Osnovna memorija

1 Uvod

U ranijim poglavljima smo pokazali kako se CPU može deliti od strane skupa procesa. Kao rezultat raspoređivanja CPU-a, možemo poboljšati kako iskorišćenost CPU-a tako i odzivnost računara prema korisnicima. Da bismo ostvarili ovo poboljšanje performansi, međutim, moramo imati mogućnost čuvanja nekoliko procesa u memoriji, to jest, moramo *deliti memoriju*.

U ovom poglavlju ćemo razmotrili različite načine upravljanja memorijom. Algoritmi za upravljanje memorijom razlikuju se od primitivnog pristupa memoriji po tome što implementiraju dve strategije: *straničenje* i *segmentaciju*. Svaki pristup ima svoje prednosti i mane. Izbor metode upravljanja memorijom za određeni sistem zavisi od mnogih faktora, naročito od hardverskog dizajna sistema. Kao što ćemo videti, mnogi algoritmi zahtevaju hardversku podršku, što diktira usko integrisan hardveru sa modulom za upravljanje memorijom operativnog sistema.

2 Osnove

Kao što smo videli ranije, memorija je centralna komponenta u radu savremenog računarskog sistema. Memorija se sastoji od velikog niza bajtova, a svaki element niza ima svoju adresu. CPU uzima instrukcije iz memorije koristeći registar *programski brojač* (*program counter* - PC). Preuzeta instrukcija može prouzrokovati dodatno učitavanje ili upisivanje na određenu memorijsku adresu. Na primer, tipični ciklus izvršavanja instrukcija najpre uzima instrukciju iz memorije. Instrukcija se zatim dekodira, što može prouzrokovati da se operandi, potrebni za izvršavanja instrukcije, preuzmu iz memorije. Nakon izvršavanja instrukcije na prezetim operandima, rezultati se opet mogu sačuvati u memoriji. Memorijska jedinica, pri tome, vidi samo tok memorijskih adresa; ne zna kako se oni generišu (programskim brojačem, direktnim ili inderektnim adresiranjem, konstantama koje predstavljaju memorijske adrese, itd.) ili čemu služe (instrukcije ili podaci). Prema tome, možemo zanemariti kako program generiše memorijsku adresu. Nas zanima samo redosled memorijskih adresa generisanih programom koji se izvršava.

Započinjemo sa diskusijom nekoliko pitanja koja su relevantna za upravljanje memorijom:

- osnovni hardver,
- povezivanje simboličkih memorijskih adresa sa stvarnim fizičkim adresama
- razlikama između logičke i fizičke adrese.

Odeljak zaključujemo diskusijom u vezi sa dinamičkim povezivanjem i deljenim bibliotekama.

2.1 Osnovni hardver

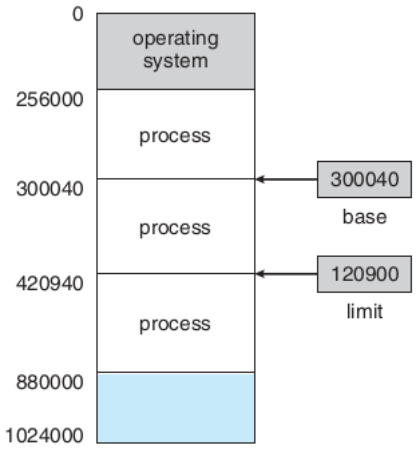
Osnovna memorija i registri, kao sastavni deo samog procesora su jedina memorija opšte namene kojoj CPU može direktno da pristupi. Postoje instrukcije koje koriste memorijske adrese kao argumente, ali nijedna od njih ne koristi adrese sa diska. Stoga sve instrukcije koje se izvršavaju i svi podaci koji se koriste u njima moraju biti smešteni u nekom od memorijskih modula kojima se direktno pristupa. Ako podaci nisu u memoriji, moraju se prebaciti tamo pre nego što CPU može da radi na njima.

Registri koji su ugrađeni u CPU su uglavnom dostupni unutar jednog ciklusa takta CPU-a. Većina procesora može dekodovati instrukcije i vršiti jednostavne operacije nad registrima brzinom jedne ili više operacija po sistemskom taktu. Isto se ne može reći za glavnu memoriju, kojoj se pristupa transakcijom na memorijskoj magistrali. Dovršenje pristupa memoriji može da traje mnogo ciklusa takta procesora. U takvim slučajevima procesor obično treba da zastane (*memorijski zastoj*), jer nema podatke potrebne za dovršavanje instrukcije koju izvršava. Ova situacija je nepodnošljiva usled učestalog pristupa memoriji. Rešenje je dodavanje brze memorije između CPU-a i glavne memorije, obično na samom CPU čipu u cilju bržeg pristupa. Keš memorija je opisana detaljno na prethodnim predavanjima, a korišćenjem keša ugrađenim u CPU, hardver automatski ubrzava pristup memoriji bez ikakve kontrole operativnog sistema.

Međutim, nije dovoljno povećati relativnu brzinu pristupa fizičkoj memoriji, već je veoma značajno osigurati ispravnu funkcionalnost. Za pravilan rad sistema moramo obezbediti zaštitu operativnog sistema od pristupa od strane korisničkih procesa. Dodatno, na sistemima sa više korisnika moramo dodatno zaštititi korisničke procese jedne od drugih. Ovu zaštitu mora da obezbedi hardver jer operativni sistem obično ne interveniše između CPU-a i osnovne memorije (zbog rezultujuće degradacije performansi). Hardver implementira ovu funkcionalnost na nekoliko različitih načina, kao što ćemo videti u toku ovog predavanja. Ovde smo izdvojili jednu moguću implementaciju.

Prvo moramo biti sigurni da svaki proces ima zaseban memorijski prostor. Odvojeni memorijski prostor za svaki proces štiti procese jedne od drugih i od suštinskog je značaja za učitavanje više procesa u memoriju, u cilju njihovog istovremenog izvršavanja. Da bismo odvojili memorijske prostore, potrebno nam je da odredimo raspon legalnih adresa kojima proces može pristupiti i da osiguramo da proces može pristupiti samo adresama iz tog opsega. Ovu zaštitu možemo pružiti upotrebom dva registra, baznog registra (eng. *base register*) i graničnog registra (eng. *limit register*), kao što je prikazano na slici 1.

U baznom registru nalazi se najniža validna adresa fizičke memorije kojoj proces može da pristupa, dok granični registar određuje veličinu opsega dostupne memorije. Na primer, ako bazni registar sadrži 0x300040, a granični registar je 0x120900, program može legalno pristupiti svim adresama od 0x300040 do 0x420939. Zaštita memorijskog prostora ostvaruje se tako što hardver CPU-a upoređuje svaku adresu generisanu u korisničkom režimu sa baznim i graničnim registrom. Svaki pokušaj programa koji se izvršava u korisničkom režimu da pristupi memoriji operativnog sistema ili memoriji drugih korisnika rezultira



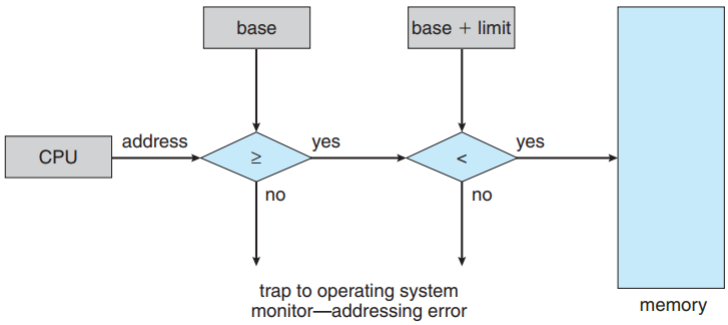
Slika 1: Logički adresni prostor definisan baznim i graničnim registrom

izuzetkom (*trap*) u operativnom sistemu, koji će takav pokušaj tretirati kao fatalnu grešku (slika 2).

Ova šema sprečava korisničkog programa da (slučajno ili namerno) menja kod ili strukture podataka bilo operativnog sistema bilo drugih korisnika.

Baznom i graničnom registru može se pristupati samo u operativnom sistemu koji koristi posebnu privilegovanu instrukciju u tu svrhu. Pošto se privilegovane instrukcije mogu izvršavati samo u režimu kernela, a budući da se u kernel režimu samo operativni sistem izvršava, samo operativni sistem može pristupati baznom i graničnom registru. Ova šema omogućava operativnom sistemu da menja vrednost registara, ali sprečava korisničke programe da menjaju sadržaj registara.

Operativnom sistemu, koji se izvršava u režimu kernela, omogućen je neo-



Slika 2: Hardverska protekcija adresa pomoću baznog i graničnog registra

graničen pristup kako memoriji operativnog sistema tako i memoriji korisnika. Ova odredba omogućava operativnom sistemu da učitava korisničke programe u korisničku memoriju, da ukloni te programe u slučaju grešaka, da pristupa i menja parametre sistemskih poziva, da izvršava U/I operacije koje čitaju iz ili pišu u korisničku memoriju i pruža mnoge druge servise. Na primer, operativni sistem u slučaju sistema sa višestrukim procesima mora izvršiti zamene konteksta, čuvajući stanje jednog procesa iz registara u glavnu memoriju pre učitavanja konteksta sledećeg procesa iz glavne memorije u registre.

2.2 Povezivanje adresa (*Address binding*)

Obično se program nalazi na disku kao binarna izvršna datoteka. Da bi se izvršio, program mora biti učitan u memoriju i smešten u proces. U zavisnosti od upravljanja memorijom koja se koristi, proces se može kretati između diska i memorije tokom njegovog izvršavanja. Procesi na disku koji čekaju da se prenesu u memoriju kako bi se izvršavali formiraju *ulazni red procesa*.

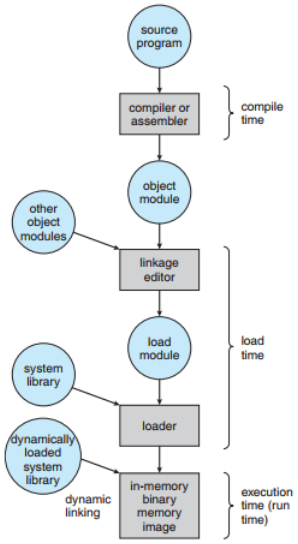
Uobičajena procedura u sistemu sa jednim zadatkom koji se izvršava (*single-task*) je odabir jednog od procesa iz ulaznog reda procesa i učitavanje tog procesa u memoriju. Dok se proces izvršava, on pristupa instrukcijama i podacima iz memorije. Na kraju, kada se proces terminira, njegov memorijski prostor se oslobađa i proglašava se dostupnim.

Većina sistema omogućava korisničkom procesu da bude smešten u bilo kojem delu fizičke memorije. Prema tome, iako adresni prostor računara može početi na adresi 0x00000000, prva adresa korisničkog procesa ne mora biti 0x00000000. Kasnije ćemo videti kako se korisnički program, odnosno proces, zaista smešta u fizičku memoriju. U većini slučajeva, korisnički program prolazi kroz nekoliko koraka - od kojih neki mogu biti opcioni - pre nego što postane spreman za izršavanje (slika 3).

Tokom tih koraka adrese mogu biti predstavljene na različite načine. Adrese u izvornom programu su uglavnom simboličke (poput naziva varijable - npr *count*). Kompajler obično ove simboličke adrese vezuje za izmestive (eng. *relocatable*) adrese (poput „14 bajta od početka ovog modula“). Linker zauzvrat vezuje ove izmestive adrese uz *apsolutne adrese* (kao što je 0x74014). Svako povezivanje je, u ovom smislu, mapiranje iz jednog adresnog prostora u drugi. Klasično, povezivanje instrukcija i podataka uz memorijske adrese može se obaviti u bilo kojem koraku:

- Vreme kompajliranja (*Compile time*). Ako se u trenutku kompajliranja zna gde će proces biti smešten u memoriji, tada se može odmah generisati kod sa apsolutnim adresama. Na primer, ako se zna da će korisnički proces biti smešten u memoriju počevši od lokacije R, tada će kod generisan od strane kompajlera početi od te adrese pa na dalje. Ako se u nekom kasnijem trenutku početna lokacija promeni, biće neophodno da se ovaj kod ponovo kompilira. MS-DOS programi .COM formatu dobijali su fizičke adrese u vreme kompajliranja.

- Vreme učitavanja (*Load time*). Ako u vreme kompajliranja nije poznato



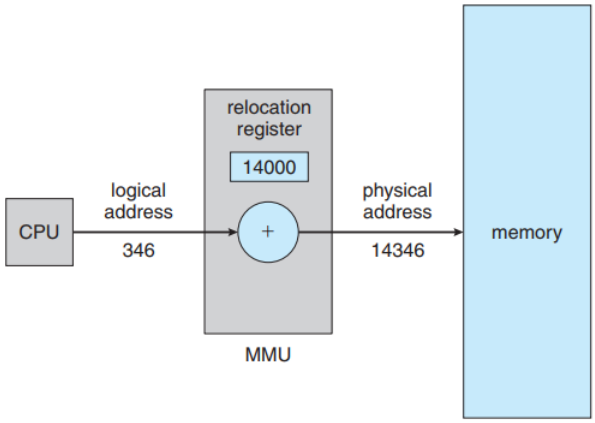
Slika 3: Pripremanje procesa za izvršavanje u nekoliko koraka

gde će proces boraviti u memoriji, tada kompajler mora generisati izmestiv kod (kod koji se može premeštati). U ovom slučaju, konačno povezivanje se odlaže do vremena učitavanja. Ako se početna adresa promeni, potrebno je samo ponovo učitati korisnički kod da bismo uzeli u obzir ovu promenu.

- Vreme izvršavanja (*Execution time*). Ako se proces može premestiti tokom njegovog izvršavanja iz jednog memorijskog segmenta u drugi, tada vezanje mora biti odloženo do trenutka pokretanja. Da bi se ovo moglo desiti, neophodno je imati na raspolaganju poseban hardver, o čemu će biti reči kasnije. Većina operativnih sistema opšte namene koristi ovu metodu.

2.3 Logičke i fizičke adrese

Adresa koju generiše CPU obično se naziva logičkom adresom, dok se adresa koju vidi memorijska jedinica, odnosno ona koja je učitana u memorijski adresni registar, obično naziva fizička adresa. Povezivanje adresa za vreme kompajliranja ili za vreme učitavanja generišu identične logičke i fizičke adrese. Međutim, shema vezivanja adresa za vreme izvršavanja rezultuje različitim logičkim i fizičkim adresama. U ovom slučaju logičku adresu obično nazivamo *virtualnom adresom*. U nastavku teksta koristimo ravnomerno termine *logička adresa* i *virtuelna adresa*. Skup svih logičkih adresa generiranih programom je *logički adresni prostor*. Skup svih fizičkih adresa koje odgovaraju ovim logičkim adresama je *fizički adresni prostor*. Stoga se, kod vezivanja adresa za vreme izvršavanja razlikuju logički i fizički adresni prostori.



Slika 4: Dinamička relokacija korišćenjem registra relokacije

Mapiranje virtualnih u fizičke adrese tokom vremena izvršavanja vrši se pomoću specijalnog hardverskog uređaja koji se zove *jedinica za upravljanje memorijom* (eng. *Memory Management Unit*). Kao što ćemo videti kasnije, postoji mnogo način na koje se ovo mapiranje može izvršiti (odeljci 4 do 6). Za sada ilustrujemo ovo mapiranje jednostavnom MMU šemom koja predstavlja generalizaciju šeme sa baznim i graničnim registrom opisanim u odeljku 2.1. Bazni registar sada se naziva registar relokacije (*relocation register*). Vrednost u registru relokacije dodaje se svakoj adresi koju generiše korisnički proces u vreme slanja adrese ka memoriji (slika 4). Na primer, ako je vrednost registra relokacije na 0x14000, pokušaj korisnika da adresira lokaciju 0 dinamički se premešta na lokaciju 0x14000, dok je pristup lokaciji 0x346 mapiran na lokaciju 0x14346.

Korisnički program nikada ne vidi prave fizičke adrese. Program može da kreira pokazivač na lokaciju 346, da ga sačuva u memoriji, manipuliše njime i upoređuje sa drugim adresama - sve kao broj 346. Samo kada se koristi kao memorijska adresa (recimo u indirektnim load/store instrukcijama) ova vrednost će biti promenjena u skladu sa registrom relokacije. Dakle, korisnički program radi sa logičkim adresama, a hardver za memorijsko mapiranje pretvara logičke adrese u fizičke adrese. O ovom obliku vezivanja za vreme izvršavanja raspravljalo se u odeljku 2.1. Konačna lokacija referencirane memorijske adrese nije poznata sve dok se referenca ne prosledi.

Sada imamo dve različite vrste adresa: logičke adrese (u opsegu od 0 do max) i fizičke adrese (u opsegu $R + 0$ do $R + max$ za vrednost bazne adrese R). Korisnički program generiše samo logičke adrese i misli da se proces izvršava na lokacijama od 0 do max. Međutim, kao što smo videli, ove logičke adrese moraju se preslikati na fizičke adrese pre nego što budu upotrebljene. Koncept logičkog adresnog prostora koji je povezan sa zasebnim fizičkim adresnim prostorom ključan je za pravilno upravljanje memorijom.

2.4 Dinamičko učitavanje

U dosadašnjoj diskusiji, bilo je podrazumevano da se kompletan program, zajedno sa svim svojim podacima, nalazi u fizičkoj memoriji, da bi se proces izvršavao. Samim tim, veličina procesa je limitirana veličinom fizičke memorije u sistemu. Kako bismo dobili bolju iskorišćenost memorijskog prostora, koristimo takozvano *dinamičko učitavanje* (eng. *dynamic loading*). Kod dinamičkog učitavanja, procedura nije učitana u memoriju sve dok se ne pozove. Sve procedure se nalaze na disku u formatu spremnom za relokaciju. Glavni program se učitava u memoriju i izvršava se. Kada procedura treba da pozove drugu proceduru, pozivajuća procedura prvo proverava da li je druga procedura učitana. Ako nije, kreće se sa relociranjem, učitavanjem i povezivanjem, kako bi se tražena procedura premestila u memoriju. Takođe, ažuriraju se tabele adresa dodeljene svakom programu kako bi ove izmene imale efekta. Tek nakon toga se kontrola prepušta pozvanoj proceduri.

Prednost ovakvog dinamičkog učitavanja je u tome što je procedura učitana samo kada je zaista potrebna. Ovaj metod je posebno koristan kada postoje veliki blokovi koda, pozivani samo u retkim situacijama, kao što su rutine za obradu grešaka. U ovom slučaju, iako je veličina programa velika, jedan njen deo koji je učitan i koji se izvršava, može biti mnogo manji.

Ovakvo dinamičko učitavanje ne zahteva specijalnu podršku od strane operativnog sistema. Odgovornost je korisnika da svoje programe dizajniraju na način koji će omogućiti da se iskoristi prednost ovakvog pristupa. Operativni sistem može pomoći programerima, sa druge strane, obezbeđujući biblioteke i rutine koje implementiraju dinamičko učitavanje.

2.5 Dinamičko povezivanje i deljene biblioteke

Dinamički povezane biblioteke su sistemske biblioteke koje se povezuju sa korisničkim programima kada se program pokreće (slika 3). Neki operativni sistemi podržavaju samo *statičko povezivanje*, kod kojeg se sistemske biblioteke tretiraju kao obični objektni moduli i koji se kombinuju zajedno sa osnovnim kodom kako bi se generisala binarna izvršna datoteka programa. Dinamičko povezivanje, nasuprot tome, je sličnije dinamičkom učitavanju. Ipak, kod njega je povezivanje odloženo je do trenutka izvršavanja. Ovaj metod se obično koristi sa sistemskim bibliotekama, kao što je biblioteka procedura (funkcija) datog programskog jezika. Ukoliko se ne bi koristilo dinamičko povezivanje, svaki program na datom sistemu bi morao da uključi u binarnu datoteku programa kopiju biblioteke programskog jezika (ili minimalno procedura koje se pozivaju u samom programu). Ovakav zahtev bi nepotrebno trošio kako prostor na disku tako i osnovnu memoriju.

Kod dinamičkog povezivanja, stub je dodat u izvršnom kodu na mestu svake reference na proceduru iz biblioteke. Stub je mali deo koda koji daje indikaciju kako da se locira odgovarajuća rutina, ukoliko ona nije dostupna. Kada se on pokrene, najpre proverava da li je tražena procedura učitana u memoriju. Ako nije, program je učitava u memoriju, a u oba slučaja (i ako jeste učitana)

stub zamenjuje sebe sa adresom na kojoj se nalazi zahtevana procedura. Kao rezultat, sledeći put kada određeni kodni segment bude pozvan, procedura iz biblioteke se pokreće direktno, ne dodajući nikakvo kašnjenje niti dodatnu „cenu“ kao posledicu korišćenja dinamičkog povezivanja. Ovaj pristup omogućava da svi procesi u sistemu koji koriste biblioteku programskog jezika izvršavaju istu kopiju koda u okviru biblioteke.

Ova osobina može dodatno biti proširena na ažuriranja biblioteke (eng. *library updates*), nastale kao npr posledica rešavanja bagova u kodu. Biblioteka može biti zamenjena novijom verzijom i svi programi koji se referenciraju na biblioteku automatski će koristiti novu verziju biblioteke. Bez dinamičkog povezivanja, svi ti programi morali bi da budu ponovo povezivani, kako bi koristili novo-ažuriranu biblioteku. U cilju sprečavanja programa da greškom pokrenu novu, ne-kompatibilnu verziju biblioteke, informacije o verziji su uključene kako u program, tako i u samu biblioteku. Više od jedne verzije biblioteke može biti učitano u memoriju, a svaki program će koristiti svoju informaciju o verziji kako bi ustanovio koju kopiju biblioteke da koristi. Verzije sa malim izmenama obično zadržavaju isti broj verzije, dok verzije sa većim izmenama povećavaju broj verzije. Na taj način, samo programi koji su kompajlirani sa novijom verzijom biblioteke, biće pod uticajem bilo koje ne-kompatibilnosti i izmene uključene u novu verziju biblioteke. Svi ostali programi, povezivani pre nego što je nova biblioteka instalirana, nastaviće nesmetano da koriste stare verzije biblioteke. Ovaj sistem poznat je i kao *deljene biblioteke* (eng. *shared libraries*).

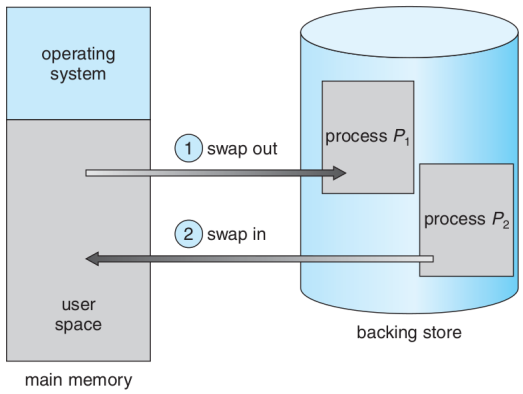
Nasuprot dinamičkom učitavanju, dinamičko povezivanje i deljene biblioteke, generalno, zahtevaju intervenciju operativnog sistema. Ako su procesi u memoriji zaštićeni jedan od drugog, onda jedino operativni sistem može proveriti da li se zahtevana procedura nalazi u okviru memorijskog prostora drugog procesa ili, dozvoliti višestrukim procesima da pristupaju istim adresama u okviru memorije. Pričaćemo više o ovom konceptu kada budemo pričali o korišćenju stranica u poglavlju 6.

3 Zamena (eng. *Swapping*)

Proces mora biti u memoriji da bi se izvršavao, kao što smo rekli. Međutim, proces može biti privremeno uklonjen (zamenjen) iz memorije u sekundarnu memoriju i kasnije vraćen kako bi nastavio sa izvršavanjem (slika 5). Zamena omogućava da ukupan adresni prostor procesa u sistemu premaši dostupnu fizičku memoriju, na taj način uvećavajući stepen multi-programiranja sistema.

3.1 Standardna zamena

Standardna zamena uključuje premeštanje procesa iz osnovne memorije u sekundarnu memoriju. Sekundarna memorija je, najčešće brzi disk koji mora biti dovoljno velik da smesti sve kopije svih memorijskih blokova svih korisnika, a mora obezbediti direktan pristup ovim blokovima memorije. U cilju toga, sistem održava *red* koji se sastoji od procesa čiji se memorijske kopije nalaze



Slika 5: Zamena dva procesa korišćenjem diska kao sekundarne memorije

u sekundarnoj memoriji, a spremni su za izvršavanje. Uvek kada CPU planer odluči da pokrene neki proces, poziva dispečera. On, zauzvrat, proverava da li je dati proces iz reda čekanja u memoriji. Ako nije i ako nema slobodnih regiona u memoriji, dispečer tada uklanja neki od procesa koji se nalaze u osnovnoj memoriji, a na njegovo mesto postavlja novi proces. Nakon toga, učitava njegove registre i prepušta kontrolu nad CPU-om odabranom procesu.

Vreme zamene u sistemu koji omogućava ovakvu zamenu je veoma dugačko. Kao ilustracija, pretpostavimo da je korisnički proces 100MB i da sekundarna memorija omogućava prenos podataka brzinom 50MB/s. Tada, prenos procesa iz osnovne memorije u sekundarnu traje 2s. Obzirom na to da moramo i drugi proces iz sekundarne memorije prebaciti u osnovnu, ukupno trajanje zamene je 4s, pri čemu ignorišemo dodatno kašnjenje u vezi sa upisom i čitanjem sa diska.

Najveći deo vremena zamene je vreme prenosa podataka. Ukupno vreme prenosa je direktno proporcionalno veličini memorije koja se zamenjuje. Ako imamo sistem sa 4GB osnovne memorije, od kojih 1GB zauzima operativni sistem, maksimalna veličina korisničkog procesa u memoriji je 3GB. Ipak, većina korisničkih procesa je znatno manja od toga i, kao što smo rekli, u slučaju procesa od 100MB, on će biti zamenjen za 2s, nasuprot 60s potrebnih da se zameni proces od 3GB. Očigledno, bilo bi veoma korisno da znamo koliko tačno memorije zahteva korisnički proces, umesto da uzimamo u obzir koliko bi mogao da zahteva. Tada bismo mogli da zamenjujemo samo ono što se zaista koristi, što će znatno smanjiti vreme zamene. Da bi ovakav metod mogao da se koristi, korisnik mora informisati sistem o izmenama zahteva za memorijom. Proces sa dinamičkim alociranjem memorije će, u tu svrhu, koristiti sistemske pozive *release_memory*() i *release_memory*() kako bi obavestio operativni sistem o promeni potrebe za memorijom.

Zamena je uslovljena i drugim faktorima. Ako želimo da zamenimo proces, moramo biti sigurni da je on potpuno besposlen u tom trenutku. Ovde se posebno vodi računa o eventualnim U/I zahtevima na koje proces eventualno

čeka. Recimo da proces čeka na neku U/I operaciju u momentu kada želimo da ga zamenimo kako bismo oslobodili memoriju. Ukoliko U/I periferija asinhrono pristupa korisničkoj memoriji i odgovarajućim U/I baferima, tada proces ne može biti zamenjen. Kao primer, posmatramo situaciju u kojoj U/I operacija čeka jer je uređaj zauzet. Ako zamenimo proces P_1 procesom P_2, U/I operacija može tada pokušati da koristi memoriju koja sada pripada procesu P_2. Postoje dva rešenja ovog problema: nikada ne dozvoli zamenu procesa koji čeka na U/I operaciju, ili dozvoli U/I operacijama da pristupaju samo baferima u okviru operativnog sistema. Transferi između bafera u okviru operativnog sistema i memorije procesa mogu da se izvrše kada se proces zameni nazad. Ovakvo *dvostruko baferovanje* očigledno je redundantno, jer imamo dodatno kopiranje podataka iz kernel memorije u memoriju korisničkog procesa pre nego što proces može da ih koristi.

Standardna zamena, kao takva, se ne koristi u modernim operativnim sistemima. Ona zahteva previše vremena za zamenu, kao što smo videli, i omogućava premalo vremena za izvršavanje da bi uopšte mogla biti razmatrana kao rešenje za upravljanje memorijom. Modifikovane verzije zamene, ipak, postoje na većini sistema, uključujući Unix, Linux i Windows. U jednoj, tipičnoj varijaciji, zamena je podrazumevano onemogućena, ali će se aktivirati ako količina slobodne memorije (dostupne za korišćenje od strane operativnog sistema ili korisničkih procesa) padne ispod određenog praga. Sa druge strane, zamena se opet onemogućava u momentu kada se količina slobodne memorije uveća dovoljno. Druga varijanta zamene se odnosi na zamenu delova procesa, pre nego celih procesa, kako bi se smanjilo vreme zamene. Tipično, ovakva forma zamene koristi se zajedno sa virtualnom memorijom i o tome ćemo pričati detaljnije na narednom predavanju.

3.2 Zamena u mobilnim sistemima

Iako većina operativnih sistema za PC računare i servere podržava neku modifikovanu verziju zamene, mobilni sistemi tipično ne podržavaju zamenu u bilo kojoj formi. Mobilni uređaji uglavnom kao stalnu memoriju koriste fleš memoriju, umesto hard diskova sa puno prostora za skladištenje podataka. Jedan od razloga zašto se ne dozvoljava zamena je upravo problem sa memorijskim prostorom. Dodatni razlozi se odnose na ograničen broj upisa koje fleš memorije tolerišu pre nego što postanu nepouzdane, kao i lošiji propusni opseg kod transfera između osnovne memorije i fleš memorije kod mobilnih uređaja.

Umesto korišćenje zamene, kada količina slobodne memorije padne ispod određenog praga, Apple-ov iOS traži od aplikacija da dobrovoljno oslobode alociranu memoriju. Podaci koji se samo čitaju (npr. kod) se uklanja iz sistema, a kasnije će ponovo biti učitan sa fleš memorije ako bude potrebe. Podaci koji su modifikovani (kao npr. stek) se nikada ne uklanjaju. Ipak, bilo koja aplikacija koja ne oslobodi dovoljno memorije, može biti terminirana od strane operativnog sistema.

Android takođe ne podržava zamenu i umesto nje koristi sličnu strategiju kao iOS. Android može terminirati proces ako je nedovoljno slobodne memorije

dostupno. Međutim, pre terminiranja procesa, Android pamti trenutno stanje aplikacije u fleš memoriji, kako bi kasnije mogao brzo i neprimetno da rekonstruiše aplikaciju tako da korisnik ni ne primeti da je proces u međuvremenu bio terminiran.

Zbog gore navedenih ograničenja, dizajneri aplikacija za mobilne sisteme moraju pažljivo da alociraju i oslobađaju memoriju, kako bi obezbedili da njihova aplikacija ne koristi previše memorije ili nema problem u vidu „curenja memorije“. Sa druge strane, i iOS i Android podržavaju korišćenje stranica (poglavlje 6), tako da oba operativna sistema imaju mogućnost upravljanja memorijom.

4 Kontinualno alociranje memorije

Osnovna memorija mora da omogući smeštanje kako operativnog sistema tako i korisničkih procesa. Stoga, moramo alocirati memoriju na najefikasniji mogući način. U ovom poglavlju ćemo najpre objasniti kontinualno alociranje memorije, metod koji se među prvima koristio.

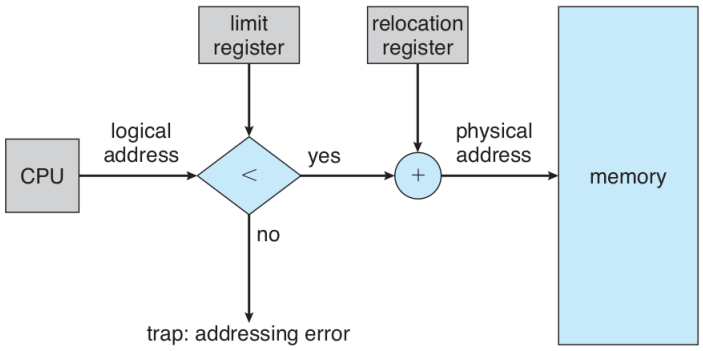
Memorija je, tipično, podeljena u dve particije: jedna u koju je smešten operativni sistem i druga koja se koristi za korisničke procese. Operativni sistem može biti smešten bilo u „nižu“ ili „višu“ memoriju, ali osnovni faktor koji utiče na tu odluku vezan je za lokaciju vektora prekida. Obzirom da su vektori prekida uglavnom smešteni na niskim memorijskim lokacijama, programeri obično smeštaju i operativni sistem u „nižu“ memoriju, pa ćemo takvu situaciju razmatrati u ovom poglavlju. Ipak, veoma slična implementacija bi bila i u slučaju da je drugačiji odabir memorijske particije rezervisane za operativni sistem.

Tipično, želimo da nekoliko procesa bude smešteno u memoriji istovremeno. Da bi to bilo ostvarivo, moramo da razmotrimo alokaciju memorije procesima koji se nalaze u redu procesa i čekaju da im bude dodeljena memorija. Kod kontunualne alokacije memorije, svaki proces je smešten u jedinstvenom bloku memorije koji je smešten odmah do susednog bloka dodeljenog drugom procesu.

4.1 Zaštita memorije

Pre nego što razmotrimo detaljnije samu alokaciju, moramo najpre prodiskutovati problem u vezi sa zaštitom memorije. Možemo sprečiti proces da pristupa memoriji koju ne poseduje kombinujući dve ideje predstavljene ranije. Ukoliko imamo sistem sa registrom relokacije (predstavljenim u poglavlju 2.3), zajedno sa graničnim registrom, postići ćemo naš cilj. Registar relokacije sadrži vrednost najniže fizičke adrese, dok granični registar sadrži opseg logičkih adresa (na primer registar relokacije 0x100040, a granični registar 0x74600). Svaka logička adresa mora da bude u opsegu definisanom sa ova dva registra. MMU tada mapira logičke adrese dinamički tako što dodaje vrednost sadržanu u registru relokacije, a ovakva mapirana adrese se prosleđuje memoriji (slika 6).

U momentu kada CPU planer odabere proces koji će se izvršavati, dispečer učitava u registar relokacije i granični registar odgovarajuće vrednosti kao deo zamene konteksta. Obzirom na to da se svaka adrese generisana od strane CPU



Slika 6: Hardverska podrška za metod sa registrom relokacije i graničnim registrom

proverava u odnosu na ova dva registra, možemo zaštititi ne samo operativni sistem, već i ostale korisničke programe i podatke i sprečiti njihovo modifikovanje od strane procesa koji se izvršava.

Samim tim, shema sa registrom relokacije obezbeđuje efektivan način koji omogućava promenu veličine operativnog sistema dinamički. Ovakva fleksibilnost je poželjna u mnogim situacijama. Na primer, operativni sistem sadrži kod i bafer koji se koristi od strane upravljačkih programa uređaja (drajvera za date uređaje). Ukoliko se upravljački program za dati uređaj (ili bilo koji drugi servis operativnog sistema) ne koristi često, bilo bi poželjno da ga ne držimo (kako kod tako ni podatke) u memoriji, obzirom na to da bismo taj prostor mogli efikasnije da upotrebimo u druge svrhe. Ovakav kod se nekada naziva *tranzijentni* kod (eng. *transient code*) operativnog sistema, pošto „dolazi“ i „nestaje“ po potrebi. Kao rezultat, korišćenje ovakvog tranzijentnog koda menja i veličinu operativnog sistema dok se izvršava.

4.2 Alokacija memorije

Jedan od najjednostavnijih metoda za alokaciju memorije je metod u kojem se memorije podeli na višestruke particije *fiksne veličine*. Svaka particija može da skladišti samo jedan proces. Sa aspekta toga, stepen multiprogramiranja je ograničen brojem particija. Kod ovakvog pristupa, kada imamo particiju koja je slobodna, proces se odabira iz ulaznog reda čekanja i učitava se u slobodnu particiju. Sa druge strane, kada se proces terminira, particija postaje dostupna za druge procese. Ovaj metod je originalno korišćen od strane IBM OS/360 operativnog sistema (koji se nazivao MFT) ali više nije u upotrebi.

Poboljšana verzija ovakvog pristupa koristi particije promenljive veličine (eng. *variable-partition*), a kod nje operativni sistem održava tabelu u kojoj se čuvaju informacije o tome koji delovi memorije su slobodni i dostupni, a koji su zauzeti. Inicijalno, sva memorija je dostupna za procese i predstavljena je

jednim velikim blokom slobodne memorije koji se zove šupljina (eng. *hole*). Tokom vremena, međutim, memorija će sadržati mnoštvo šupljina različitih veličina. Kako procesi dospevaju u sistem, smeštaju se u ulazni red čekanja. Operativni sistem prilikom alokacije memorije uzima u obzir memorijske zahteve od procesa, kao i količinu dostupne memorije, i u skladu sa tim određuje koji proces će biti smešten u memoriju. Nakon što mu se dodeli memorija, proces se učitava u tu memoriju i od tog momenta je spreman da se takmiči za procesorsko vreme sa ostalim procesima koji su spremni da se izvršavaju. Nakon što se proces terminira, on oslobađa svoju memoriju, koju operativni sistem opet može koristiti kako bi smestio neki drugi proces iz ulaznog reda čekanja.

U bilo kom trenutku, imamo listu dostupnih blokova različitih veličina i ulazni red procesa koji čekaju na memoriju. Operativni sistem može da sortira procese iz ulaznog reda u skladu sa algoritmom raspoređivanja, a memorija se alocira procesima sve dok, konačno, memorijski zahtevi jednog od procesa ne mogu biti ispunjeni. To se dešava jer nijedan blok slobodne memorije (šupljina) nije dovoljno velik da bi se u njega smestio proces. Operativni sistem tada može čekati dok se dovoljno velika šupljina ne pojavi, ili jednostavno može proveriti da li naredni proces iz ulaznog reda čekanja, sa manjim memorijskim zahtevima, možda može negde biti smešten.

Generalno, dostupni memorijski blokovi čine skup šupljina različitih veličina razbacanih po celoj memoriji. Kada proces pristigne i zahteva memoriju, sistem pretražuje taj skup šupljina kako bi pronašao onu koja je dovoljno velika za taj proces. Ako je ta šupljina prevelika, biće podeljena na dva dela: jedan deo će biti alociran procesu, dok se drugi vraća u skup šupljina. Kada se proces terminira, on oslobađa blok memorije, koji se onda opet smešta u skup šupljina. Ako je ova „nova“ šupljina susedna sa nekom već postojećom (u najboljem slučaju dve takve), susedne šupljine se spajaju u jednu veću. Tada operativni sistem mora da proveri da li postoji proces koji je čekao na memoriju i da li ova nova (eventualno rekombinovana sa drugim šupljinama) šupljina može da zadovolji memorijske zahteve bilo kog od procesa koji čekaju.

Gore opisana procedura je specijalan slučaj generalnog problema dinamičke alokacije memorije, koji se odnosi na zadovoljavanje zahteva veličine n, u skupu slobodnih šupljina. Postoji više rešenja ovoga problema, a najčešće korišćena su ***first-fit***, ***best-fit*** i ***worst-fit*** strategija za odabir šupljine iz skupa postojećih šupljina.

1. *First-fit* alokacija alocira prvu šupljinu koja je dovoljno velika. Pretraga za takvom može početi bilo od početka skupa, bilo od lokacije gde je prethodni prolaz kroz skup završio. Pretraga se prekida čim se pronađe slobodna šupljina koja je dovoljno velika.

2. *Best-fit* alokacija alocira najmanju šupljinu koja je dovoljno velika. Da bismo nju našli, moramo pretražiti kompletan skup, osim ukoliko je sama lista sortirana po veličini. Ovakva strategija rezultuje najmanjom preostalom šupljinom koja nastane kada se proces dodeli pronađenoj šupljini.

3. *Worst-fit* alokacija alocira najveću šupljinu. Opet, moramo pretražiti ceo

skup (celu listu), osim ako je lista sortirana po veličini. Ova strategija rezultuje najveću preostalu šupljinu, što može biti više korisno nego najmanja moguća preostala šupljina koju dobijamo primenom best-fit alokacije.

Simulacije su pokazale da su *first-fit* i *best-fit* alokacije bolje od *worst-fit* alokacije, u terminima utrošenog vremena i iskorišćenosti memorije. Od ova dva algoritma, nijedan nije očito bolji, ali je *first-fit*, generalno, brži.

4.3 Fragmentacija

I *first-fit* i *best-fit* strategije za memorijsku alokaciju pate problema poznatog kao ***eksterna fragmentacija***. Kako se procesi učitavaju i uklanjaju iz memorije, slobodan memorijski prostor se „razbija“ na više malih delova (šupljine različitih veličina). Eksterna fragmentacija je fenomen koji nastaje kada, iako postoji dovoljno dostupne memorije u sistemu da bi se ona alocirala nekom procesu, to se ne dešava jer ta memorija ne pripada *kontinualnom bloku memorije*, odnosno ne zauzima kontinualan memorijski prostor. Memorijski prostor je fragmentiran u veliki broj malih šupljina, a ovakav problem može imati ozbiljne posledice. U najgorem slučaju, mogao bi da nastane blok slobodne memorije koja ne može biti iskorišćena između memorijskih zona dodeljenih bilo kojim dvoma procesima. Da su, nasuprot tome, ove šupljine sačinjavale jednu veliku šupljinu, mogli bismo da pokrenemo nekolicinu novih procesa.

Tip strategije koji se koristi (first-fit ili best-fit strategija) može da utiče na količinu fragmentacije: u nekim sistemima first-fit se bolje ponaša, dok se na nekim drugim bolje ponaša best-fit. Još jedan faktor koji utiče na fragmentaciju odnosi se na „kraj“ memorije koja se alocira, odnosno, da li će preostala „neiskorišćena“ memorija da se nalazi na „vrhu“ ili na „dnu“ bloka koji je podeljen na dve particije.

Ipak, bez obzira na to koja se strategija koristi, i sa kojeg kraja se alocira memorija, eksterna fragmentacija će biti problem. U zavisnosti od ukupne količine memorije i prosečne veličine procesa, eksterna fragmentacija može predstavljati zanemariv ili nezanemariv problem. Statističke analize first-fit algoritma, na primer, pokazuju da, čak i sa određenim optimizacijama, na alociranih N blokova ide $0.5N$ blokova koji se gube kao rezultat fragmentacije. To znači da je jedna trećina memorije neupotrebljiva, a ova osobina je poznata i kao **pravilo 50%** (eng. *50-percent rule*).

Slično kao što može biti eksterna, fragmentacija takođe može biti i ***interna***. Ako razmatramo particije sa fiksnim veličinama, kod kojih je veličina šupljine 18 464 bajta, kada proces zahteva 18462 bajta memorije, ako mu se dodeli slobodan blok, preostala memorija će biti 2. Praćenje ovakve šupljine će biti skuplje sa stanovišta memorijskih resursa od same veličine šupljine. Generalni pristup koji bi trebao da izbegne ovaj problem jeste da se celokupna fizička memorija podeli na blokove fiksne veličine, a da se memorija koja se alocira računa kao broj ovakvih elementarnih blokova. Ovakvim pristupom, memorija alocirana za proces može biti donekle veća od zahtevane memorije (od strane procesa), a razlika ove dve veličine zove se ***interna fragmentacija***, obzirom da se odnosi

na neiskorišćenu memoriju koja je interna u okviru particije.

Sa druge strane, jedno rešenje problema eksterne fragmentacije je *sabijanje* (eng. *compaction*). Cilj je da se memorijski sadržaj preraspodeli tako da se sva slobodna memorija grupiše u jedan veliki blok slobodne memorije. Sabijanje, ipak, nije uvek moguće. Ako je relokacija statička i ako se povezivanje vrši u trenutku kompajliranja ili učitavanja, ona ne može biti korišćena. Nju je moguće vršiti samo ako je relokacija dinamička i ako se povezivanje vrši tokom izvršavanja programa. Ako se adrese relociraju dinamički, premeštanje zahteva samo pomeranje programa i podataka, nakon čega se menja bazni registar (registar relokacije), kako bi se koristile nove adrese. U slučajevima kada je sabijanje moguće, moramo odrediti cenu njene primene. Najjednostavniji algoritam za sabijanje odnosi se na pomeranje svih procesa ka jednom kraju memorije, pri čemu će sve šupljine biti automatski pomerane ka drugom, kreirajući jednu veliku šupljinu koja predstavlja slobodnu memoriju. Ipak, ovakav pristup može biti neprihvatljiv.

Drugi pristup rešavanju problema eksterne fragmentacije odnosi se na omogućavanje da fizički adresni prostor procesa bude ne-kontinualan, što opet omogućava procesu da mu se alocira slobodna fizička memorija kada god takva memorija postoji. Postoje dve tehnike koje pružaju ovakvo rešenje: segmentacija i straničenje. Često, ove dve tehnike mogu da budu i kombinovane.

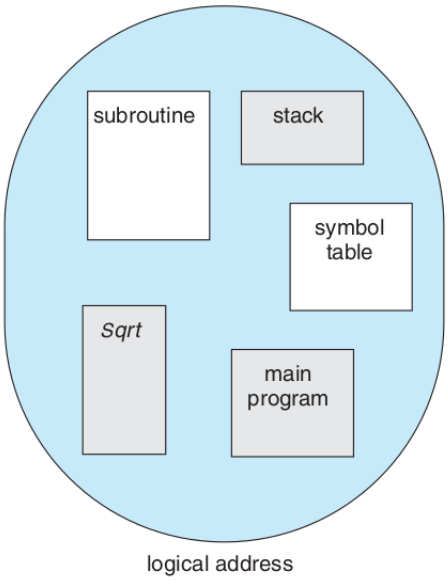
5 Segmentacija

Upravljanje memorijom u pogledu karakteristika fizičke memorije nije najzgodnije sa stanovišta operativnog sistema ili programera. Šta ako bi hardver obezbedio mehanizam za upravljanje memorijom koji bi mapirao memoriju kako je vide programeri u fizičku memoriju. Tada bi sistem imao više slobode u upravljanju memorijom, dok bi programeri mogli da ostanu u prirodnom programerskom okruženju. Segmentacija omogućava ovakav mehanizam.

5.1 Osnovna metoda

Ako pitate programera da li memoriju vidi kao linearan niz lokacija, od kojih neke sadrže kod a neke podatke, većina bi rekla „ne“. Nasuprot tome, programeri bi radije gledali na memoriju kao na kolekciju segmenata različitih veličina, bez potrebe za sortiranjem ili bilo kakvih slaganjem segmenata (slika 7)

Kada piše program, programer razmišlja o njemu kao glavnom programu, sa skupom procedura ili funkcija. Veoma verovatno tu su uključeni i stek, nizovi, objekti, promenljive, itd. Svaki od ovih modula je karakterisan svojim imenom. Programeri govore o „steku“, „biblioteci za rad sa matematičkim funkcijama“ i „glavnom programu“ bez brige o tome koje adrese u okviru memorije ti moduli zauzimaju. Njih ne zanima da li se stek nalazi „ispod“ ili „iznad“ *sqrt*() funkcije. Segmenti mogu varirati u pogledu veličine, a veličina svakog je inherentno određena ulogom tog segmenta u programu. Elementi unutar segmenta



Slika 7: Programerski pogled na program

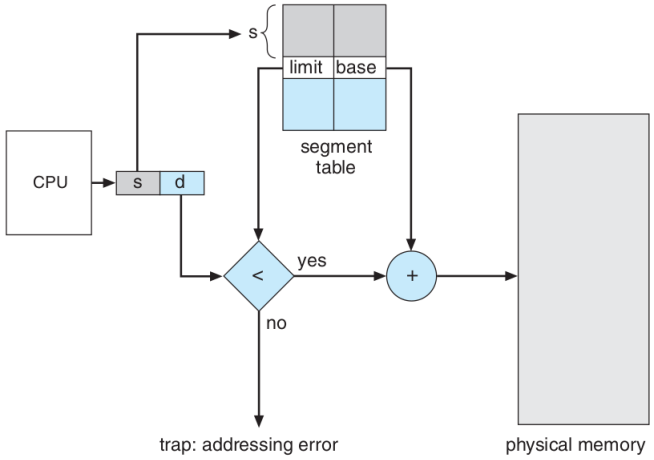
su karakterisani ofsetom u odnosu na početak segmenta: peti iskaz u programu, sedmi zapis u steku, peta instrukcija $sqrt()$ funkcije, itd.

Segmentacija je shema za upravljanje memorijom koja obezbeđuje podršku ovakvom pristupu. Logički adresni prostor je kolekcija segmenata, pri čemu svaki segment ima svoje ime i dužinu. Adresa specificira kako ime segmenta, tako i ofset u okviru segmenta. Radi jednostavnije implementacije, segmenti su numerisani i referencirani putem *broja segmenta*, pre nego naziva segmenta. Kao rezultat, logička adresa se sastoji od uređenog para

$$< broj - segmenta, ofset >$$

Normalno, kada se program kompajlira, kompajler automatski kreira segmente u skladu sa programom koji se kompajlira. C program može kreirati zasebne segmente za:

1. kod
2. globalne promenljive
3. heap memoriju odakle se vrši dinamička alokacija memorije
4. zaseban stek za svaku od niti izvršavanja
5. standardnu C biblioteku



Slika 8: Hardver za segmentaciju

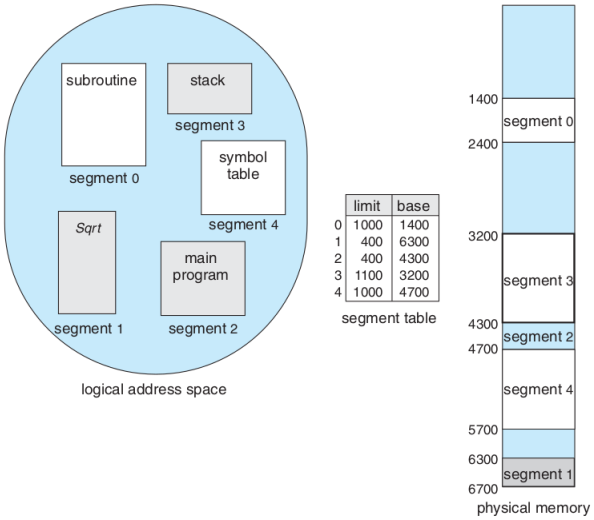
Bibliotekama povezanim sa programom za vreme kompajliranja mogu biti dodeljeni zasebni segmenti. Prilikom učitavanja, svi ovi segmenti će dobiti svoje brojeve segmenata.

5.2 Hardver za segmentaciju

Iako ovakav pristup omogućava programeru da se objektima u svom programu obraća putem dvodimenzionih adresa, fizičkoj memoriji se i dalje mora pristupati kao jednodimenzionom nizu lokacija. Stoga, moramo definisati način na koji će se implementirati mapiranje dvodimenzionih korisnički definisanih adresa u jednodimenzione fizičke adrese. Ovo mapiranje se vrši korišćenjem *tabele segmenata* (eng. *segment table*). Svaki upis u tabeli segmenata ima svoju *bazu segmenta* i *granicu segmenta* (eng. *segment base*, *segment limit*). Baza segmenta sadrži početnu fizičku adresu na kojoj počinje segment u memoriji, dok granica segmenta specificira dužinu segmenta.

Korišćenje tabele segmenata prikazano je na slici 8. Logička adresa se sastoji iz dva dela: broj segmenta s, i ofset u tom segmentu d. Broj segmenta se koristi kao indeks prilikom pristupa tabeli segmenata. Sa druge strane, ofset d logičke adrese mora da se nalazi između 0 i *granice segmenta*. Ukoliko to nije slučaj, generisaće se izuzetak (trap) ka operativnom sistemu (pokušaj pristupa logičkoj adresi koja je van granica segmenta). Kada je ofset legalan, on se dodaje na bazu segmenta pročitanu iz tabele segmenata, kako bi se generisala adresa zahtevane lokacije u fizičkoj memoriji. U svetlu toga, tabela segmenata je zapravo niz uređenih parova baza/granica, za svaki od segmenata.

Kao primer, razmatramo situaciju sa slike 9. U ovom primeru imamo pet segmenata numerisanih od 0 do 4. Segmenti su smešteni u fizičku memoriju kao što je prikazano na slici. Tabela segmenata sadrži zaseban unos za svaki



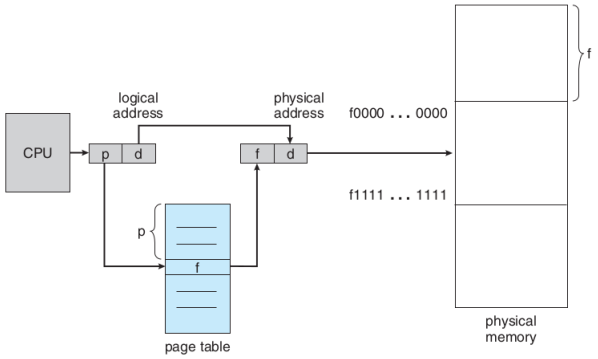
Slika 9: Primer segmentacije

segment, a sastoji se od početne adrese datog segmenta u fizičkoj memoriji (baza) i dužine segmenta (granica). Na primer, segment 2 je 400 bajta velik i počinje na lokaciji 4300. Stoga, referenca na 53 bajt segmenta 2 mapiraće se na lokaciju 4300+53=4353. Referenca na 852 bajt segmenta 3 će se mapirati na 3200 (osnova segmenta 3) + 852 = 4052. Referenca na bajt 1222 segmenta 0 rezultovaće izuzetkom u okviru operativnog sistema, obzirom na to da je segment 0 samo 1000 bajta velik.

6 Straničenje (*Paging*)

Segmentacija omogućava da prostor fizičkih adresa procesa ne mora da bude neprekidan. *Straničenje* predstavljaju još jedna šemu upravljanja memorijom koja nudi ovakvu prednost. Međutim, korišćenjem stranica izbegava se eksterna fragmentacija i potreba za sabijanjem, dok u slučaju segmentacije to nije slučaj. Takođe, korišćenje stranica rešava značajan problem smeštanja delova memorije različitih veličina u sekundarnu (stalnu) memoriju. Većina mehanizama za upravljanje memorijom korišćenih pre nastanka straničenja pogođeni su ovim problemom. Problem nastaje kada fragmenti koda ili podaci koji se nalaze u glavnoj memoriji moraju biti *zamenjeni* (*swapped out*), neophodno je pronaći odgovarajući prostor u okviru sekundarne memorije. Stalna memorija ima iste probleme vezane za fragmentaciju o kojima smo govori u vezi sa osnovnom memorijom, ali obzirom da je pristup stalnoj memoriji mnogo sporiji, sabijanje nije opcija.

Usled svojih prednosti u odnosu na ostale metode, straničenje u različitim



Slika 10: Hardver koji omogućava korišćenje stranica

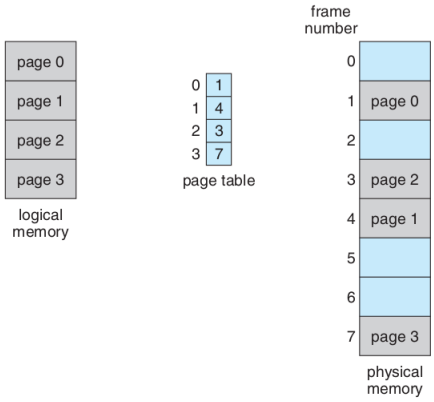
oblicima koristi se u većini operativnih sistema, počevši od onih za *mainframe* računare, do onih za pametne telefone. Korišćenje stranica je implementirano kroz međusobnu saradnju operativnog sistema i hardvera računara.

6.1 Osnovni metod

Osnovna metoda za straničenje uključuje deljenje fizičke memorije u blokove fiksne veličine koji se nazivaju *okviri* (eng. *frames*), a logičke memorije takođe u blokove fiksne veličine koji se nazivaju *stranice*. Kada se treba izvršiti neki proces, njegove *stranice* se učitavaju u proizvoljne dostupne okvire memorije, nakon čitanja sa njihove originalne lokacije (izvršna datoteka u okviru stalne memorije). Stalna memorija je takođe podeljena na blokove fiksne veličine, i to blokove koji su ili iste veličine kao okviri memorije ili veličine klastera više okvira memorije. Ova prilično jednostavna ideja ima sjajnu funkcionalnost uz ogroman uticaj na sistem u celini. Na primer, prostor logičkih adresa sada je potpuno odvojen od prostora fizičkih adresa, pa proces može imati logički 64-bitni adresni prostor iako sistem ima manje od 2^{64} bajta fizičke memorije.

Hardverska podrška za straničenje je prikazana na slici 10. Svaka adresa koju generiše CPU podeljena je u dva dela: broj stranice (p) i ofset u okviru stranice (d). Broj stranice koristi se kao indeks u tabeli stranica. Tabela stranica sadrži baznu adresu svake stranice u fizičkoj memoriji. Ova bazna adresa je kombinovana sa ofsetom stranice za definisanje adrese fizičke memorije koja se šalje memorijskoj jedinici. Model memorije koja koristi stranice prikazan je na slici 11.

Veličina stranice (kao i veličina okvira) je definisana hardverom. Veličina stranice je uvek stepena broja 2, a varira između 512 bajta i 1 GB po stranici, zavisno od arhitekture računara. Izbor stepena broja 2 kao veličine stranice znatno olakšava mapiranje logičke adrese u broj stranice i offset stranice. Ako je veličina logičkog adresnog prostora 2^m, a veličina stranice 2^n bajta, tada m - n viših bita logičke adrese označavaju broj stranice, a n donjih bita označavaju



Slika 11: Model korišćenja stranica - logičke i fizičke adrese

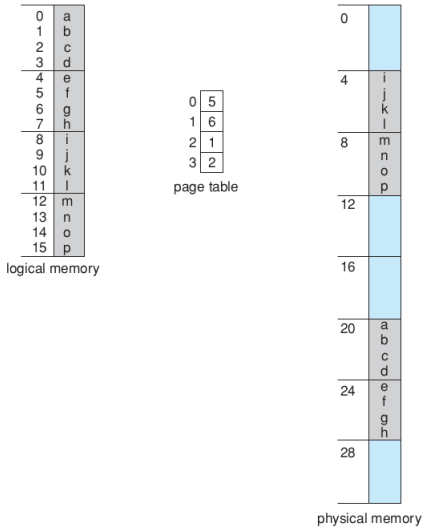
ofset stranice. Dakle, logička adresa je sledeća:

gde je p indeks u tabeli stranica, a d je ofset unutar stranice.

Kao konkretan (iako minimalan) primer, pogledajmo memoriju sa slike 12. Ovde, za logičke adrese važi da je n = 2 i m = 4. Koristeći stranicu veličine 4 bajta i fizičku memoriju od 32 bajta (8 stranica), prikazujemo kako izgled memorije sa stanovišta programera može da se preslika u fizičku memoriju. Logička adresa 0 je stranica 0, ofset 0. Indeksiranjem u tabeli stranica nalazimo da je stranica 0 u okviru 5. Dakle, logička adresa 0 preslikava se na fizičku adresu 20 [= (5 × 4) + 0]. Logička adresa 3 (strana 0, ofset 3) preslikava se na fizičku adresu 23 [= (5 × 4) + 3]. Logička adresa 4 je strana 1, ofset 0, a sudeći po tabeli stranica, stranica 1 se preslikava u okvir 6. Dakle, logička adresa 4 mapira se na fizičku adresu 24 [= (6 × 4) + 0]. Slično Logička adresa 13 mapira na fizičku adresu 9.

Straničenje, samo po sebi, je jedan oblik dinamične relokacije. Svaka logička adresa povezana je od strane hardver za straničenje na neku fizičku adresu. Upotreba straničenja slična je korišćenju tabele sa baznim registrima (ili registrima relokacije), pri čemu se jedan koristi za svaki okvir memorije.

Kada koristimo straničenje, nemamo problem sa eksternom fragmentacijom: bilo koji slobodni okvir može se dodeliti procesu kojem je potreban. Međutim, potencijalno ćemo imati internu fragmentaciju. Primetite da su okviri alocirani kao osnovne jedinice memorije. Ako se zahtevi za memorijom od strane nekog procesa ne podudaraju sa granicama stranice, poslednji dodeljeni okvir možda neće biti u potpunosti popunjen. Na primer, ako je veličina stranice 2,048 bajta, za proces od 72,766 bajtova biće potrebno 35 stranica plus 1,086 bajta. U tom slučaju, procesu će biti dodeljeno 36 okvira, što rezultuje internom

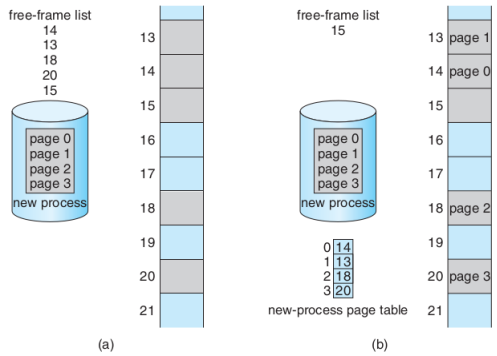


Slika 12: Primer jednostavne memorije sa stranicama

fragmentacijom od 2,048 - 1,086 = 962 bajta. U najgorem slučaju, za alokaciju potrebne memorije od strane nekog procesa, trebalo bi alocirati n stranica plus 1 bajt. Tada bi njemu bilo dodeljeno $n + 1$ okvira, što bi rezultovalo internom fragmentacijom gotovo celog okvira.

Ako je veličina procesa nezavisna od veličine stranice, očekujemo da će interna fragmentacija biti jednaka, u proseku, polovini stranice po procesu. Ovo razmatranje sugeriše da su poželjne male veličine stranica. Međutim, sa druge strane, dodatna cena koja se odnosi na svaki unos u tabeli stranica, smanjuje se kako se veličina stranice povećava. Takođe, disk kao U/I je efikasniji kada je količina podataka koja se prenosi veća. Uopšteno, veličine stranica su se vremenom povećavale u skladu sa uvećanjem procesa, podataka i osnovne memorije. Danas su stranice obično veličine između 4 KB i 8 KB, a neki sistemi podržavaju veće veličine stranica. Neki procesori i kerneli čak podržavaju više različitih veličina stranica. Na primer, Solaris koristi stranice od 8 KB i 4 MB, zavisno od podataka koji se smeštaju u stranice. Danas se razvija podršku za promenljivu veličinu stranica koja se može menjati u vremenu tokom izvršavanja operativnog sistema.

Najčešće, na 32-bitnom CPU-u svaki unos u tabeli stranica je 4 bajta širok, ali ta vrednost može varirati. 32-bitna vrednost unosa u tabeli stranica može pokazivati na jedan od 2^{32} okvira fizičke memorije. Ako je veličina okvira 4 KB (2^{12}), sistem sa 4-bajtnim unosima u tabeli stranica, samim tim, može adresirati 2^{44} bajta (ili 16 TB) fizičke memorije. Ipak, kao što ćemo videti u nastavku, uvodićemo dodatne informacije koje se moraju čuvati u tabeli stranica. Te informacije smanjuju broj bita koji su dostupni za adresiranje okvira. Kao posledica, sistem sa 32-bitnim unosima u tabeli stranica može adresirati manje



Slika 13: Slobodni okviri (a) pre i (b) posle alokacije za novi proces

fizičke memorije od mogućeg maksimuma. Sa druge strane, 32-bitni CPU koristi 32-bitne adrese, što znači da logički adresni prostor datog procesa može biti samo 2^{32} bajta (4 TB) velik. Međutim, korišćenje tabele stranica omogućava upotrebu fizičke memorije čiji je kapacitet veći od adresnog prostora koji se može adresirati od strane takvog procesora.

Kada proces koji treba da se izvrši stigne u sistem, ispituje se njegova veličina, izražena u stranicama. Svaka stranica procesa zahteva jedan okvir memorije. Prema tome, ako proces zahteva n stranica, mora da bude dostupno najmanje n okvira u memoriji. Ako je dostupno n okvira, oni će se dodeliti ovom dostiglom procesu. Prva stranica procesa se učitava u jedan od dodeljenih okvira, a broj okvira se stavlja u tabelu stranica za ovaj proces (na indeksu koji odgovara broju te stranice). Sledeća stranica je učitana u drugi okvir, njen broj okvira je postavljen u tabelu stranica, i tako dalje (slika 13).

Važan aspekt straničenja je jasno razdvajanje programerskog viđenja memorije od stvarne fizičke memorije. Programer vidi memoriju kao jedan jedinstven memorijski prostor, koji sadrži samo jedan program. Zapravo, u stvarnosti, korisnički program je raštrkan po fizičkoj memoriji, koja istovremeno sadrži i ostale programe koji se izvršavaju. Razlika između prikaza memorije sa stanovišta programera i stvarne fizičke memorije omogućena je specifičnim hardverom za transliranje adresa. Logičke adrese prevode se u fizičke adrese. Ovo mapiranje je skriveno od programera i kontroliše ga operativni sistem. Izuzetno je važno primetiti da ovakvim mapiranjem korisnički proces, po definiciji, ne može pristupiti memoriji koju ne poseduje. Proces može pristupiti samo memoriji upisanoj u tabeli stranica, a tabela stranica uključuje samo one stranice koje proces poseduje.

Budući da operativni sistem upravlja fizičkom memorijom, on mora biti svestan detalja u vezi sa dodeljivanjem fizičke memorije: koji su okviri dodeljeni, koji okviri su dostupni, koliko ukupnih okvira postoji, i tako dalje. Te se informacije obično čuvaju u strukturi podataka koja se naziva *tabela okvira* (eng. *frame table*). Tabela okvira ima jedan upis za svaki fizički okvir memorije, koji

pokazuje da li je odgovarajući okvir dostupan (slobodan) ili dodeljen i, ako je dodeljen, kojoj stranici kog procesa ili kojih procesa je dodeljen.

Pored toga, operativni sistem mora biti svestan da korisnički procesi rade u korisničkom prostoru, a sve logičke adrese moraju se preslikati da bi se generisale fizičke adrese. Ako korisnik pozove sistemski poziv (na primer U/I zahtev) i prosledi adresu kao parametar (na primer, adresu bafera), ta adresa mora biti translirana da bi se generisala ispravna fizička adresa. Operativni sistem održava kopiju tabele stranica za svaki proces, baš kao što održava i kopiju programskog brojača i sadržaja ostalih registara. Ova kopija koristi se za transliranje logičkih adresa u fizičke adrese kad god operativni sistem mora preslikati logičku adresu u fizičku adresu. Takođe, ovu tabelu koristi i CPU dispečer za popunjavanje hardverske tabele stranica u trenutku kada treba dodeliti CPU datom procesu. Straničenje, samim tim uvećava vreme zamene konteksta.

6.2 Hardverska podrška

Svaki operativni sistem ima svoje metode za čuvanje tabela stranica. Neki održavaju tabelu stranica za svaki proces, a *pokazivač* na tabelu stranica se čuva sa ostalim registrima (poput programskog brojača) u kontrolnom bloku procesa. Kada se dispečeru signalizira da treba da izvršava dati proces, on mora (ponovo) učitati korisničke registre i postaviti ispravne vrednosti u hardversku tabelu stranica na osnovu sačuvane tabele stranica iz kontrolnog bloka procesa. Drugi operativni sistemi imaju jednu ili eventualno par tabela stranica, što olakšava i uprošćava zamenu konteksta.

Hardverska implementacija tabele stranica može se obaviti na više načina. U najjednostavnijem slučaju, tabela stranica implementira se kao skup registara namenjenih u tu svrhu. Ovi registri bi trebalo da budu implementirani korišćenjem izuzetno brzih logičkih kola, kako bi omogućili efikasno transliranje adresa. Svaki pristup memoriji mora proći kroz tabelu stranica, tako da je efikasnost najvažnija. CPU dispečer, prilikom zamene konteksta, mora učitavati i ove registre, baš kao što učitava i ostale registre. Instrukcije za učitavanje ili modifikovanje registara tabela stranica su, naravno, privilegovane, tako da samo operativni sistem može da menja mapiranje memorije određeno ovim registrima. DEC PDP-11 je primer takve arhitekture. Kod nje, adresa se sastoji od 16 bita, a veličina stranice je 8 KB. Tabela stranica se sastoji od osam unosa koji se čuvaju u brzim registrima.

Upotreba registara za tabelu stranica je zadovoljavajuća pod uslovom da je tabela stranica relativno mala (na primer, 256 unosa). Međutim, većina savremenih računara omogućava da tabela stranica bude veoma velika (na primer, sa milion unosa). Za ove mašine upotreba brzih registra za implementaciju tabele stranica nije nimalo prihvatljiva. Umesto toga, tabela stranica se čuva u osnovnoj memoriji, a *bazni registar tabele stranica* (*Page-Table Base Register* - PTBR) pokazuje na tabelu stranica. Promena tabele stranica zahteva tada promenu samo ovog registra, značajno smanjujući vreme zamene konteksta.

Problem kod ovog pristupa odnosi se na vreme potrebno za pristup bilo kojoj lokaciji u okviru korisničke memorije. Ako želimo pristupiti lokaciji i,

prvo moramo čitati tabelu stranica, tako što na vrednost u PTBR registru dodamo broj stranice *i*. Da bismo ovo uradili, neophodno je da pristupimo memoriji. Čitanjem lokacije sa te adrese dobijamo broj okvira koji se kombinuje sa ofsetom u okviru stranice kako bismo generisali stvarnu fizičku adresu. Tek tada možemo pristupiti željenoj lokaciji u memoriji. Kao rezultat, za ovakvo mapiranje potrebna su dva pristupa memoriji da bi se pristupilo jednom bajtu (jedan za čitanje unosa u tabeli stranica, a jedan za bajt). Kao posledica, pristup memoriji je usporen sa faktorom dva, što je u većini slučajeva neprihvatljivo.

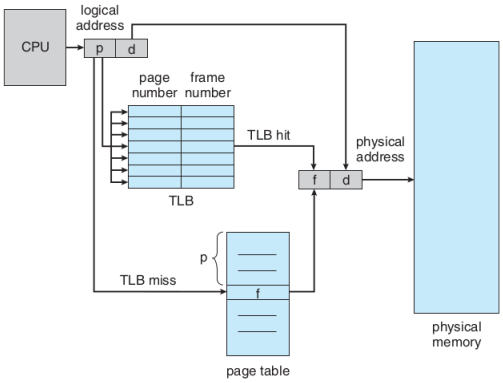
Standardno rešenje ovog problema je korišćenje specijalnog, malog, hardverskog keša koji se naziva *bafer za keširanje translacija* (eng. Translation Look-aside Buffer - TLB). Bafer za keširanje translacija (TLB) je asocijativna memorija velike brzine. Svaki unos u TLB-u sastoji se od dva dela: *ključa* (ili oznake-*tag*) i vrednosti. Kada se asocijativnoj memoriji prosledi neki upit, on se upoređuje sa svim ključevima istovremeno. Ako je odgovarajući ključ pronađen, vraća se odgovarajuća vrednosti. Ova pretraga je izuzetno brza - pretraga TLB-a u savremenom hardveru deo je protočne obrade instrukcija (*instruction pipeline*), što, u osnovi, nimalo ne pogoršava performanse. Međutim, da bi se mogla izvršiti pretraga TLB-a u sklopu instukcijskog ciklusa, TLB mora biti mali - obično veličine između 32 i 1.024 unosa. Neki procesori implementiraju zasebno TLB za instrukcije i TLB za podatke. To može udvostručiti broj dostupnih unosa u TLB, jer se pristupi instrukcijama i podacima dešavaju u različitim fazama protočne obrade instrukcija.

TLB se koristi sa tabelama stranica na sledeći način. TLB sadrži samo nekoliko unosa iz tabele stranica. Kada CPU generiše logičku adresu, broj stranice iz adrese se prosleđuje TLB-u. Ako se pronađe ključ sa datim brojem stranice, njen odgovarajući broj okvira memorije je odmah dostupan i koristi se za pristup memoriji. Kao što je već spomenuto, ovi koraci se izvode kao deo instrukcije u CPU-u, bez ikakvih pogoršanja performansi u poređenju sa sistemom koji ne koristi straničenje.

Ako, ipak, dati broj stranice nije u TLB-u (situacija poznata kao TLB promašaj - TLB *miss*), mora se najpre pristupiti memoriji kako bi se pročitala tabela stranice. Zavisno od CPU-a, ovo se može uraditi automatski u hardveru ili putem prekida u operativnom sistemu. Kako god da se izvrši, kada dobijemo broj okvira memorije, možemo ga koristiti za pristup memoriji (slika 14).

Dodatno, u TLB moramo dodati broj stranice i broj okvira, kako bismo ih brzo pronašli prilikom eventualnog narednog pristupa memoriji. Ako je TLB već popunjen, postojeći upis u TLB-u mora biti zamenjen i prepisan. Način na koji se ovo vrši (politika zamene ili *replacement policy*) može biti:

- odaberi onu lokaciju u TLB koja je najdavnije korišćenja (*Last Recently Used* -LRU)
- Round-Robin gde se u „kružnom“ maniru odabiraju TLB lokacije za zamenudo
- potpuno slučajan (Random) kod kojeg se na slučajan način odabira jedan od unosa u TLB.



Slika 14: Hardver za straničenje sa TLB

Neki procesori omogućavaju operativnom sistemu da učestvuje u zameni unosa korišćenjem LRU, dok drugi to obavljaju bez intervencije operativnog sistema. Dodatno, neki TLB-ovi omogućavaju da se određeni unosi fiksiraju, što znači da se oni ne mogu ukloniti iz TLB-a. Uobičajeno je da su u TLB-u fiksirani unosi korišćeni za kritični kod kernela.

Neki TLB-ovi dodatno smeštaju *identifikatore adresnog prostora* (Address-Space IDentifiers- ASID) kao deo svakog TLB upisa. ASID jedinstveno identifikuje svaki proces i koristi se za pružanje zaštite adresnog prostora za taj proces. Kada TLB pokuša razrešiti pristigle brojeve stranica, ASID za trenutno aktivni proces mora da se podudara sa ASID-om dodeljenim toj stranici. Ako se ASID-ovi ne podudaraju, pokušaj se tretira kao TLB promašaj. Pored toga što pruža zaštitu adresnog prostora, ASID dozvoljava TLB-u da istovremeno sadrži unose za nekoliko različitih procesa. Ako TLB ne podržava odvojene ASID-e, svaki put kada se postavi nova tabela stranica (na primer, sa svakom zamenom konteksta), TLB se mora izbrisati (*flush*) kako bi se osiguralo da naredni aktivni proces ne koristi pogrešne informacije o transliranju adresa. U suprotnom, TLB bi mogao da sadrži unose koji sadrže validne virtualne adrese (brojeve stranica), ali imaju pogrešne ili nevažeće fizičke adrese (brojeve okvira) koje su korišćene od strane prethodnog aktivnog procesa.

Procenat pristupa TLB-u u kojima se pronađe broj željene stranice zove se *procenat uspešnosti* (eng. *Hit ratio*). Na primer, procenat uspešnosti od 80% znači da u TLB-u pronalazimo željeni broj stranice 80% vremena. Ako je za pristup osnovnoj memoriji potrebno 100 nanosekundi, pristup mapiranoj memoriji traje 100 nanosekundi kada je broj stranice u TLB-u. Ako ne uspemo da nađemo broj stranice u TLB-u, prvo moramo pristupiti memoriji na lokaciji gde je smeštena tabela stranice i pronaći broj željenog okvira (100 nanosekundi), a zatim pristupiti željenoj lokaciji u memoriji (100 nanosekundi), što ukupno daje 200 nanosekundi. Pri tome, pretpostavljamo da pretraga tabele stranica zahteva samo jedan pristup memoriji, ali to ne mora nužno biti zadovoljeno, kao što ćemo videti uskoro. Efektivno vreme pristupa memoriji tada se računa kao:

effective access time = 0.80 x 100ns + 0.20 x 200ns = 120ns

U ovom primeru prosečni pristup memoriji će biti 20% sporiji, ali za procenat uspešnosti 99%, koji je znatno realniji:

effective access time = 0.99 x 100ns + 0.01 x 200ns = 101ns

što daje usporenje od samo 1%.

Današnji procesori obezbeđuju višestruke nivoe bafera za keširanje translacija. Stoga, računanje vreme pristupa memoriji je znatno komplikovanije od računice iznad. Na primer, Intel Core i7 CPU ima L1 instrukcijski TLB sa 128 unosa i L1 TLB za podatke sa 64 unosa. U slučaju TLB promašaja, treba mu 6 CPU ciklusa da proveri da li se data adresa nalazi u L2 TLB sa 512 upisa. Ovaj L2 TLB promašaj znači da CPU mora ili prošeta kroz sve upise u osnovnoj memoriji kako bi pronašao odgovarajuću adresu okvira, što može potrajati stotinak CPU ciklusa, ili se može koristiti prekid koji će omogućiti da operativni sistem odradi posao.

TLB je hardverska komponenta i, kao takav, nije od preteranog značaja za dizajnere operativnih sistema. Ipak oni treba da razumeju funkcionalnost i karakteristike bafera za keširanje translacija, koji se menjaju od sistema do sistema. U cilju optimalnog korišćenja, dizajn operativnog sistema za datu platformu mora implementirati straničenje u skladu sa dizajnom TLB-a na datoj platformi. Slično, izmena u implementaciji TLB-a (na primer naredna generacija Intel procesora) može zahtevati izmene u implementaciji straničenja kod operativnih sistema koji ih koriste.

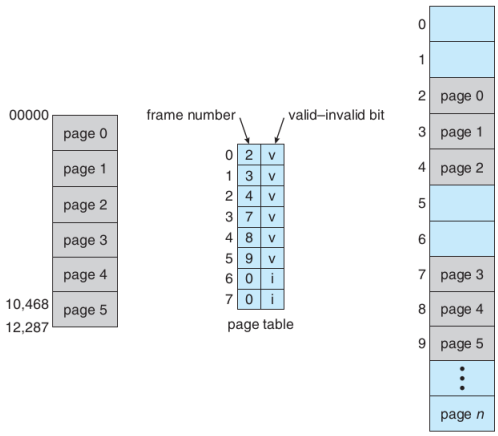
6.3 Zaštita

Zaštita memorije u sistemu koji koristi stranice je postignuta korišćenjem *bita protekcije* asociranih sa svakim okvirom. Normalno, ovi biti se čuvaju u tabeli stranica.

Jedan bit definiše da li se stranica može samo čitati (*read-only*) ili može u nju i da se upisuje (*read-write*). Svaka referenca ka memoriji prolazi kroz tabelu stranica, kako bi se pronašao odgovarajući broj okvira. Istovremeno, sa računanjem fizičke adrese, biti za protekciju se mogu proveravati kako bi se sprečio upis u stranice kod kojih je dozvoljeno samo čitanje. Pokušaj da se tako nešto uradi rezultovaće izuzetkom operativnog sistema (*memory-protection violation*).

Ovaj pristup se lako može proširiti kako bi se omogućila protekcija na višem nivou. Na ovaj način, moguće je označavati stranice kao stranice za čitanje, stranice za čitanje i upis, ili izvršne stranice. Takođe, zasebnim bitima za svaku od ove tri vrste protekcije, moguće su i njihove kombinacije. Svaki nedozvoljen pristup generisaće izuzetak.

Osim ovih, jedan poseban bit je dodat svakom upisu u tabelu stranica: *bit validnosti* (*valid-invalid bit*). Kada je ovaj bit postavljen na vrednost *validan*, stranica asocirana sa njim pripada logičkom adresnom prostoru procesa i stoga



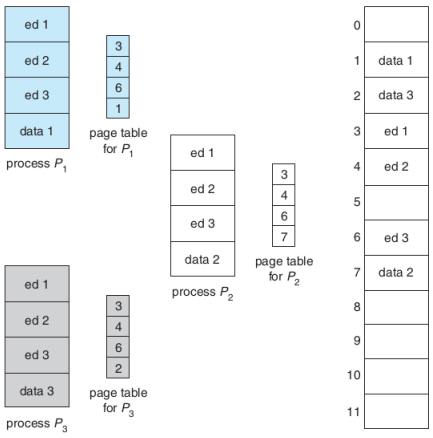
Slika 15: Korišćenje bita validnosti u tabeli stranica

je ona validna stranica. Kada je, nasuprot, bit postavljen na vrednost *nevalidan*, data stranica ne pripada logičkom prostoru procesa. Pristup ilegalnim adresama izazvaće izuzetak, na osnovu korišćenja bita validnosti, a operativni sistem postavlja ovaj bit za svaku stranicu, kako bi dozvolio ili zabranio pristup toj stranici.

Pretpostavimo, na primer, da sistem koristi 14-bitni adresni prostor (adrese 0 do 16383), a imamo program koji bi koristio samo adrese od 0 do 10468. Ukoliko je veličina stranice 2kB, imamo situaciju prikazanu na slici 15.

Adrese na stranicama 0, 1, 2, 3, 4 i 5 su mapirane normalno u tabeli stranica. Bilo koji pokušaj da se generiše adresa iz stranica 6 ili 7, sa druge strane, dovešće do toga da će odgovarajući bit validnosti biti prepoznat kao 0, i kao posledica desiće se izuzetak operativnog sistema (poznat kao *invalid page reference*). Primetite da ova shema ima problem, jer se program prostire do adrese 10468, te bi bilo koja referenca na memoriju van tog opsega trebala da bude ilegalna, takođe. Ipak reference na stranicu 5 su klasifikovane kao validne, tako da je i pristup bilo kojoj memorijskoj lokaciji do lokacije 12287 takođe validan. Jedino opseg adresa između 12288 i 16383 se smatra ilegalnim, a ovaj problem je posledica korišćenja stranica veličine 2kB i zapravo je posledica interne fragmentacije.

Obzirom na to da procesi retko koriste kompletan adresni prostor, odnosno, da uglavnom koriste samo mali deo dodeljenog im adresnog prostora, nepotrebno je kreirati tabelu stranica sa unosima za svaku stranicu u okviru adresnog prostora u tim slučajevima. Veći deo takve tabele bi bio neiskorišćen, ali bi svakako zauzimao (uvek) kritičan adresni prostor. Stoga, neki sistemi obezbeđuju poseban hardver u formi *registra za veličinu tabele stranica* (eng. *page-table length register*), kako bi omogućili indikaciju veličine tabele stranica. Vrednost iz ovog registra se poredi sa svakom logičkom adresom kako bi se verifikovalo da je ta adresa u legalnom opsegu adresa, a adresa koja nije automatski bi dovela do izuzetka u operativnom sistemu.



Slika 16: Deljenje koda u sistemu koji koristi stranice

6.4 Deljene stranice

Još jedna prednost straničenja odnosi se na mogućnost deljenja zajedničkog koda. Ovo razmatranje je posebno važno u okruženju sa deljenim vremenom (*time-sharing*, *multitasking*). Razmotrite sistem koji podržava 40 korisnika, od kojih svaki izvršava kod uređivača teksta (tekst editora). Ako se uređivač teksta sastoji od 150 KB koda i 50 KB prostora podataka, potrebno nam je 8000 KB da bismo istovremeno usluživali 40 korisnika. Međutim, ako je kod takozvani *čist kod* (eng. *reentrant code* ili *pure code*), on se, može deliti, kao što je prikazano na slici 16. Ovde vidimo tri procesa koji dele uređivač teksta koji zauzima tri stranice, od kojih je svaka veličine 50 KB (velika veličina stranice se koristi za pojednostavljenje slike i bolju preglednost). Svaki proces ima svoju stranicu koja skladišti podatke.

Čisti kod je kod koji ne može sam sebe da modifikuje, tj. kako god i u kojim god okolnostima da se pozove ovaj kod, on će uvek imati isti tok izvršavanja i dobijeni rezultati se neće menjati. Kao posledica, dva ili više procesa mogu istovremeno izvršavati isti kod.

Svaki proces ima svoju kopiju registara i svoj memorijski prostor u kojem skladišti podatke koji se koriste tokom izvršavanja procesa. Podaci za dva različita procesa će, naravno, biti različiti.

Sa druge strane, samo jedna kopija koda uređivača teksta mora se čuvati u fizičkoj memoriji. Tabela stranica svakog korisnika mapira njihove stranice na istu fizičku kopiju uređivača teksta, ali se stranice koje se odnose na podatke mapiraju na različite okvire memorije. Dakle, da bismo podržali 40 korisnika, sada nam je potreban samo jedan primerak uređivača teksta (150 KB), plus 40 kopija prostora podataka po korisniku (svaki 50 KB). Ukupan potreban prostor sada je 2.150 KB umesto 8.000 KB, što je značajna ušteda.

I drugi često korišćeni programi (kompajleri, deljene biblioteke, baze poda-

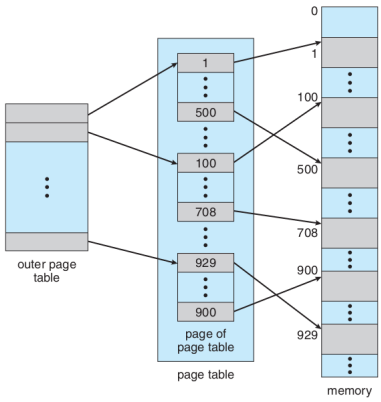
taka i tako dalje) takođe mogu biti deljeni. Jedini uslov da bi neki kod mogao da se deli, jeste da taj kod bude čist kod. Sa druge strane, operativni sistem mora da obezbedi da prostor u kojem se nalazi deljeni kod bude dostupan samo za čitanje. Nikako se ne sme osloniti na to da će sam kod biti takav da sam ne menja svoj adresni prostor.

Deljenje memorije između procesa u sistemu je slično deljenju adresnog prostora od strane niti u okviru procesa, kao što smo već opisali ranije. Takođe, pričali smo od deljenoj memoriji kao jednom od načina za međuprocesnu komunikaciju - neki operativni sistemi implementiraju deljenu memoriju koristeći upravo gore opisane deljene stranice.

7 Struktura tabele stranica

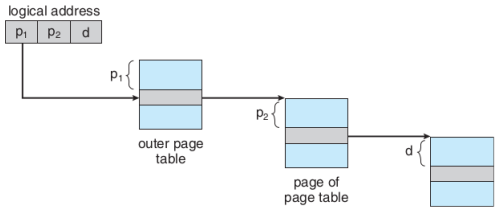
U ovom odeljku posmatraćemo neke od najčešće korišćenih tehnika za struktuiranje tabela stranica i time ćemo završiti sa pričom o straničenju.

7.1 Hijerarhijsko straničenje



Slika 17: Tabela stranica sa dva nivoa

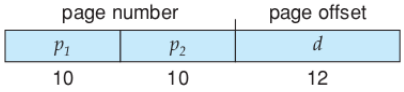
Većina modernih računarskih sistema podržava velike logičke adresne prostore (2^{32}do 2^{64}). U ovakvom okruženju, tabele stranica same po sebi mogu postati neprihvatljivo velike. Na primer, posmatrajmo sistem sa 32-bitnim logičkim adresnim prostorom. Ako je veličina stranice u takvom sistemu 4KB (2^{12}), tada tabela stranica može sadržati i do milion upisa ($2^{32}/2^{12}$). Pod pretpostavkom da je svaki upis u tabeli stranica 4 bajta, svaki proces bi trebao 4MB fizičkog adresnog prostora samo da skladišti tabelu stranica. Očigledno, ne želimo da skladištimo ovakvu tabelu stranica u bloku kontinualne memorije, a jednostavno rešenje da se tako nešto uradi jeste da se sama tabela stranica podeli u manje delove. Ovu podelu možemo da izvršimo na više načina.



Slika 18: Transliranje adresa kod tabele stranica sa dva nivoa

Jedan način jeste da se koristi *tabela stranica sa dva nivoa,* u kojem je tabela stranica takođe straničena (slika 17).

Na primer, razmatrajmo opet sistem sa 32-bitnim logičkim adresnim prostorom i stranicama od 4KB. Logička adresa je podeljena u broj stranice koji ima 20 bita i ofset koji je 12 bita. Pošto želimo da straničimo tabelu stranica, broj stranice je, dalje, podeljen u 10-bitni broj stranice i 10-bitni ofset. Kao rezultat, logička adresa izgleda ovako

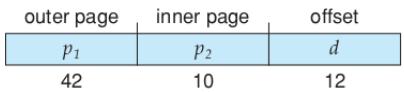


gde je $p_1$indeks u spoljnjoj tabeli stranica, a p_2 ofset koji se koristi u unutrašnjoj tabeli dobijenoj čitanjem spoljašnje tabele stranica. Ovakva translacija adresa prikazana je na slici 18.

Obzirom da se kod ovog pristupa translacija vrši od spoljnje ka unutrašnjoj tabeli, ovaj pristup je takođe poznat i kao *tabela stranica mapirana „unapred“* (*forward-mapped page table*).

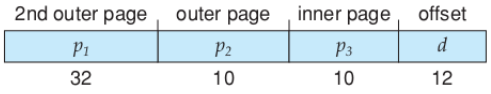
Koja je prednost ovakvog hijerarhijskog straničenja? Straničenje na ovaj način omogućava operativnom sistemu da neke delove tabela stranica ostavi neiskorišćenim sve dok ne postanu neophodne odnosno dok proces ne zatraži dodatnu memoriju. Brojne sekcije virtualnog adresnog prostora su najčešće neiskorišćene, a tabele stranica sa više nivoa u tom slučaju nemaju upisane lokacije koje odgovaraju tim neiskorišćenim prostorima, što značajno smanjuje veličinu memorije neophodne da se smeste tabele stranica.

U sistemu sa 64-bitnim logičkim adresnim prostorom, shema sa dva nivoa nije prikladna. Da bismo to ilustrovali, zamislimo da je veličina stranice i dalje 4KB (realna veličina stranice). U ovom slučaju, tabela stranica sadrži 2^{52} unosa. Ako bismo koristili shemu sa dva nivoa, tada bi bilo zgodno da unutrašnje stranice budu baš jednu stranicu velike, odnosno da sadrže 2^{10} 4-bajtnih upisa. Adresa bi tada izgledala



Spoljašnja tabela stranica bi tada sadržala 2^{42} upisa, odnosno bila bi velika 2^{44} bajta. Način da se izbegne ovako velika tabela jeste da se i spoljašnja tabela stranica dalje podeli u manje blokove, a ovaj pristup se nekad koristi i na 32-bitnim platformama kako bi se povećala fleksibilnost i efikasnost.

Spoljašnju tabelu možemo opet da podelimo na više načina. Na primer, možemo da straničimo spoljašnju tabelu tako da rezultujuća struktura bude straničenje sa tri nivoa. Međutim, ako bi spoljašnja tabela opet bila postavljena tako da odgovara veličini stranice, druga spoljašnja tabela bi i dalje bila neprihvatljivo velika (16GB).



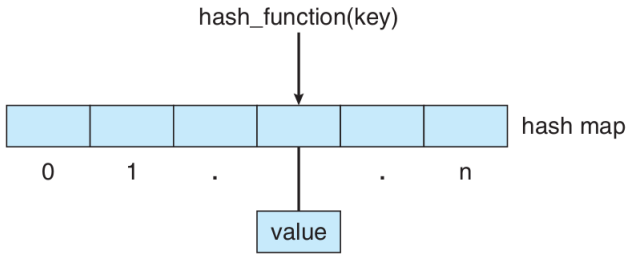
Očigledno je da se ovo nastavlja prelaskom na 4 nivoa i više. Stoga, na 64-bitnim arhitekturama, ovakav način hijerarhijskog straničenja se smatra neprihvatljivim.

7.2 Heširano (Hashed) straničenje

Heš funkcija (*hash function*) je funkcija koja za dati ulaz, izvodi na njemu numeričke operacije i vraća broj kao izlaz. Povratna vrednost funkcije može biti korišćena kao indeks u tabeli (tipično nizu), kako bi se brzo dobio željeni podatak. Dok pretraga za željenim podatkom u listi od n elemenata može trajati $O(n)$ vremena u najgorem slučaju, korišćenjem heš funkcije podaci se vraćaju u konstantnom vremenu O(1). Zbog ove činjenice, heš funkcije se često koriste u operativnim sistemima.

Jedna od osnovnih karakteristika heš funkcija jeste da one retko vraćaju iste vrednosti za različite ulazne parametre i zbog toga su uopšte i interesantne. Ipak, u određenim slučajevima, to se može desiti i ta situacija se naziva *heš kolizija* (*hash collision*). Jedan način da se reši problem heš kolizije jeste da se čuva povezana lista dodeljena svakom unosu u tabeli (koja se indeksira povratnim vrednostima heš funkcije), a u toj povezanoj listi se nalaze svi elementi koji rezultuju istom heš vrednošću (heš funkcija daje istu povratnu vrednost za te elemente korišćene kao parametre heš funkcije). Naravno, iako ovo jeste rešenje, što više heš kolizija postoji, to je sistem neefikasniji jer se gubi osnovna primena heš funkcija - eliminisanje pretrage.

Jedna implementacija heš funkcije jeste takozvana heš mapa (*hash map*) koja mapira ključeve na vrednosti korišćenjem heš funkcije kao na slici ispod.



Na primer, zamislimo da je dato korisničko ime u sistemu mapirano na lozinku tog korisnika. Autentikacija lozinke se tada svodi na to da korisnik unese svoje korisničko ime i lozinku. Unešeno korisničko ime se koristi kao ulaz u heš funkciju, a njen izlaz se koristi kao indeks u tabeli lozinki. Pročitana lozinka se, na kraju, poredi sa lozinkom unešenom od strane korisnika prilikom autentifikacije.

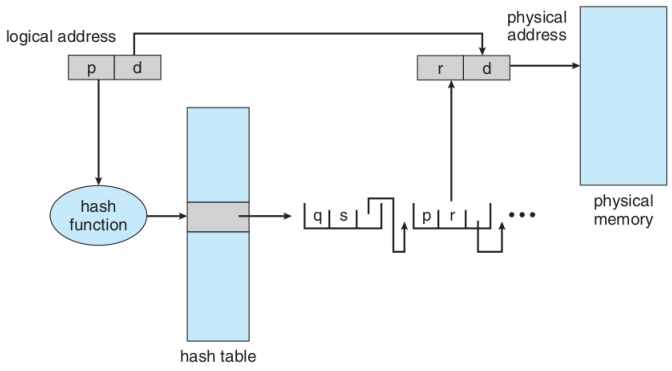
Tipičan pristup kod adresnog prostora većeg od 32 bita jeste da se koristi heširana tabela stranica, kod koje će heš vrednost biti broj virtualne stranice. Svaki unos u heš tabeli sadrži i povezanu listu elemenata za koje heš funkcija vraća istu vrednost odnosno sve virtualne brojeve stranica koje pokazuju na istu lokaciju u memoriji (na ovaj način se rešava heš kolizija u ovom slučaju). Svaki element u povazanoj listi se sastoji od tri polja: broj virtualne stranice, vrednost mapiranog okvira i pokazivač na naredni element u povezanoj listi.

Algoritam je tada, sledeći:

1. broj virtualne stranice iz virtualne adrese se koristi kao ulaz za heš funkciju,
2. izlaz heš funkcije koristi kao indeks u heš tabeli,
3. poredi broj virtualne stranice sa prvim poljem prvog elementa u povezanoj listi asociranoj sa dobijenim poljem iz heš tabele
4. ako se poklapa, odgovarajući broj okvira (polje 2 istog elementa) se koristi kako bi se generisala tražena fizička adresa
5. ako se ne poklapa, korak tri se ponavlja sa narednim elementima iz povezane liste

Algoritam je prikazan na slici 19.

Jedna varijacija ovog algoritma koja je korisna u slučaju 64-bitnih adresnih prostora koristi *klasterisane tabele stranica* (eng. *clustered page tables*). Klasterisane tabele stranica su slične heširanim sa izuzetkom što svaki element u povezanoj listi umesto jednog broja okvira sadrži više okvira (npr 16). *Faktor klasterisanja* se tada odnosi na broj okvira koji pripadaju jednom upisu u tabeli stranica. Klasterisane tabele stranica su posebno koristne kod „retkih“ adresnih prostora (*sparse adress spaces*), gde su reference na memoriju diskontinualne i razbacane po celom adresnom prostoru.



Slika 19: Heširana tabela stranica

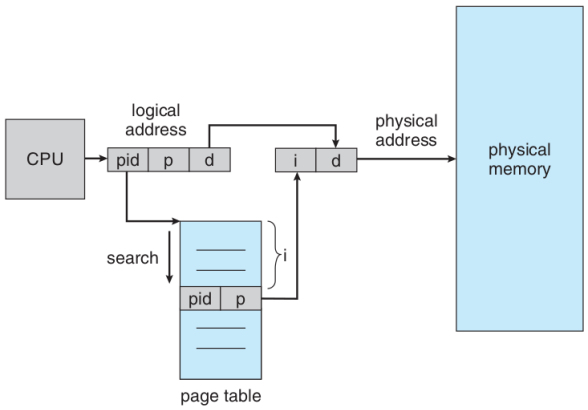
7.3 Invertovane tabele stranica

Kao što smo već rekli, tipično, svaki proces ima svoju tabelu stranica. Tabela stranica ima po jedan unos za svaku stranicu koju proces koristi, odnosno po jedan unos za svaku virtualnu adresu bez obzira na to da li će kasnije biti validna ili ne. Ovakva reprezentacija stranice je prirodna, obzirom na to da procesi referenciraju stranice kroz virtualne adrese stranica, a operativni sistem onda mora translirati tu adresu u fizičku memorijsku adresu. Pošto je tabela sortirana po virtualnim adresama, operativni sistem je u stanju da izračuna gde je u tabeli smeštena željena fizička adresa kako bi je direktno koristio. Jedna od mana ovog pristupa je, ipak, činjenica da svaka tabela stranica može da se sastoji od više miliona unosa, te stoga mogu zauzimati neprihvatljivo veliku količinu fizičke memorije, samo da bi se vodila evidencija na koji način se ostatak fizičke memorije koristi.

Da bi se rešio ovaj problem, pogotovo u sistemima sa manje fizičke memorije, moguće je koristiti *invertovane tabele stranica* (eng. *inverted page tables*). Invertovana tabela stranica ima jedan unos za svaki okvir (fizičke) memorije što znači da će na lokaciji i biti unos koji se odnosi na okvir sa brojem i. Svaki unos u tabeli stranica se sastoji od virtualne adrese stranice koja se mapira na odgovarajući okvir (slika 20), zajedno sa identifikatorom procesa koji poseduje tu stranicu. Kao rezultat, u celom sistemu imamo samo jednu tabelu stranica, koja ima samo jedan unos za svaki okvir fizičke memorije.

Kao što smo rekli, invertovane tabele stranica zahtevaju identifikator adresnog prostora (ASID, poglavlje 6.2) upisane u svakom od redova tabele, obzirom da tabela uglavnom sadrži više različitih adresnih prostora (adresnih prostora različitih procesa) koji se mapiraju na fizičku memoriju.

Dodavanje ASID-a svakom upisu u tabeli obezbeđuje da će se logička stranica datog procesa mapirati na odgovarajući fizički okvir. Ulogu ASID-a, pritom, često igra identifikator procesa, kao jedinstveni identifikator na nivou sistema. Na primer, kod IBM-a, virtualna adresa je formata:



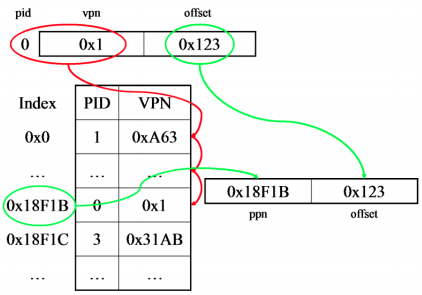
Slika 20: Invertovana tabela stranica

<process-id, page-number, offset>

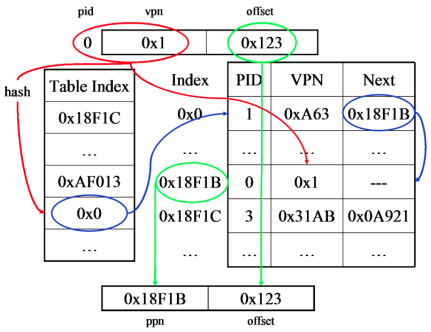
Kada se pristupa memoriji, deo virtualne adrese <*process-id*, *page-number*> se prosleđuje podsistemu za upravljanje memorijom. Invertovana tabela stranica se tada pretražuje, i ako se pronađe poklapanje, recimo na lokaciji *i*, fizička adresa <*i*, *offset*> je generisana. U suprotnom, ako nema poklapanja, došlo je do pokušaja pristupa nedozvoljenoj lokaciji.

Iako ovakav pristup smanjuje količinu memorije potrebnu da se skladište tabele stranica, on povećava vreme potrebno da se pretraži tabela u potrazi za traženom virtualnom adresom (i identifikatorom adresnog prostora). Pošto su invertovane tabele stranica sortirane po fizičkim adresama, ali pretraga se vrši po virtualnim adresama, u najgorem slučaju cela tabela mora biti pretražena da bi se našla odgovarajuća virtualna adresa (slika 21).

Da bi se ovaj problem rešio, koristi se *heširana invertovana tabela stranica* (eng. *hashed inverted page table*), na način sličan onome koji smo opisali u prethodnom poglavlju.



Slika 21: Pretraživanje invertovane tabele stranica



Slika 22: Heširana invertovana tabela stranica

Osim tabele stranica, koristi se dodatna heš tabela, ispred tabele stranica. Ulaz za heš tabelu su ASID i broj virtualne stranice, a pročitana vrednost iz heš tabele je unos u tabeli stranica. Međutim, da bi se razrešio problem sa eventualnom heš kolizijom, sama tabela stranica mora sadržati još jedno polje koje pokazuje na sledeći upis kojem odgovara isti heš. Na ovaj način, pretraga se svodi na jednu, maksimalno par provera upisa u tabeli stranica (slika 22[1]).

Naravno, svaki pristup heš tabeli dodaje jedan pristup memoriji, tako da jedno referenciranje virtualne memorije zahteva barem dva čitanja iz fizičke memorije: jedno za heš tabelu, a drugo za tabelu stranica. Naravno, i dalje se bafer za keširanje translacija (TLB) pretražuje prvi, što popravlja performanse sistema).

Međutim, sistemi koji koriste invertovane tabele stranica imaju problem sa implementacijom deljene memorije. Deljena memorija je obično implementirana tako da se više virtualnih adresa (po jedna za svaki proces koji koristi deljenu memoriju) mapira na jednu fizičku adresu. Ovaj standardni metod ne može biti korišćen kod invertovanih tabela stranica, obzirom na to da samo jedan unos sa virtualnom adresom postoji za svaki okvir fizičke memorije, te jedan okvir ne može imati dve (ili više) virtualnih adresa koje ga koriste.

[1] Napomena: u primeru sa slike 22, vrednost 0x0 iz heš tabele tabele mora biti pročitana sa lokacije koja odgovara identifikatoru procesa i broju virtualne stranice (na primeru sa slike, ako je pid=0 i vpn=0x1, indeks bi mogao biti 1 što se ne vidi sa slike).