu obiecte noi. Pentru că aceste obiecte nu au fost niciodată iniţializate, alocatorul cheamă funcţia constructor care a fost indicată pentru fiecare nou obiect. Observaţi că acest constructor este chemat numai prima oară cînd pagina este obţinută; după ce un obiect a fost dealocat constructorul nu mai este chemat din nou.

Fiecare alocator foloseşte propriile lui pagini într-un mod similar cu alocatorul cu puteri ale lui doi. La sfîrşitul fiecărei pagini alocatorul rezervă o zonă pentru o structură de date care descrie

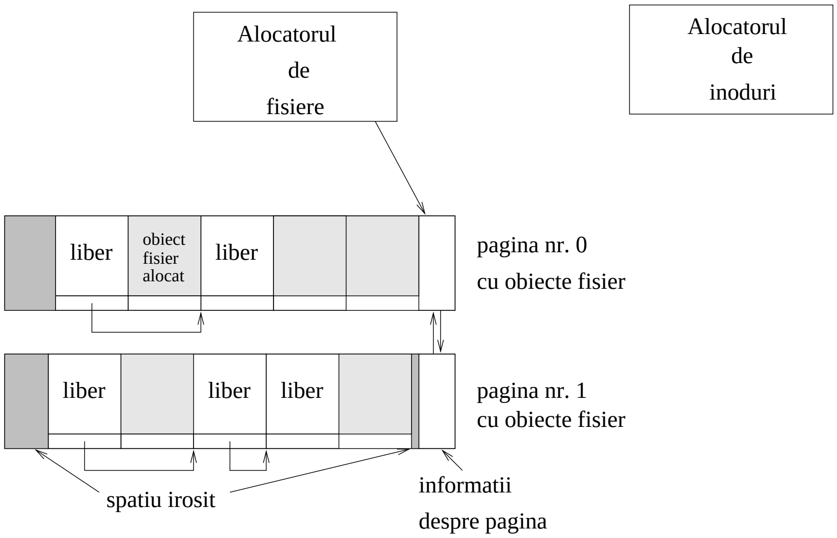

Figura 7: Alocatorul "slab". Fiecare obiect are propriul lui alocator, care menţine în pagini obiectele alocate şi libere. În fiecare pagină primul obiect alocat începe la altă adresă, pentru a încărca uniform liniile din cache-ul microprocesorului. Fiecare pagină mai posedă anumite structuri de date folosite la întreţinere (cum ar fi o listă dublu înlănţuită a tuturor paginilor pentru un anumit obiect).

ocupaţia acelei pagini. Primul obiect în pagină este plasat la o distanţă aleatoare de marginea pagini; acest plasament are efectul de a pune obiecte din pagini diferite la adrese diferite în cache, exact cum am indicat mai sus.

Alocatorul slab poate returna sistemului de paginare paginile în care toate obiectele sunt nefolosite. Alocatorul are o "urmă" mică, pentru că majoritatea cererilor accesează o singură pagină. Acest tip de alocator risipeşte ceva resurse datorită modului de plasare în pagină şi pentru că are o zonă diferită pentru fiecare tip de obiect. Performanţa lui este însă excelentă, şi rămîne unul din cele mai puternice alocatoare implementate.

11 Concluzii

Alocarea memoriei este un subiect foarte generos, care suscită în continuare interes cercetătorilor. Gestiunea memoriei este un din funcţiile principale ale unui sistem de operare. Nucleul unui sistem de operare gestionează întreaga memorie fizică a unui calculator; el oferă memorie atît sieşi (nucleului), pentru funcţionarea sa, cît şi proceselor executate de utilizatori. Procesele utilizatorilor gestionează la rîndul lor bucăţelele primite de la nucleu.

Dacă subiectul vă interesează mai aveţi multe de aflat în domeniu. Dacă nu, sper să-i alocaţi măcar 2-3 neuroni în memoria dumneavoastră. Alegeţi un algoritm în acest scop.