

# Upravljanje memorijom u modernim operacijskim sustavima

© 2005 Željko Vrba

Ovaj članak daje kratak prikaz memory managementa u modernim operacijskim sustavima te objašnjava osnovne pojmove. Nakon osnovnih pojmova, opisan je osnovni<sup>1</sup> POSIX API koji aplikacijama nudi na raspolaganje velike mogućnosti pri upravljanju memorijom.

1	Osnovni pojmovi	1
2	Mehanizmi upravljanja memorijom	3
2.1	Virtualni adresni prostor	3
2.2	Zaštita memorije	3
2.3	Alokacija fizičke memorije	4
3	UNIX API-ji	4
3.1	Alokacija memorije i pristup datotekama	4
3.2	Dijeljena memorija	5
3.3	Ostale funkcije	5

## 1 Osnovni pojmovi

Danas se spominju mnoge “vrste” memorije, poput RAM, cache, virtualna ili fizička, itd. Ovdje će se razjasniti značenje i uloga svake pojedine vrste memorije. Ulaženjem u dubinu bi ovaj članak postao predugačak pa su stoga neke teme obrađene samo konceptualno i zato možda ne potpuno točno u svim tehničkim detaljima.

**Fizička memorija** je ona koja (kako samo ime kaže) fizički postoji u kompjuteru. To je RAM (random-access memory), odn. glavna radna memorija procesora. Da bi procesor mogao obraditi neki podatak, on se mora nalaziti u memoriji<sup>2</sup> ili registru.

**Adresa** je pozitivan cijeli broj koji jednoznačno omogućava dohvat nekog podatka. Kada procesor želi dohvatiti podatak dešava se sljedeće (tzv. *memorijski ciklus čitanja*):

1. Procesor na *adresnu sabirnicu* stavlja adresu podatka.
2. Procesor šalje memoriji signal da je na adresnoj sabirnici ispravna adresa (tzv. ADDRESS STROBE signal).
3. Memorija na *podatkovnu sabirnicu* postavlja podatak.
4. Memorija šalje procesoru signal da je traženi podatak postavljen na podatkovnu sabirnicu (tzv. DATA STROBE signal).
5. Procesor dohvaća podatak sa podatkovne sabirnice u registar.

Ovo je vrlo pojednostavljen tijek događaja (izostavljen je ostali prisutni hardware, poput memorijskog i cache kontrolera, arbitraže na sabirnici itd.).

Ako se radi o read-modify-write instrukciji, nakon izvršenja instrukcije postoji sličan *memorijski ciklus pisanja* kojim procesor zapisuje podatak nazad u memoriju.

**Brzina memorije** (ili vrijeme pristupa) jest vrijeme potrebno da procesor iz memorije dohvati traženi podatak. Vrijeme je kratko (reda veličine nanosekundi), ali za današnje procesore koji

<sup>1</sup> Postoje i napredni POSIX API-ji koji su opcionalni.

<sup>2</sup> U nastavku će se pod “memorija” uvijek podrazumijevati fizička memorija. Ako bude riječi o nekoj drugoj vrsti, to će se eksplicitno naglasiti.

rade na gigahercnim frekvencijama – izuzetno sporo. Kada procesor zatraži podatak iz memorije, a ne nalazi se u cacheu, ne može raditi ništa osim *čekati* da memorija “isporuči” podatak.

**Registri** su “memorija” koja je prisutna na samom čipu procesora. Dobra strana registara je da rade na brzini procesora – nema vremena pristupa i procesor može odmah započeti obradu podatka (nema čekanja da podatak stigne). Idealno bi bilo sve podatke držati u registrima. Tu se pokazuje njihova loša strana: ima ih *jako malo* (ovisno o procesoru, ali obično < 128). Kada se ne mogu svi podaci držati u registrima, neki se moraju smjestiti u memoriju, čime se automatski usporava rad programa.

**Cache memorija** služi kao indirektni kanal između procesora i fizičke memorije. Fizička memorija se tehnički izvodi kao DRAM (dinamičke ćelije), dok se cache memorija izvodi kao SRAM (statičke memorijske ćelije). SRAM ćelije su *brže* od DRAM ćelija, ali na silicijskoj pločici *zauzimaju više mjesta* – na istu površinu stane manji kapacitet memorije. Samim time su *skuplje*.

Kada procesor dohvaća podatak iz memorije, memorijski kontroler prvo pogleda da li podatak već postoji u brzom cacheu. Tek ako ne postoji, dohvaća se iz RAM-a.

Postoje *razine* cache memorije: L1 (level 1) se nalazi na istom čipu kao i procesor (i jednako je brza kao procesor) dok se L2 nalazi odvojeno od procesora i nešto je sporija (ali i dalje brža od glavne memorije). Kako je prostor na čipu procesora ograničen, tako je ograničena i količina L1 memorije (L1 je danas obično reda veličine 64kB, dok je L2 reda veličine megabajta).

Iako L1 cache radi brzinom procesora, instrukcije koje referenciraju memoriju su i dalje sporije od instrukcija koje rade sa registrima. I dalje postoji memorijski ciklus, jedino je eliminirano vrijeme čekanja na dohvat podatka iz memorije.

Treba napomenuti da količina cache memorije *ne povećava* količinu memorije raspoloživu za aplikacije: postojanje cache memorije je u potpunosti transparentno procesoru – sav posao obavi memorijski kontroler. što je veća količina cache memorije, sistem brže radi jer je veća vjerojatnost da se traženi podatak nalazi u cacheu.

**MMU** (memory management unit) je dio procesora koji prevodi virtualne adrese u fizičke. Sa stanovišta MMU-a, fizička memorija je podijeljena na *page frameove* fiksne veličine, obično 4kb ili 8kb (kod nekih arhitektura procesora je to konfigurabilno).

**Page tables** su strukture podataka (za njih OS rezervira dio RAM-a) pomoću kojih MMU radi translaciju virtualnih u fizičke adrese. Translacija se ne radi za najmanje jedinice adresiranja (tzv. *riječi*, danas 32 ili 64 bita) nego za intervale adresa veličine jednog page framea. Tako je u page tablici potreban samo *jedan* podatak od 32 bita<sup>3</sup> za cijeli niz od 4kB adresa (ili koliko je već veličina page framea).

**Swap** je unaprijed rezervirani prostor na *disku* na koji OS sprema dijelove “neaktivnih” programa kada ponestane fizičke memorije. Naravno, to dosta usporava rad tih programa kada opet pristupe podacima na disku jer je disk nekoliko redova veličine sporiji od RAM-a.

**Buffer cache** je<sup>4</sup> dio RAM-a koji OS rezervira radi bržeg čitanja sa *diska*. Kada je potrebno dohvatiti sektor sa diska, OS prvo pogleda da li se taj sektor već nalazi u buffer cacheu i tek ako ga tamo nema, dohvaća se sa diska (koji je mnogo sporiji od RAM-a).

<sup>3</sup> Na 32-bitnoj arhitekturi.

<sup>4</sup> U danima DOS-a to se zvalo samo cache jer se znalo na što se misli: ono što je radio `smartdrv.exe` program. Danas je potrebno naglasiti na koji cache se misli.

Veličina buffer cache-a nije fiksna već ju OS dinamički prilagođava prema ukupnoj trenutnoj potražnji memorije svih aktivnih programa, uzimajući u obzir i druge parametre.

Većina ovdje opisanih pojmova oslanja se na tzv. *princip lokalnosti*: za podatak koji je upravo zatražen (bilo sa diska, RAM-a ili swapa) postoji velika vjerojatnost da će biti potreban u skoroj budućnosti. Za ogromnu većinu današnjih aplikacija taj princip vrijedi i upravo stoga je moguće imati mnogo pokrenutih programa koji ukupno zauzimaju mnogo više memorije nego što je fizički raspoloživo, a da se cijeli sistem ne usporava jako.

## 2 Mehanizmi upravljanja memorijom

Sadržaj ovog odjeljka podjednako vrijedi za mnoge operacijske sustave danas u upotrebi (Win32, razne varijante UNIX-a, MacOS X itd.)

### 2.1 Virtualni adresni prostor

Svaki proces u trenutku svog stvaranja dobiva nesegmentirani<sup>5</sup> 32-bitni ili 64-bitni adresni prostor. Proces ima *iluziju* da ima svu memoriju za sebe. Kada adresira memoriju, on zapravo adresira *virtualnu adresu*. MMU virtualnu adresu pretvara u fizičku adresu (pomoću page tablica koje uspostavi OS) te na adresu sabirnicu stavlja pravu fizičku adresu.

Postupak pretvaranja virtualne u fizičku adresu je sasvim transparentan za aplikaciju pa čak i za sam procesor *osim* u jednom slučaju: ako u je page tablicama adresirani page frame označen da *nije prisutan*. Tada se automatski poziva tzv. *page fault handler* (posebna rutina unutar OS-a) koji mora pronaći gdje se na swap prostoru diska nalazi tražena virtualna adresa.<sup>6</sup> Zatim u fizičkoj memoriji traži slobodan page frame kamo će se sa diska učitati podaci. Nakon toga modificira page tablice procesa tako da virtualna adresa pokazuje na pravi page frame. Ukoliko page fault handler ne nađe slobodan page frame u fizičkoj memoriji, tada mora izabrati jedan koji sprema u swap prostor na disk *prije* učitavanja potrebnog page framea.

### 2.2 Zaštita memorije

MMU ima i drugu ulogu: *zaštitu memorije*. Naime, dva različita procesa često generiraju *iste adrese*. No OS postavi page tablice tako da se iste virtualne adrese “mapiraju” na različite fizičke adrese. Tako su programi zaštićeni (ne mogu čitati i mijenjati tuđe podatke) jedni od drugih. “Rušenje” jednog programa neće izazvati rušenje drugih ili čak “pad” cijelog OS-a.<sup>7</sup>

Iduća vrsta zaštite memorije jesu user/supervisor oznake. Procesor ima obično dva<sup>8</sup> načina rada: user i supervisor. Isključivo u supervisor modu je dozvoljeno izvršavanje nekih instrukcija koje mogu ozbiljno poremetiti rad sustava (npr. isključivanje interrupta). Neke virtualne adrese se u page tablicama procesa mogu označiti kao supervisor i proces tada nema nikakav pristup. U slučaju da im proces ipak pokuša pristupiti, aktivira se page fault handler koji detektira pristup nedozvoljenoj adresi i “nasilno” završava program.<sup>9</sup>

<sup>5</sup> Neki operacijski (u starije vrijeme) sustavi imaju segmentirani memorijski model; najbolji primjer je DOS. Takav model je problematičan jer viši programski jezici ne podržavaju segmentaciju.

<sup>6</sup> Zapravo treba dvije informacije: o procesu i virtualnoj adresi.

<sup>7</sup> Teoretski, kao što ste se sigurno imali prilike uvjeriti gledajući u tzv. BSOD – Blue Screen of Death na Win OS-ovima.

<sup>8</sup> Npr. x86 arhitektura ima 4.

<sup>9</sup> Na UNIX-u se isporučuje SIGSEGV signal.

Posljednja vrsta zaštite<sup>10</sup> je dozvoljena vrsta pristupa memoriji: neki page frameovi mogu biti označeni samo za čitanje, a neki mogu biti označeni da nije dozvoljeno izvršavanje koda. Neodgovarajuća vrsta pristupa (npr. pisanje po adresama ili izvršavanje koda sa adresa koje nemaju odgovarajuću dozvolu) aktivira page fault handler koji detektira nedozvoljenu vrstu pristupa i završava program.

## 2.3 Alokacija fizičke memorije

Fizička memorija je dijeljeni resurs kojim može raspolagati jedino OS. Kada program treba neku količinu memorije, određenim *sistemskim pozivima* (o kojima će biti govora kasnije) traži od OS-a memoriju. Standardne C funkcije poput `malloc` su *library funkcije* koje unutar sebe imaju pozive OS-u za alociranje ili oslobađanje memorije.

Uobičajeni postupci pri alokaciji memorije su "lijena" alokacija i tzv. "memory overcommit". Kad proces zatraži memoriju jezgra OS-a (kernel) samo učini važećim određeni raspon adresa. Stvarna alokacija memorijskih stranica se dešava tek pri prvom pristupu procesa tom rasponu adresa u memoriji. Memorija je "prebukirana" i tom smislu da se alokacija dozvoljava procesu ako ima dovoljno slobodnog prostora u adresnom prostoru tog procesa, a ne provjerava se da li ima dovoljno stvarne, fizičke memorije (RAM + swap).

Može se desiti da ukupna količina memorije dodijeljena svim procesima prelazi ukupnu količinu fizičke i swap memorije. Ako svi procesi odluče u isto vrijeme koristiti svu memoriju koja im je dodijeljena sustavu ponestaje memorijskih resursa i obično se u tom slučaju gasi ("ubija") neke procese. Ovaj scenarij se rijetko dešava u praksi. Prebukiranje memorije obično ne stvara nikakve probleme i olakšava zahtjeve na količinu slobodne memorije, a posebno swap memorije. (Solaris npr. sa isključenom opcijom prebukiranja zahtijeva da svaka alokacija RAM memorije bude popraćena alokacijom odgovarajuće količine swap prostora.)

Postoji nekoliko pravila za odabir koji proces će biti "ubijen", najčešće je to slučajni odabir ili proces koji trenutno koristi najviše memorije. U nekim scenarijima gdje gašenje nekog procesa nije prihvatljivo (npr. serveri kritičnih baza podataka), prebukiranje memorije se može isključiti i ne dozvoliti alokaciju memorije u slučaju kad nema dovoljno fizičke memorije. U takvim slučajevima sama aplikacija koja je tražila memoriju može obaviti radnje potrebne za slučaj manjka memorije.

## 3 UNIX API-ji

Na UNIX-u postoji mnogo API-ja kojima aplikacija može upravljati dodijeljenim joj adresnim prostorom.

### 3.1 Alokacija memorije i pristup datotekama

Bibliotečna funkcija `sbrk` i pripadajući sistemski poziv `brk` su najstariji način za davanje zahjeva za memorijom operacijskom sustavu, potekli iz vremena kada još nije postojala virtualna memorija. Oni rade na principu najveće adrese kojoj se može pristupiti u adresnom prostoru procesa. Ne skalira se dobro na moderne 32-bitne i 64-bitne adresne prostore koji često imaju mnogo "rupa". Ne

<sup>10</sup> Ako se uzima samo MMU u obzir. x86 arhitektura nudi puno finije načine zaštite mehanizmima segmentacije, no te mogućnosti su u današnjim OS-ovima neiskorištene. Razlog je uglavnom portabilnost OS-a koji je većinom pisan u C-u te arhitekture drugih procesora koje nemaju segmentaciju, imaju samo user/supervisor mod i koriste isključivo MMU za zaštitu programa.

nalazi se u POSIX standardu ali implementira ju svaka moderna varijanta UNIX-a (vjerojatno zbog kompatibilnosti sa starim programima i jednostavnosti izvedbe), ponegdje s nekim ograničenjima.

Noviji API je `mmap` funkcija koja mapira *dio datoteke* u adresni prostor procesa. Ako je mapiranje dvosmjerno (čitavanje i pisanje) tada se ne može koristiti za proširivanje datoteke preko početne veličine (pokušaj da se to napravi rezultirat će signalom SIGBUS ili ekvivalentnom koji se dostavlja procesu). Čest pristup za čistu memorijsku alokaciju je otvoriti uređaj `/dev/zero` i mapirati dobiveni datotečni deskriptor što nam daje memorijske stranice s vrijednostima postavljenim na 0. Za čistu memorijsku alokaciju uveden je pojam "anonimnog mapiranja", odn. postupak koji ne zahtijeva korištenje ispravnog opisnika datoteke. `mmap` sučelje se može koristiti i za simuliranje dijeljene memorije preko zajedničke datoteke.

Ukoliko aplikacija promijeni sadržaj datoteke pisanjem po memoriji koju je vratio `mmap`, obavezno mora pozvati `msync` funkciju kako bi se promjene stvarno zapisale u datoteku – jer se to neće desiti automatski nakon zatvaranja datoteke sa `close`. Ono što je ipak garantirano jest da ako jedna aplikacija preko mapiranja promijeni neku datoteku, tu promjenu će vidjeti svaka druga aplikacija<sup>11</sup> i bez poziva `msync` funkcije u aplikaciji koja je učinila promjenu.

`munmap` funkcija briše mapiranja koja su napravljena sa `mmap` i oslobađa sistemske resurse.

## 3.2 Dijeljena memorija

System V UNIX je uveo nekoliko IPC mehanizama među kojima su i segmenti dijeljene memorije (shared memory segments). Svaki segment ima jedinstveni *ključ* preko kojega mu pristupaju sve aplikacije. Aplikacije se ili unaprijed dogovore oko ključa (npr. pomoću environment varijabli) ili ga generiraju na temelju neke poznate datoteke (u tome pomaže `ftok` funkcija).

Segmenti dijeljene memorije su stalni ("persistent"): oni postoje i zauzimaju resurse sve dok ih se eksplicitno ne pobriše, čak i kad proces koji ih je koristio više ne postoji.

Funkcije za rad sa dijeljenom memorijom su sljedeće:

- `shmget` koja na temelju ključa vraća ID postojećeg segmenta ili kreira novi segment.
- `shmat` koja omogućava korištenje segmenta u aplikaciji tako što joj dodijeli konkretnu adresu u virtualnom adresnom prostoru procesa.
- `shmdt` koja poništava efekt `shmat` funkcije: nakon njenog poziva aplikacija više ne može pristupiti segmentu, a pointer koji je vratila `shmat` funkcija više nije ispravan.
- `shmctl` kojom je moguće, između ostaloga, izbrisati segment.

## 3.3 Ostale funkcije

`mlock`, `mlockall`, `munlock` i `munlockall` se koriste za zaključavanje, odn. otključavanje nekih ili svih memorijskih stranica nekog procesa i dostupne su samo root (super-user) korisniku.<sup>12</sup> Često se koriste u algoritmima koji rade u realnom vremenu ili u obradi podataka s visokom razinom sigurnosti (npr. GnuPG ih koristi kad može tako da ključevi koji su inače enkriptirani ne bi završili dekriptirani na swap prostoru).

`mprotect` funkcijom se mogu promijeniti dozvole pristupa nad nekim nizom virtualnih adresa. Npr. kod aplikacije je često označen samo za čitanje, a ovim pozivom se može omogućiti i pisanje. Ovaj poziv se najčešće koristi kada defaultne dozvole koje dodijeli OS ne odgovaraju potrebama aplikacije.

<sup>11</sup> Bilo da mapira datoteku ili joj pristupa na tradicionalan način sa `read`.

<sup>12</sup> "Zaključati" dio memorije znači spriječiti OS da ga u bilo kojem trenutku izbací na swap.

