1. Savremeni mikroprocesori

Denardovo skaliranje

- Robert Denard (Dennard): elektroinženjer, istraživačka karijera u IBM-u
- 1974.: snaga integrisanog kola po jedinici površine ostaje konstantna sa porastom gustine komponenti
- Uprošćeno: promena fizičkih dimenzija tranzistora menja njegove električne karakteristike tako da očuva gustinu snage, a dopušta rast radne frekvencije

Murov zakon

- Gordon Mur (Moore): inženjer, saosnivač Intela, osnivač Fairchild Semiconductors
- 1965.: broj komponenti integrisanog kola se udvostručuje svake godine; kasnije revidirano na 18 meseci ili dve godine (zbog ove promenljivosti i nije „zakon“ u fizičkom smislu)
- Murovo zapažanje je poznatije od Denardovog
- Rade zajedno—posledica je da performanse/W rastu brže od porasta broja komponenti

Uticaj na sistemsko programiranje— eksponencijalni rast

● Osnovno merilo performansi kod mikroprocesora opšte namene je broj celobrojnih operacija u jedinici vremena

Primeri spoljašnjih uticaja na arhitekturu

● Primer 1: operacije u pokretnom zarezu (razlomljene vrednosti), koje se masovno koriste u numeričkim modelima i naučnim izračunavanjima

● Rešenje: matematički koprocesori (dodatni procesori sa instrukcijama za rad u pokretnom zarezu), kasnije sastavni deo procesora

● Primer 2: rad sa većim količinama radne memorije (RAM=Random Access Memory), izvedene u dinamičkoj tehnologiji

Kategorije paralelizma

● Kad linearne pojedinačne performanse nisu dovoljne, može se razmatrati paralelizam

● Paralelizam podataka: jer postoje podskupovi podataka nad kojima se operacije mogu obavljati istovremeno

● Paralelizam zadataka: jer se celokupni zadaci često mogu podeliti na relativno nezavisne podzadatke

● Ove kategorije se mogu iskoristiti organizaciono ili tehnički—nas zanima tehničko iskorišćenje u okviru mikroprocesora

Iskorišćenje paralelizma na nivou mikroprocesora

● Paralelizam na instrukcijskom nivou: korišćenje paralelizma podataka za istovremeno izvršavanje skupova instrukcija

● Vektorske arhitekture, grafički koprocesori, multimedijalne instrukcije: korišćenje paralelizma podataka tako što se jedna instrukcija primenjuje na veću količinu podataka

● Paralelizam na nivou lanaca izvršavanja (thread-level): paralelizam podataka ili zadataka u tesno spregnutoj interakciji pojedinačnih lanaca izvršavanja

Flinova taksonomija

● Metod klasifikacije procesorskih sistema i/ili skupova instrukcija po stepenu paralelizma (Flynn, 1966)

● SISD (Single Instruction, Single Data), klasične celobrojne ili pokretno-zarezne instrukcijama

● SIMD (Single Instruction, Multiple Data), vektorske ili multimedijalne instrukcije

● MISD, ne postoji u praksi

● MIMD, kombinacija SIMD za više izvršnih jedinica, podrazumeva paralelizam zadataka

2. Savremeni mikroprocesori

Model rada (klasična fon Nojmanova arhitektura)

● Memorija je linearna i sekvencijalno adresirana (veličina najmanje adresibilne jedinice nije bitna, danas je to 8-bitni bajt/oktet)
● Instrukcije se izvršavaju strogo sekvencijalno
– izvršavanje naredne instrukcije počinje tek kad je prethodna završena
– naredna instrukcija vidi sve efekte prethodne
● Integrisana kola su sinhrona
– ovo nema neposredne veze sa načinom izvršavanja instrukcija, ali ima presudan uticaj na arhitekturu

Instrukcijski ciklus

● Koraci potrebni za izvršavanje jedne instrukcije, apstraktno gledano; realni procesori imaju mašinski ciklus
● (1) Prihvatanje instrukcije
– iz memorijske lokacije na koju pokazuje **programski brojač,** registar koji prati izvršavanje instrukcija
– po završetku ovog koraka programski brojač će pokazivati na sledeću instrukciju u nizu
● (2) Dekodiranje instrukcije
● (3) Čitanje efektivne adrese (može se posmatrati i kao potkorak drugog koraka)
– ako instrukcija koristi indirektno adresiranje, efektivna adresa se čita iz memorije
– ako je instrukcija memorjska i direktna, u ovom koraku se ništa ne dešava
– isto važi i za instrukcije koje ne operišu s memorijom
● (4) Izvršavanje
– ukoliko je u pitanju izvršena instrukcija grananja, postaviće novu vrednost u programski brojač

Petostepeni transport

● „5-stage pipeline“, jedan od načina da se instrukcijski ciklus mapira na hardver
● Istorijski, prvobitno upotrebljen (za mikroprocesore) u RISC arhitekturama 1980-ih

IF	Instruction fetch	Prihvatanje instrukcije
ID	Instruction decode	Dekodiranje instruckije
EX	Execute	Izvršavanje
MEM	Memory access	Pristup memoriji
WB	Register writeback	Zapis registara

Primarni zahtevi „pipelined“ ahitekture

● Instrukcije su uniformne dužine, najčešće 32 bita
● Koriste se proste instrukcije za memorijski pristup
– „Load/store architecture“, bez komplikovanih režima adresiranja
● Sve ostale instrukcije rade s registrima
● Široko korišćeno u RISC arhitekturama, kanonički primer: MIPS, moderna iteracija: RISC-V

Problemi u „pipelined“ arhitekturi

● Dve osnovne kategorije problema
● Prva, veće zauzeće memorije za instrukcije nego što je neophodno
– ovo znači i indirektan pritisak na interne procesorske memorijske strukture, što ćemo obraditi kasnije
● Druga, narušavanje sekvencijalne semantike izvršavanja:
– prilikom rada s podacima
– prilikom grananja

Zauzeće memorije za instrukcije

● Uobičajen izbor je bio veličina mašinske reči, 32 bita
● Ovo je dovoljno da se nedvosmisleno kodiraju troadresne instrukcije sa skupom od 32 registra
● Značajan podskup programa ne zahteva ovako veliki registarski prostor
● Uvode se kompaktne, 16-bitne instrukcije koje rade na smanjenom skupu registara i ograničenom skupu instrukcija
● Kanonički primer: ARM Thumb

Varijante kompaktne reprezentacije

● Prva varijanta (Thumb u prvobitnoj realizaciji) zahteva da se procesor prebaci u poseban režim rada da bi izvršavao ovakve instrukcije
● Ovo je problematično jer se često dešava da je u nizu kompaktnih instrukcija potrebno izvršiti nekoliko instrukcija standardne veličine, pa treba preskakati između režima rada
● Modernija izvedba (RISC-V, ARM Thumb2) omogućava da se standardne i kompaktne instrukcije slobodno mešaju
● Rezultat (RISC-V): gustina koda vrlo slična onoj koju ima Intel x86_64 arhitektura

Semantika pristupa podacima

● Uopšteno: problem nastaje ako paralelno izvršavanje delova instrukcija ili čitavih instrukcija izazove efekat nemoguć u sekvencijalnom kodu
● Uvek je moguće sprečiti semantičke probleme ako se u izvršavanje unese čekanje, čija je krajnja granica ekvivalent sekvencijalnog izvršavanja
● Ovo obara performanse, pa se, ako je moguće, traže rešenja koja će raditi bez čekanja

Klasifikacija opasnosti

● Nepoželjni efekti se zovu opasnosti (hazards) i označavaju se skraćenicama
● Oznake mogu da deluju zbunjujuće, jer ukazuju na poželjan efekat
● Svaka od opasnosti uključuje zapisivanje vrednosti u nekoj od kombinacija; samo čitanje nikad ne može narušiti semantiku
● Kad budemo opisivali opasnosti, navešćemo prvo poželjan, pa nepoželjan efekat, koji treba sprečiti
● RAW (Read After Write):
– želimo da čitanje posle pisanja vrati upravo zapisanu vrednost
– opasnost: naredna instrukcija ne vidi rezultat izvršavanje prethodne
● WAR (Write After Read):
– želimo da čitanje pre pisanja vrati originalnu vrednost – opasnost: prethodna instrukcija vidi rezultat izvršavanja naredne
– nemoguće u klasičnoj petostepenoj arhitekturi, može da se desi ako se instrukcije izvršavaju preko reda
● WAW (Write After Write):
– želimo da konačan rezultat ove sekvence bude druga zapisana vrednost
– opasnost: ukupan efekat je rezultat izvršavanja prethodne instrukcije, a ne naredne
– takođe nemoguće u standardnoj petostepenoj arhitekturi
● Ilustrovaćemo samo RAW opasnost jer je najčešća, relativno lako shvatljiva, i često ima tehničko rešenje

RAW problem (rešiv bez čekanja)

ADDU R2,R1,R3	IF	ID	EX	MEM	WB	
SUBU R4,R2,R5		IF	ID	EX	MEM	WB

● Instrukcija ADDU izračunava rezultat u stepenu EX, ali ga zapisuje u R2 tek u stepenu WB
● Instrukcija SUBU čita registar R2 u stepenu ID, tako da vidi raniju vrednost
● Rešenje: hardverska indikacija promene tako da rezultat postaje vidljiv već u stepenu ID

RAW problem (nerešiv bez čekanja)

LW R1,0(R2)	IF	ID	EX	MEM	WB

SUBU R4,R1,R5 IF ID EX MEM WB

- Instrukcija LW (Load Word) ima vrednost na raspolaganju tek u stepenu MEM, kada se instrukcija SUBU već izvršava

● Mora se čekati bar jedan mašinski takt da bi učitana vrednost mogla da se pročita u stepenu ID od strane instrukcije SUBU

Grananje

● Dva osnovna problema s grananjem
● U principu se ne može pouzdano znati koja će se instrukcija semantički izvršiti posle instrukcije grananja
● Kod petostepene arhitekture, u trenutku dekodiranja instrukcije grananja naredna instrukcija (u memoriji) već je pročitana, tako da je trošak njenog čitanja i dekodiranja straćen ukoliko se grananje izvrši
● Za prvi problem, bez dodatnih komplikacija nema rešenja sem čekanja
● Za drugi, neki RISC procesori čine vidljivom (i izvršavaju) instrukciju iza grananja bez obzira da li je do grananja došlo: „branch delay slot"
● Ovo je zbunjujuće za programere i nezgodno za kompajlere, tako da su kasniji procesori ovo izbegavali

Dodatno unapređenje paralelizma

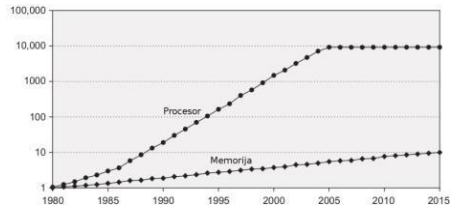
● Prekoredno izvršavanje (Out of order execution) – izvršava instrukcije u zavisnosti od dostupnih podataka
● Superskalarni procesori – izvršavaju više od jedne instrukcije istovremeno
  – koriste postojanje više izvršnih jedinica u okviru procesora
  – (ovo još uvek nije višeprocesorski sistem)
● Spekulativno izvršavanje (Speculative execution) – izvršavanje unapred da bi se amortizovali troškovi grananja
● Predviđanje grananja (Branch prediction), sklop se zove prediktor
– prosti statički (uslovno grananje se neće izvršiti)
– statički (→ se neće izvršiti, ← će se izvršiti)
– primena spekulativnog izvršavanja

3. Memorijska hijerarhija i radna memorija

Memorijska hijerarhija

● Što je izvor podataka bliži procesoru, ima veće performanse, višu cenu po jedinici kapaciteta i manji kapacitet
● Hijerarhija postoji od početka razvoja računara, a nivoi su bili srazmerno stabilni tokom decenija
● Standardni nivoi:
  – Procesorski registri
  – Keš memorija
  – Radna memorija
  – [PCM, Flash] masovna memorija
  – Magnetni diskovi

Osnovna disproporcija u performansama

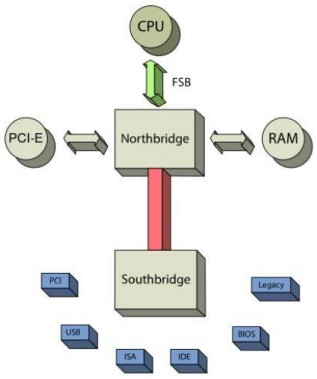


Tipičan hardver računara opšte namene

● Jedan do dva fizička procesora (socket) – retko se ide na više fizičkih procesora zbog komplikovanja konstrukcije
● Prostor za 16 GB – 256 GB dinamičke memorije
– velike razlike između serverskih i desktop varijanti
– za veće servere su dostupni i memorijski kapaciteti reda TB

- Integrisani periferni uređaji
- Relativno slična osnovna arhitektura

Šema hardverske organizacije (do ~2011.)



Osnovni elementi organizacije

● Northbridge: direktno komunicira s procesorom
– FSB = Front Side Bus
– zadužen za upravljanje memorijom
– dodatno, za brze periferne uređaje
● Southbridge: ostali periferni uređaji
– sve sporije magistrale
– memorija gde brzina pristupa nije kritična (BIOS)

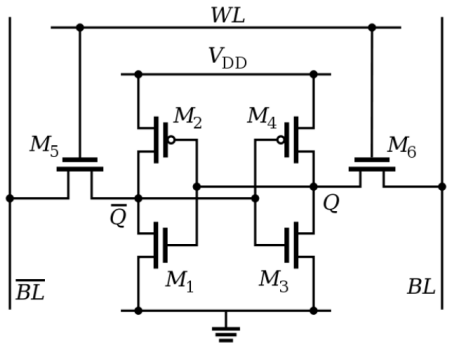
Posledice organizacije

● Komunikacija između fizičkih procesora koristi FSB
● Sva komunikacija s memorijom ide preko NB – memorija ima jednu pristupnu tačku (port)
– višestruki pristup kod specijalizovanih uređaja
● Komunikacija sa perifernim uređajima zakačenim na SB ide preko NB
● Brzi periferni uređaji direktno pristupaju memoriji – DMA (Direct Memory Access)
– Ovime se rasterećuje procesor...
● ... Ali stvara konkurenciju s procesorom za pristup memoriji preko NB
● Veza NB s memorijom i njeno iskorišćenje su kritični za performanse

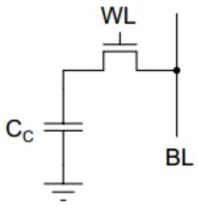
Kako povećati propusnu moć

● NB ne mora sam da upravlja memorijom, već za to mogu postojati posebna kola
● Više memorijskih kontrolera na NB – veća količina podržane memorije
– bolja propusna moć
● Ograničenje postaje interna propusna moć NB
● Integrisani memorijski kontroler na fizičkom procesoru – današnja varijanta
● U ovoj organizaciji ne postoji poseban NB, SB se zove „Platform Controller Hub“ (kod Intela)
● Više procesora znači i više kontrolera, sa prednostima kao kod prethodne varijante
● Problem: kod više fizičkih procesora, pristup memoriji više nije uniforman
– ekstremna varijanta: NUMA (Non-Uniform Memory Access)

Statički RAM (6 tranzistora)



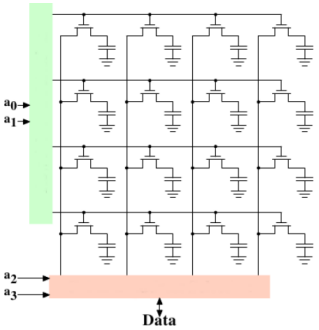
Dinamički RAM (1 tranzist



Opšte napomene

- Dinamički RAM je očigledno jednostavniji od statičkog
- Problemi: – čitanje sadržaja je posle izvesnog broja ciklusa destruktivno
  – kapacitet kondenzatora ne može biti veliki
  – mora se periodično osvežavati
- U toku osvežavanja pristup je nemoguć
- Direktno čitanje je nemoguće
- Moraju postojati pauze tokom čitanja/pisanja

Organizacija dinamičkog RAM-a

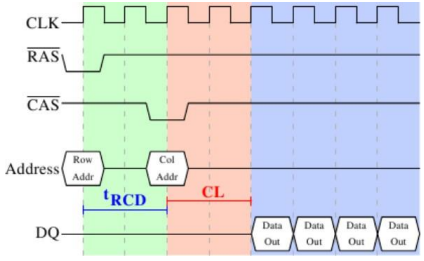


- Ćelije su organizovane u matricu, koja treba da ima jednak broj vrsta i kolona ako je moguće

– raskorak komplikuje hardver za (de)multipleksiranje na većoj strani

- Vrsta se bira signalom RAS (Row Address Selection) – linija iznad znači da je signal invertovan
- Kolona se bira signalom CAS (Column Address Selection)
- RAS se demultipleksira na osnovu dela adrese (tj. signal se usmerava na jednu od vrsta u zavisnosti od kombinacije **a0/a1**)
- Aktiviranje vrste će kao rezultat imati iščitavanje svih ćelija u okviru te vrste, čiji se signali sprovode do multipleksera kolona
- CAS aktivira multiplekser na osnovu ostatka adrese, kombinacije **a2/a3**, i na izlazu se pojavljuje očitavanje adresirane ćelije
- Sama adresa je često multipleksirana

Dijagram pristupa RAM-u



Posledice načina pristupa

● Pristup je mnogo sporiji u odnosu na kapacitet procesora – vremenski, 10–15:1
● Pristup proizvoljnoj lokaciji zahteva čekanje
● Prenos susednih lokacija je efikasan
● Moguće je raditi čitanje unapred da bi se smanjilo čekanje

Drugi korisnici memorije

● Grafički podsistem – integrisana grafika bez odvojene memorije
– ako se zahtevaju bolje performanse ove komponente su izdvojene
● Disk kontroleri
● Mrežni kontroleri – uporedivi sa (magnetnim) diskovima po sirovoj brzini

4. Keširanje

Rekapitulacija za memoriju/procesor

● Inverzna korelacija složenosti memorije i njenih performansi
● Praktičan sistem mora imati veliku količinu memorije da bi mogao da obrađuje realne skupove podataka
– dakle, radna memorija mora biti sporija u odnosu na procesor
● Skupovi podataka su često veći i od praktično upotrebljive količine radne memorije
– sekundarna memorija je nekoliko redova veličine sporija od glavne
– ovo se donekle menja postojanjem bržih sekundarnih memorija (flash)

Moguće arhitekture međumemorije

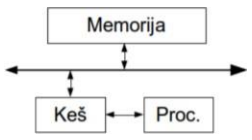
● Uvodi se mala količina brze međumemorije
● Jedna od mogućnosti je da se memorija tretira kao prošireni skup registara pod kontrolom korisničkih procesa
– procesi moraju znati hardverske detalje pojedinačnih procesora
– svaki proces mora imati disjunktni podskup u odnosu na druge
– trošak organizovanja pristupa bi poništio prednosti
● Druga mogućnost, koja se univerzalno koristi, jeste međumemorija (uglavnom) transparentna za korisnika

Keš memorija: principi

● Brza memorija koja sadrži privremene kopije podataka za koje se može pretpostaviti da će biti potrebni za izvršavanje koda
● Ovo je izvodljivo jer programi ispoljavaju lokalitet
● Prostorni lokalitet: mali kontinualni podskup podataka se koristi u kontinuitetu
– na primeru koda: petlja
– izvesne konfiguracije podataka su takođe prostorno bliske
● Vremenski lokalitet: verovatnoća da će neki podskup podataka biti opet potreban ne dugo posle obrade
– za kod: funkcijski poziv u petlji (kod funkcije je prostorno udaljen, ali potreban kod svake iteracije)
– za podatke: ukupan radni skup relativno ograničen, ali ne mora biti prostorno kontinualan

Konfiguracija keša

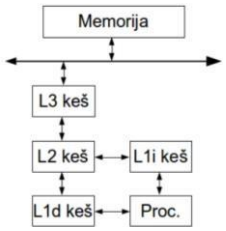
● Prvobitni sistemi:



● Sav pristup memoriji ide kroz keš

● Veza između keša i procesora je specijalno projektovana za brzinu
● Ovo funkcioniše, ali moguća su poboljšanja:
– razdvajanje memorije za instrukcije i podatke: modifikacija klasične fon Nojmanove arhitekture
– organizacija u nivoima, zbog neekonomičnosti povećanja jedinstvenog memorijskog prostora

Moderna konfiguracija keša



Osnovni parametri keša

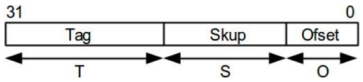
Vrsta pristupa	Br. ciklusa	Vreme ns
L1 pogodak	4	1,2
L2 pogodak	10	3,0
L3 pogodak	40	12,0
lokalni RAM	100	60
udaljeni RAM	160	100

● Pretpostavka: 100 elemenata, 100 iteracija
● Svaki pristup kroz lokalni RAM: 100·100·100 = 1.000.000 ciklusa
● Svaki pristup kroz L2 keš: 100·100·10 = 100.000 ciklusa
● Ušteda: 90%
● U realnim situacijama iskorišćenja iznad 90% su uobičajena

Osnovna organizacija keša

● Tipično, keš je hiljadama puta manji od količine RAM-a
● Potrebno je naći efikasan način za popunjavanje i adresiranje sadržaja
● Svaki zapis u kešu mora biti dostupan na osnovu adrese podatka koji se traži
– deo zapisa koji predstavlja adresu zove se tag
● Granularnost zapisa na nivou mašinske reči bila bi krajnje neefikasna
– zato se postavlja na 32 ili 64 bajta, što predstavlja jednu liniju
● Prilikom izmene memorije, cela linija prvo mora da se učita u keš
– u principu, procesor nema operacije koje rade nad celom linijom
● Izmena neke lokacije u liniji označava celu liniju kao zaprljanu
– zaprljane linije su one koje su izmenjene, a nisu zapisane nazad u memoriju
● Prilikom čitanja, najčešće se neka od postojećih linija mora izbaciti iz keša
– ako su svi kandidadi zaprljani, moraju se zapisati, što unosi kašnjenje
– izbor kandidata je generalno težak problem
● Za primer: 4 MB, 64 bajta, 65536 linija (22 bita)

Adresiranje linija



● Segment O je fiksan i zavisi od dužine linije: O = log2L
● Segmenti S i T zavise od načina organizacije adresiranja
– potpuno asocijativno
– direktno mapirano
– skupovno asocijativno

Potpuno asocijativni keš

● S = 0, T = 32 - O

- Svaka linija može sadržati kopiju bilo koje memorijske lokacije – obratite pažnju: i dalje postoji mogućnost promašaja
- Za svaku od linija mora postojati komparator koji će proveriti jednakost tagova

● Hardverski preterano zahtevno
● Komparatori ne mogu da rade iterativno jer se inače gubi smisao keširanja

Direktno mapirani keš

● Hardverski problem se rešava ograničavanjem opsega pretrage
● Na jednoj krajnosti, svaki tag se mapira na tačno jednu liniju
● S = log2N, gde je N broj linija – u našem slučaju N = 65536, S = 16
● Može dobro da radi ako su adrese ravnomerno raspoređene u prostoru određenom za mapiranje
– najčešće nisu
– praktična posledica: neke linije se stalno zamenjuju, neke ostaju prazne

Skupovno asocijativni keš

● Kombinacija prethodna dva
● S više ne bira pojedinačne linije, već skupove linija
– slično direktno mapiranom, ali je promenjena granularnost adresiranja
● Unutar svakog skupa, linije koje mu pripadaju porede se s tagom u paraleli
– slično potpuno asocijativnom, samo za mali broj elemenata
● Na našem primeru, ako je keš 8-struko asocijativan, imaće 8192 skupa
– 8192·8 = 65536
– S = log28192 = 13

Primer—direktno mapiran keš

● Polazna memorijska adresa: 0x9cb2008
● Binarno, sa grupisanim ciframa: 1001 1100 1011 0010 0000 0000 1000
● Ista ta vrednost, sa razdvojenim tagom i selektorom linije unutar keša: 1001 1100 1011 0010 0000 0000 1000
tag: 0x27, selektor: 0x2c80, ofset: 0x8
● Adresa: 0xa0b2008, isti postupak: 1010 0000 1011 0010 0000 0000 1000
tag: 0x28, selektor: 0x2c80, ofset: 0x8
● Ako program naizmenično pristupa ovim adresama, linija će svaki put morati da se obnavlja iz RAM-a

Primer—8-struko asocijativni keš

● Polazna memorijska adresa: 0x9cb2008, isti postupak kao i ranije, ali ovog puta će selektor biti 13 bita umesto 16:
1001 1100 1011 0010 0000 0000 1000
tag: 0x139, selektor: 0xc80, ofset: 0x8
● Polazna adresa 0xa0b2008: 1010 0000 1011 0010 0000 0000 1000
tag: 0x141, selektor: 0xc80, ofset: 0x8
● Obe adrese se mapiraju na isti skup, ali imaju različite tagove, pa pošto skup ima kapacitet od osam linija, veće su šanse da za obe linije ima mesta u kešu

Keširanje instrukcija

● Jednostavnije od keširanja podataka
● Programski prevodioci generišu kod koji bolje vodi računa o iskorišćenju keša
– autori prevodilaca generalno bolje poznaju ponašanje procesora
● Tok izvršavanja programa je predvidljiviji od šema pristupa podacima
● Programi po pravilu imaju dobar prostorni i vremenski lokalitet
● Eventualni problem: samomodifikujući kod

5. Virtuelna memorija

Jednoprocesni sistem

- Ograda: sistem koji sinhrono reaguje na spoljašnje događaje nikad ne može da bude striktno jednoprocesni
- Proces vidi fizičku memoriju, kako je hardver mapira u adresni prostor procesora – ± memorijski mapirani uređaji
- Sve memorijske adrese su fizičke adrese

● Ovo stvara poteškoće: – ako je potrebno pokretati pozadinske procese
– ako memorijski raspored nije fiksan

Primer: MS-DOS

Problemi kod višeprocesnog rada

● Nestabilnost adresnog prostora
– apsolutna odredišta grananja ne mogu da se koriste u fiksnom obliku
– reference na statički alocirane podatke takođe
– rešava se dinamičkom relokacijom
● Problem stabilnosti sistema
– proces može da pristupi osetljivim delovima memorije (npr. mapiranja za uređaje, prekidni vektori)
– ovo može da izazove nestabilnost ili krah
● Problem bezbednosti/privatnosti
– svaki proces može da vidi podatke svih drugih
– podnošljivo kod jednokorisničkog sistema, nedopustivo kod višekorisničkog

Upravljanje memorijom

● Za rešavanje navedenih problema uvodi se koncept upravljanja memorijom
● Poseban podsistem na procesoru koji se zove jedinica za upravljanje memorijom (Memory Management Unit, MMU)
● Jedan od zadataka je da se memorija organizuje tako da svaki proces ima:
– sliku raspoložive memorije kao da je jedini koji se izvršava na sistemu
– memorijski prostor zaštićen od drugih procesa

Virtuelna memorija

● Uvođenje MMU omogućava virtuelizaciju memorije
● Zaštita procesa je samo jedan (ali veliki) zadatak virtuelizacije
– drugi je omogućavanje rada sa adresnim prostorom većim od raspoložive fizičke memorije
– korišćenje sekundarne memorije (disk) kao podrške za memorijski sadržaj
– mehanizam sličan keširanju
● Memorijske reference koje program koristi više se ne odnose direktno na fizičku memoriju
– sve adrese su virtuelne adrese
– mora postojati način za prevođenje iz virtuelnih u fizičke adrese

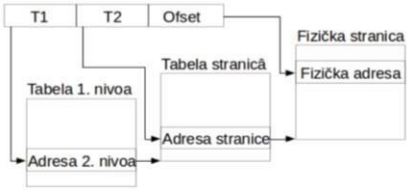
Principi prevođenja

● Memorija se deli na stranice (pages, 4 KB – 4 MB)
● Mora postojati memorijska struktura koja pokazuje na pojedinačne stranice
● Ova struktura se zove tabela stranica (Page Directory)
● Mogući su različiti pristupi prevođenju, u zavisnosti od veličine i broja stranica koje treba pokriti

Prosto prevođenje

● Veoma nalik na keširanje – pokazivač na tabelu stranica = tag
– ofset u okviru stranice = ofset u okviru linije
● Realan primer: stranice od 4 MB – 22 bita za ofset, 10 bita za indeks stranice
– 1024 elementa u tabeli
● U opštem slučaju, velike stranice su neefikasne
– veliki broj operacija zahteva poravnanje na veličinu stranice, što povećava utrošak memorije
● Za manje stranice, jedan nivo je takođe neefikasan
– tabela stranica je velika
– svakom procesu je potrebna posebna tabela
– posledica: velika potrošnja memorije na tabele za upravljanje memorijom(!)

Više nivoa prevođenja



● Podelom adrese stranice na više delova omogućava se izbegavanja zauzimanja memorije za nealocirane delove u memorijskoj mapi
● Kod realne organizacije procesa čest je slučaj da su zauzeti disjunktni blokovi memorije na suprotnim krajevima adresnog prostora
● Ova organizacija omogućava i korišćenje većeg fizičkog adresnog prostora u odnosu na adresni kapacitet procesora
● Tabele su hijerarhijske
● Proces konstruisanja prevođenja je nužno iterativan, ne može se paralelizovati
● Ako je MMU aktivan, prevođenje je potrebno prilikom svakog memorijskog pristupa
– za dva nivoa, dva pristupa memoriji
– tabele mogu imati do četiri nivoa
– ako su tabele u L1d, bar četiri dodatna ciklusa po pristupu
● Potreban je način da se ovaj postupak ubrza

TLB keš

● Rešenje koje se primenjuje jeste da se kešira ukupan rezultat prevođenja: fizička adresa stranice
● Keš u kome se drže ovi podaci zove se TLB (=Translation Lookaside Buffer)
● Ovaj keš mora biti veoma brz da bi se izbeglo čekanje prilikom svakog memorijskog pristupa
● Obično realizovan kao potpuno asocijativni, sa malim brojem elemenata

6. Sistemske interakcije virtuelne memorije

Četiri pitanja memorijske hijerarhije

● Gde se memorijski blok može smestiti na višem nivou?
● Kako pronaći blok na višem nivou?
● Koji blok zameniti prilikom promašaja?
● Šta se dešava prilikom pisanja vrednosti (ili, kakva je politika pisanja)?
● Ova pitanja se podjednako mogu postaviti i za keš i za virtuelnu memoriju; prvo ćemo diskutovati keš

Keš: pitanja hijerarhije ● Kod keša, blok = linija (64 bajta)
● Prvo pitanje (gde se linija može smestiti) je već diskutovano: asocijativnost
● Drugo pitanje (kako naći liniju) takođe je postavljeno, u vezi s prvim, ali ćemo ga dodatno obraditi zbog interakcije s virtuelnom memorijom
● Za preostala dva pitanja postoje uočljive paralele između keširanja i virtuelne memorije

Keš: koju liniju zameniti?
● Ako ima slobodnih, nije problem
● U suprotnom, nekoliko strategija:
– (pseudo)slučajni izbor sa ravnomernom raspodelom
– LRU (Least Recently Used) najmanje skoro korišćena; oslanja se na logičku posledicu lokaliteta —ako linija nije skoro korišćena, verovatno neće biti skoro potrebna
– FIFO (First In, First Out) ali u obrnutom redosledu: uzima se najstarija linija kao aproksimacija najmanje skoro korišćene. FIFO se lakše implementira nego LRU

Keš: šta se dešava prilikom pisanja?
● Kod realnih programa čitanje memorije je češće od pisanja, otprilike 5:2 u nekim merenjima
● Dodatno, svako prihvatanje instrukcije je čitanje
● Čitav memorijski podsistem se može praktično optimizovati za čitanje
● Pisanje se mora obavljati pažljivije od čitanja:
– vrednosti se ne smeju zapisivati dok niste sigurni da je prâva linija u kešu
– mora se strogo voditi računa o veličini promena

Keš: politika pisanja
● Prolazno (write through): sve vrednosti se odmah zapisuju i u keš i na nižem nivou
● Zadržano (write back): vrednosti se isprva zapisuju samo u keš, mogu se zapisati niže prilikom zamene linije ili periodično
● Svaka od politika ima svoje prednosti i mane
● Po potrebi, mogu se kombinovati

Keš: karakteristike politika pisanja
● Prolazno pisanje se jednostavnije implementira
● Takođe, sadržaj keša je uvek čist i ažuran
● Ovo naročito ima prednosti kod višeprocesorskih sistema
● Kod zadržanog pisanja:
– pisanje se obavlja brzinom keša, a ne nižeg nivoa memorije
– potencijalno više zapisa u keš biva pokriveno jednim zapisom u niži nivo
– na taj način, ova politika zahteva manji propusni opseg memorijskog podsistema

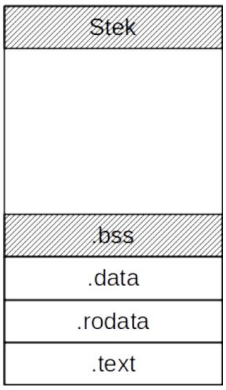
Keš i virtuelna memorija: interakcija

● Pošto postoje virtuelne i fizičke adrese, pitanje je kako ih razrešavati prilikom traženja linije u kešu
● Podsećanje, kad se odbaci ofset (u okviru linije) deo ostatka adrese služi za izbor skupa (kod skupovno asocijativnog keša)—označili smo sa S
● Taj izbor se zove indeksiranje, koje onda može biti virtuelno ili fizičko
● Ostatak adrese je tag, takođe može biti virtuelni ili fizički

Kombinacije indeksiranja i tagovanja
● Četiri moguće: PIPT, PIVT, VIPT, VIVT
● Problem fizičkog indeksiranja je što se virtuelna adresa mora prvo prevesti, što unosi kašnjenje, ali je ukupna implementacija jednostavnija
● Kod VIPT, indeksiranje se može pokrenuti u paraleli s prevođenjem, koje je potrebno za tag
● Virtuelno tagovanje se u praksi ne koristi zbog veće verovatnoće postojanja duplikata
● Jedna uobičajena implementacija: L1 je VIPT, L2/L3 su PIPT

Četiri pitanja primenjena na virtuelnu memoriju
● Ovde je blok = stranica
● Gde se stranica smešta u radnoj memoriji? Obično gde god ima slobodnog mesta (ekvivalent potpune asocijativnosti) zbog velikog troška čitanja sadržaja sekundarne memorije
● Kako naći stranicu? → Tabela stranica
● Koju stranicu zameniti? Najčešće LRU, ali postoji problem masovnog sekvencijalnog čitanja
● Politika pisanja? Isključivo zadržano

Idealizovana memorijska mapa procesa



Segmenti memorijske mape
● Svaki od segmenata ima određene privilegije: – R: samo čitanje – W: čitanje i pisanje – X: izvršavanje
● .text: instrukcije, R,X
● .rodata: konstante, R
● .data: promenljivi, inicijalizovani podaci, W
● .bss: promeljivi podaci inicijalizovani na vrednost 0, W (Block Started by Symbol), vode se posebno jer ne zauzimaju mesto u izvršnom fajlu

Primer memorijske mape
● U izvršnom fajlu: objdump -h /bin/bash
13 .text
41bb20
15 .rodata 4a6da0

24 .data 6de6a0
25 .bss 6e6aa0
● U virtuelnoj memoriji: /proc/$$/maps
400000-4dd000 r-xp
6dd000-6de000 r--p
6de000-6e7000 rw-p

Realnija situacija
● Raspored u virtuelnoj memorijskoj mapi zavisi od prevodioca
● Specifični raspored adresa neće sam po sebi uticati na performanse (tj. nije naročito bitno da li program počinje od 0x400000 ili 0x500000)
● Postoje tehnike koje namerno menjaju položaj segmenata u memoriji, kao što je ASLR (Address Space Layout Randomization)
● Relokacija i indirekcija svejedno moraju postojati zbog dinamičkog povezivanja

Statički i dinamički povezani programi
● Statički program sadrži u sebi sve komponente potrebne za izvršavanje
● Prednost: samodovoljan, pogodniji za distribuciju i instalaciju
● Mana: veliki broj programa sadrži delove koda koji su zajednički, iz sistemskih biblioteka
– ovo povećava prostor za skladištenje (disk/flash), danas nije toliko bitno
– povećan je utrošak memorije prilikom istovremenog izvršavanja
– komplikuje ispravku grešaka u zajedničkom kodu

Dinamičke biblioteke
● Biblioteke (skupovi zajedničkih funkcija) prevedeni i povezani na poseban način
● Koriste poziciono nezavistan kod, tako da mogu biti virtuelno mapirane na različite adrese u okviru različitih procesa, a da koriste istu fizičku memoriju
● Zbog toga moraju imati posebnu podršku za relokaciju i indirekciju poziva
● Konačno povezivanje i pokretanje programa radi dinamički linker

Primer dinamički povezanog progama
● Komanda: ldd /bin/bash
linux-vdso.so.1 => (0x00007ffe5e176000)
libtinfo.so.5 => /lib64/libtinfo.so.5 (0x00007ff5146e9000)
libdl.so.2 => /lib64/libdl.so.2 (0x00007ff5144e5000)
libc.so.6 => /lib64/libc.so.6 (0x00007ff514118000)
/lib64/ld-linux-x86-64.so.2 (0x00007ff514913000)
● Prvi red nema mapiranje jer je u pitanju specijalna dinamička biblioteka
● Poslednji red je dinamički linker

Primer deljenja fizičke memorije
● Dva procesa koja koriste istu dinamičku biblioteku:
7f90ed0f7000 … r-xp … libc-2.17.so
7f4a184f4000 … r-xp … libc-2.17.so
● Prevod prve virtuelne adrese u fizičku: Vaddr:
0x7f90ed0f7000, Page_size: 4096

PFN: 0x36a97
● Prevod druge virtuelne adrese u fizičku:
Vaddr: 0x7f4a184f4000, Page_size:
4096 PFN: 0x36a97
● PFN je Page Frame Number, fizička stranica

7. Višeprocesni rad i virtuelizacija

Terminologija

● Proces: kontekst za izvršavanje programskog koda
– obično se vezuje za pojedinačni korisnički program, ali može se odnositi i na druge vrste organizacije
– uključuje sve što je potrebno kao okruženje za izvršavanje
● memorijska mapa
● pristup sistemskim resursima (niži nivo)
● prisutp sistemskim servisima (viši nivo)
● Proces se lako mapira na sekvencijalni model izvršavanja
● Šta ako nam treba više procesa?
● Konkurentno izvršavanje
– i dalje sekvencijalno, u jednom logičkom lancu instrukcija
– vremenski podeljeno, tako da se svaki proces izvršava kratko vreme, pa pređe na drugi
– dve varijante vremenskog deljenja:
● kooperativno (cooperative multitasking)
● diktirano (preemptive multitasking)
● I dalje se može relativno lako rezonovati o efektima izvršavanja
– ali zgodno je imati izvesne garancije ponašanja radi uspešnijih apstrakcija i izbegavanja anomalija
● Paralelno izvršavanje
– više nezavisnih lanaca izvršavanja
– svaki od tih lanaca može biti podržan:
● potpuno nezavisnim fizičkim procesororom
● integrisanim ali nezavisnim procesorskim jezgrom
● skupom funkcionalnih jedinica u okviru jednog jezgra (Intel Hyper-threading)
● Uvodi potpuno nove probleme prilikom rezonovanja o efektima izvršavanja

Sistemska podrška za višeprocesni rad

● Svi procesi bi mogli da se izvršavaju u istom globalnom adresnom prostoru
– na sistemima kao što je MS/PC-DOS i originalni MacOS to i jeste bio slučaj
● Problem: šta ako proces zbog greške ili iz zle namere poremeti memoriju drugog procesa?
– već smo razmatrali: upravljanje memorijom/MMU
● Sledeći problem: šta ako proces iz istih razloga poremeti MMU tabele?
– jedan nivo indirekcije do memorije i resursa drugih procesa

Privilegovani režim rada

● Uvodi se poseban, privilegovan, režim rada procesora
– ovo deli način izvršavanja na najmanje dva dela: neprivilegovan i privilegovan
● U privilegovanom režimu moguće je pristupiti sistemskim resursima, koji postaju nedostupni u neprivilegovanom režimu
– MMU tabele

– prekidne tabele
– U/I portovi
● Privilegovani kod se najčešće grupiše u celinu koja se zove kernel

Nivoi privilegija
● Privilegovani režim može imati više nivoa privilegija, u najprostijoj varijanti samo jedan
● Nivoi se u standardnoj nomenklaturi zovu
– hijerarhijske oblasti zaštite (hierarchical protection domains)
– prstenovi zaštite (protection rings)
● Recimo, sa četiri prstena (x86, amd64):
– prsten 0: kernel
– prsten 1: drajveri
– prsten 2: privilegovani korisnički procesi
– prsten 3: neprivilegovani korisnički procesi
● Linux, Windows NT+, moderni macOS: 0 + 3

Tipična memorijska mapa

Prelazak između nivoa
● Neprivilegovanom kodu su potrebne usluge sistema
● Mora postojati način da se (kontrolisano) pređe na izvršavanje privilegovanog koda
● Za x86/amd64:
– softverski prekid: int 0x80
– SYSENTER/SYSEXIT (x86, 32-bitni režim)
– SYSCALL/SYSRET (amd64, 64-bitni režim)
● Menja memorijske mape!

Prelazak između procesa
● Ako je svaki korisnički proces slično mapiran, MMU tabele moraju da se menjaju prilikom prelaska na sledeći proces
● Isto važi i prilikom prelaska između neprivilegovanog i privilegovanog režima rada – korisnički proces ne sme da ni da vidi, a kamoli da menja kernelske strukture podataka
● Problem: TLB zapisi, jer je TLB globalni resurs

Rukovanje TLB zapisima
● Najjednostavnije: zaboraviti sve TLB zapise prilikom prelaska
– ali neefikasno: primera radi, kernelske mape su uglavnom fiksne, a sistemski poziv ne mora da poremeti veliki broj mapiranja
● Malo bolje: omogućiti individualno zaboravljanje TLB zapisa
● Još bolje: proširiti tagove za TLB identifikatorima adresnih prostora

– kernelski i korisnički prostori dobijaju dodatni (kratak) identifikator
– prostor za identifikatore je jako mali, jer je TLB keš vrlo ograničen i kritičan resurs
– čak i jedan dodatni bit popravlja situaciju

Dodatni motiv: virtuelizacija

● Apstrakcija čitave hardverske platforme, tako da jedna fizička mašina može da izvršava više virtuelnih mašina
● Ako je postupak efikasan, ovo je dobar način da se iskoriste hardverski resursi
● Svaka virtuelna mašina ima sopstveni operativni sistem
● Mora postojati komponenta koja se izvršava na osnovnom nivou i vodi računa o ostalim VM: zove se hipervizor (hypervisor) ili VMM (Virtual Machine Monitor)

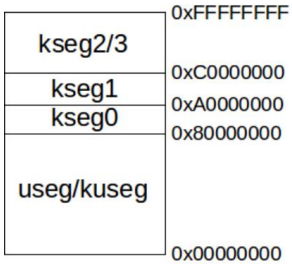
Zahtevi za virtuelizaciju (Popek/Goldberg)

● Ekvivalentnost/Vernost
– program koji se izvršava pod hipervizorom ponaša se identično kao i na fizičkoj mašini
● Kontrola resursa/Bezbednost
– hipervizor mora u potpunosti da kontroliše virtualizovane resurse
● Efikasnost/Performanse
– statistički dominantan deo mašinskih instrukcija mora se izvršavati bez intevencije hipervizora

Načini virtuelizacije

● Puna virtuelizacija
– potpuna imitacija hardverskog okruženja, operativni sistem unutar VM može se izvršavati bez modifikacija
● Hardverski potpomognuta virtuelizacija
– omogućava punu virtuelizaciju uz pomoć hardverskih sklopova na fizičkom procesoru
● Paravirtuelizacija
– zahteva modifikaciju operativnog sistema tako da eksplicitno sarađuje sa hipervizorom

Ilustracija: MIPS-32



MIPS-32, inicijalizacija

● Ima dva obavezna režima rada, kernelski i korisnički, sistem se posle resetovanja budi u kernelskom režimu
● U memorijskoj mapi pojedini segmenti se mogu referencirati samo iz privilegovanog režima (iznad 0x80000000)
● Takođe, segmenti se razlikuju prema tome da li se keširaju i mapiraju kroz MMU
– kseg0: kernelski, nemapirani
– kseg1: kernelski, nemapirani, nekeširan
– kseg2/3: kernelski, mapirani
– useg/kuseg: korisnički/kernelski, mapirani
● Kod ovog procesora, i keševi i MMU se moraju inicijalizovati softverski

8.Paralelno izvršavanje

Podsetnik

● Paralelno izvršavanje = više zaista nezavisnih lanaca izvršavanja instrukcija
– pretpostavka: svaki od ovih lanaca vidi istu (do izvesnih granica) sliku memorije
– u suprotnom, sistem je distribuiran
● Konkurentno izvršavanje = sukcesivno izvršavanje odlomaka lanaca instrukcija vezanih za pojedinačne procese tako da se dobije iluzija istovremenog rada
● Konkurentno i paralelno izvršavanje se mogu kombinovati

Terminologija

● Proces = kontekst izvršavanja (od ranije)
● Sužena definicija: kontekst koji ima:
– nezavisnu mapu radne memorije
– nezavisne pokazivače na izvesne sistemske resurse (npr. deskriptori za pristup fajlovima i mrežnim konekcijama)
● Lanac = "Thread":
– konkurentno izvršavanje u kontekstu jednog procesa
– deljena slika memorije
– deljeni deskriptori

Implementacija

● Klasični Unix:
– proces se dobija sistemskim pozivom fork()
– za "thread" se pokreće LWP (lightweight process)
● Linux:
– primitivna operacija je clone(), koja može/ne mora da deli adresni prostor
– fork() se izvodi iz clone()
– korisnička konkurentnost je i dalje moguća

Osnovni problem paralelnog rada

● Da bi bili korisni, procesi moraju na neki način međusobno da komuniciraju
● Najjednostavnija komunikacija je preko zajednički vidljivog bloka memorije
● Problem je očuvanje sekvencijalne semantike izvršavanja pri mogućnosti paralelnih izmena memorije
● Sve manifestacije paralelnog rada se na kraju posmatraju u skladu s uticajem na ovaj problem

Trivijalna ilustracija

● Šta se sve može desiti prilikom paralelnog izvršavanja ovog koda?
– potpitanje: a šta sa konkurentnim izvršavanjem

```
flag = False;
 if not flag:
        flag = True
        # kritični deo
        print('uradi samo jednom')
```

Garancije pristupa

● Generalno, želimo da postignemo da se kritičnom delu može pristupiti samo iz jednog procesa
● Postoje različite strukture podataka i protokoli njihovog korišćenja koje ovo omogućavaju
● Hardver mora da pruža neku vrstu garancije, inače je čisto softversko rezonovanje nemoguće

Generička struktura: semafor

● U opštem slučaju, brojač koji upravlja pristupom nekom resursu
● Dokle god je vrednost brojača veća od nule, pristup je moguć
● Dve operacije:
– umanjivanje (operacija P): ako vrednost za jedan. Ako je posle ovoga vrednost negativna, smesti proces u red i čekaj
– uvećavanje (operacija V): uvećaj vrednost za jedan. Ako je vrednost bila negativna, uzmi prvi proces iz reda čekanja
● Semafor ima početnu vrednost veću od nule koja kaže koliko je instanci resursa na raspolaganju

Muteks

● Binarni semafor = muteks (mutex: mutual exclusion)
● Samo jedna instanca resursa je na raspolaganju
● Veoma često korišćena struktura, u implementaciji i praktičnom rezonovanju se tretira posebno u odnosu na generički semafor

Praktična implementacija

● Atomičke operacije: procesor(i) garantuje/u da tokom njihovog izvršavanja samo jedan proces na čitavom sistemu ima pristup nekom resursu, obično memorijskoj lokaciji
● Nekoliko mogućih operacija:
– proveri i postavi (test-and-set): postavi vrednost lokacije na 1, vrati staru vrednost
– preuzmi i uvećaj (fetch-and-add)
– uporedi i zameni (compare-and-swap)

Proveri i postavi

● Muteks implementiran pomoću ove instrukcije (pseudokod)

```
while test_and_set(mutex) == 1:
        pass
# kritični deo
print('uradi samo jednom')
# oslobodi muteks
mutex = 0
```

Sinhronizacija memorijskog pristupa

● Procesori zbog povećanja performansi smeju da preuređuju redosled izvršavanja instrukcija
– ako zaključe da naredna instrukcija u nizu ne utiče na rezultat cele sekvence
– problem: ovo zaključivanje je lokalno
● U prethodnoj implementaciji, mora se postići da se mutex postavi na nulu tek kada su svi prethodni memorijski pristupi završeni
● Način da se ovo postigne je uvođenje memorijskih barijera

– garancije da će promene i/ili pristupi memoriji biti lokalno ili globalno vidljivi u određenom redosledu
– granularnost i efektivnost zavise od procesora
– npr. x86 ima jače osnovne garancije u odnosu na ARM procesore

Dodatne komplikacije
● Višeprocesorski sistemi: sinhronizacija keševa
– sve promene u jednom kešu moraju se propagirati ostalima
– redosled promena vrednosti mora ostati očuvan
– različiti protokoli za koherentnost keševa
● Takođe: sinhronizacija MMU mapiranja za virtuelnu memoriju
– ako se mapiranje promeni na jednom sistemu, TLB zapisi se moraju očistiti svugde (TLB shootdown)

Opšti tretman paralelizacije
● Amdalov zakon daje teorijsku granicu ubrzanja izvršavanja nekog zadatka pri poboljšanju raspoloživih resursa
● Često korišćen pri proceni dobitaka od paralelizacije
● Ključno: nisu svi delovi nekog problema podložni paralelizaciji
● Parafrazirano: ako je dobitak u paralelizaciji manji od troška paralelizovanja, ne isplati se

Drugačiji pristupi konkurentnosti
● CSP model (Communicating Sequential Processes)
● Formalizam za predstavljanje načina interakcije u konkuretnim sistemima
● Osnova je da procesi nemaju zajednički vidljivo promenljivo stanje
● Komunikacija se odvija pomoću kanala, procesi su anonimni
● Primer implementacije: programski jezik Go
● Aktorski model
● Takođe predstavlja model za konkurentnu interakciju i programiranje
● Ima sličnosti sa CSP: komunikacija je takođe putem poruka, samo je lokalno stanje aktora promenljivo (nema globalnih promena)
● Razlike: aktori su imenovani, procesi ne; prenos poruka je potpuno asinhron; kanali za prenos su implicitni
● Primer implementacije: programski jezik Erlang
● Transakciona memorija
– grupe memorijskih pristupa se tretiraju slično transakcijama kod baza podataka
– moguće su i hardverska i softverksa implementacija
● Funkcionalno programiranje
– veoma širok pojam
– ključan je oslonac na nepromenljive strukture i njihovu transformaciju
– programski jezici kao što su Haskell, (donekle) Scala i Rust