

W. Stallings: "Operating Systems Internals and Design", 7th edition, Pearson 2007

A. Silberschatz, P. B. Galvin, G. Gagne: "Operating System Concepts",
7th edition, J. Wiley, 2005

L. F. Bie, C. Shaw: "Operating System Principles": Prentice-Hall 2003

ZNAČILNOST PROGRAMSKE OPREME V TK SISTEMIH

Včasih: koračni zbiralniki, specifična program. oprema
posebni programski jeziki SDL

Danes: računalnik v sistemu: vsebujejo programsko opremo; bolj
podobna vseh računalniških opremih

splošno namenski operacijski sistemi

centralni del: switch krmiljen z račun. strežniki - telekomunikacij.

Značilnosti

(NAVIDEZNA SOČASNOST)

① VEDNO TEČE VZPOREDNO VEČJE ŠTEVILU OBDELAV (VZPOREDNIH AKTIVNOSTI)

v TK sistemu vedno teče več strank hkrati

② PROGRAMI TEČEJO V REALNEM ČASU

a) soft real time - običajno

b) hard real time system

③ PROGRAMSKA OPREMA JE PORAZDELJENA PO VEČJEM ŠTEVILU RAČUNALNIKOV

④ PROGRAMI NAPISANI V VIŠJH PROGRAMSKIH JEZIKIH IN UPORABIMO
STORITVE PORAZDELJENIH SISTEMIH

⑤ PROGRAMI UPORABLJAJO DISTRIBUIRANE PODATKOVNE BAZE

⑥ PROGRAMI UPORABLJAJO NEKO SPECIFIČNO PROGRAMSKO OPREMO
odstopanje od drugih?; nalaganje, konfiguracija

⑦ PROGRAMI MORAJO IZPOLNJEVATI STROGE ZAHTEVE GLEDE ZANESLIVOSTI
koliko časa sme izpast telefonska centrala v enem letu

⇒ PROGRAMSKA OPREMA RELATIVNO DISTRIBUIRANEGA SISTEMA

↳ moramo najprej poznati elemente & značilnosti operacijskega
sistema enega računalnika

SOCASNOST: moramo poskrbeti koordinacijo pride do interakcije: isti vir

⇒ takrat pride do težave

SEKVENČNI PROGRAMI: pride do napak

↳ ODRAVLJANJE NAPAK: trace

VZPOREDNO ODRAVLJANJE NAPAK: večkrat se programi izvajajo na drugačen način ⇒ težko odpravljanje napak

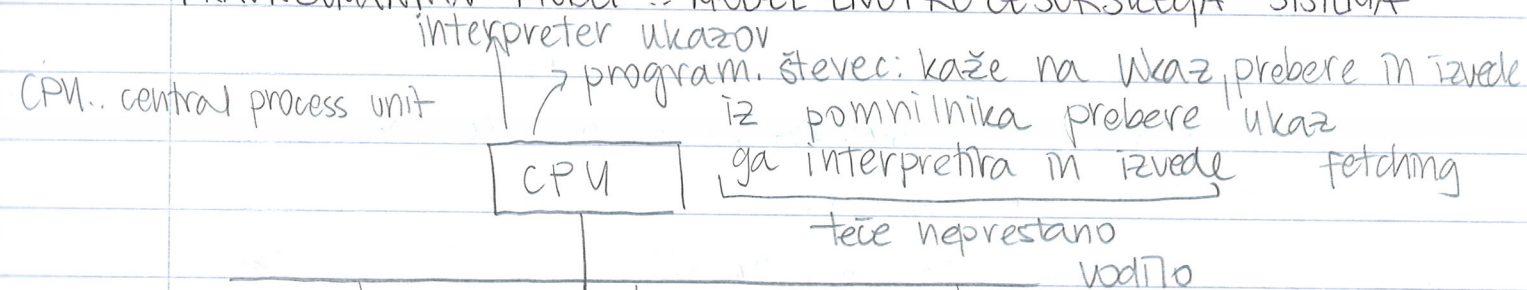
⇒ REŠITEV: PROGRAMSKA OPREMA vsilil NAČIN ZA ODRAVLJANJE NAPAK: PASCAL, C: pravila

STRUKTURA: pravila za odpravljanje napak, za nadzor
↳ naša naloga

OPERACIJSKI SISTEM: program, ki mora reagirati na dogodke, ki niso zanesljivi

ADAPTIVNA OPREMA

FRAUNHEIMANNOV MODEL - MODEL ENOPROCESORSKEGA SISTEMA



MULTIPROGRAMIRANJE

več uporabnikov naenkrat v obdelavi/pomnilniku, čeprav imamo samo enega na procesorju

⇒ Osnovna metoda za vse operacijske sisteme

V Fraunheimannovega modela tega ni mogoče napraviti

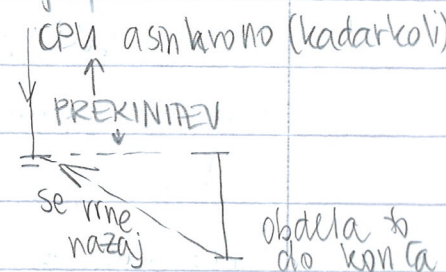
⇒ DODATNA APARATURA/MEHANIZMI

OBSTOJ VHODNO-IZHODNEGA KANALA

- CPU I/O kanalu samo da ukaz (izvede (sproži) operacijo) in potem CPU je prosta, I/O kanal pa mora to izvesti do konca (teče avtonomno). Potem ko izvede to operacijo do konca, o tem obvesti CPU
- CPU & I/O delujeta sočasno, asinhrono

OBSTOJ PREKINITVENEGA SISTEMA → to gledamo kot dogodek

- zunanje enote lahko o določenih dogodkih obvestijo CPU preko posebnih signalov, ki jim rečemo PREKINITVENI SIGNALI
- ta mehanizem se izvaja med izvajanjem vsakega ukaza posebej. Ko CPU zazna te signale, preneha z redno obdelavo in skoči na neko novo obdelavo.
- podobno kot preskok na podprogram in vemo v katerem delu programa je prišlo do podprograma. Tukaj pa ne vemo kdaj se to zgodi, lahko pride kadarkoli



OBSTOJ ČASOVNEGA PREKINITVENEGA SIGNALA

Ura, ki periodično /aperiodično sproži nek prekinilveni signal

Doseže, da bo procesor preskočil na operacijski sistem

↳ OS dobi nadzor nad procesorjem

Imamo en procesor na voljo, ki si ga morajo programi deliti

Vsak program dobi procesor na določen čas.

Kaj je z OS? $\Rightarrow$ Zato OS (nadzornik) je ta časovni

PREKINITVENI SIGNAL

$\Rightarrow$ časovni prekinilveni signal poskrbi, da OS dobi procesor

~~OS~~ ENOTE ZA ZAŠČITO POMNILNIKA: MEMORABLE UNIT

Problem: zaščita pomnilnika

Programi, ki morajo biti med seboj zaščiteni, ^{ker} uporabnik lahko paravi OS.

Uporabniku je dodeljen določen del pomnilnika in čim skuša generirati nekaj izven tega dela, se sproži prekinitev.

Ko nek uporabnik dobi procesor, se hkrati nastavijo parametri za zaščito pomnilnika.

Pri prehodu iz enega na drugega uporabnika je treba spremeniti okolje

OBSTOJ PRIVILEGIRANIH/SISTEMSKIH UKAZOV PROCESORJA

procesor ima razdeljene ukaze na vsaj 2 skupini

1) ukazi, ki jih lahko izvaja vsak - UPORABNIŠKI UKAZI

2) ukazi, ki jih lahko izvaja OS - PRIVILEGIRANI UKAZI

$\hookrightarrow$ deluje lahko OS

3) ukazi, ki jih uporabnik ne sme izvajati

- N ne sme spreminjati parametre zaščite pomnilnika

- N ne sme brati podatke iz diska preko I/O enote

- prehod iz uporabnika na privilegirani način

- prekinilveni sistem

DMA: NEPOSREDNI DOSTOP DO POMNILNIKA

Direct Memory Access

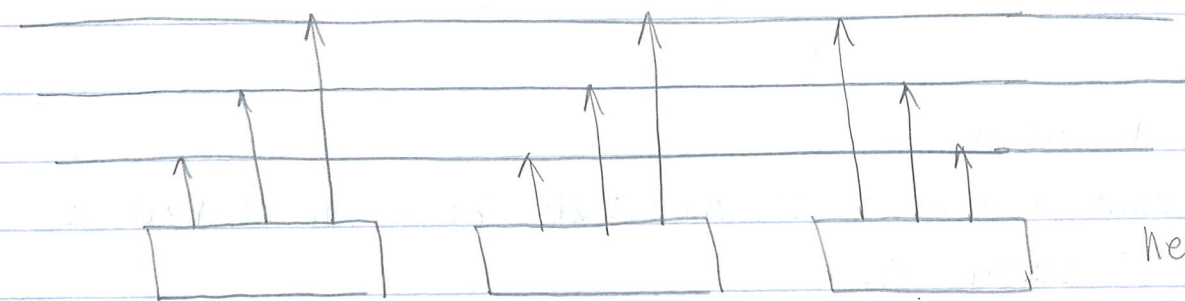
Če hočemo prenesti podatke iz okolice I/O unit na M

lahko bi vse preko CPU: prepočasno

CPU samo da ukaz in ta proces teče avtonomno

Prenos iz diska v centralni pomnilnik: CACHE

SISTEMSKA VODILA



Prenašanje med enotami preko vodila

neposredno
povezav
 $\frac{n(n-1)}{2}$

Vzporednost $\rightarrow$ prepletanje $\rightarrow$ morajo pravilno delovati
interakcija

$\hookrightarrow$ na določenih mestih opustimo vzporednost in vsilimo
zaporednost

$\hookrightarrow$ na nekem mestu moramo imeti ukaze, ki se izvajajo
kot neprekinjene enote

Na vodilu pride do interakcije: kako

Kako dosežemo to neprekinjenost:

PODATKI NA VODILU

- lahko oddaja le ena naprava (zaradi kolizij)

- uporablja se pesimističen dostop: želimo doseči, da ni kolizij

- sprejemnika dobimo z adresno, broadcastinga pri sistemskih vodilih

- Adreso določa bus muster (določi adresno, oddajnik odda pakete, Sprejem. sprejme)

- Odnos med masterjem (mojster) in ostalimi

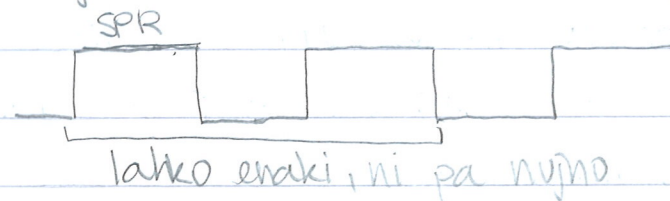
- Synchronizacija: naprava da adresno, potem podatke, spr. jih sprejme

Kako oddajnik ve kdaj je N sprejel podatke in kdaj
nadaljuje z oddajanjem?

$\Rightarrow$ rešitev: PROTOKOLI VODILA

① SINHRONI PROTOKOL VODILA

Vpeljemo sistemsko vro: niz pravokotnih signalov



Dva dogovora:

- 1) Sprejemnik opazuje vodilo samo takrat, ko je v visokem stanju
- 2) V tem visokem stanju mora tudi prebrati podatke

⇒ Oddajnik postavi podatke na vodilo kadar je ura v nizkem stanju i vsi prehodni pojavi izvenijo

V naslednjem stanju lahko naprava odstrani oddajnik iz vodila

SLABOSTI: - v visokem stanju morajo vse enote v vodilu prebrati podatke

↳ problem: različne hitrosti

⇒ to vodilo mora biti prilagojeno najpočasnejši enoti

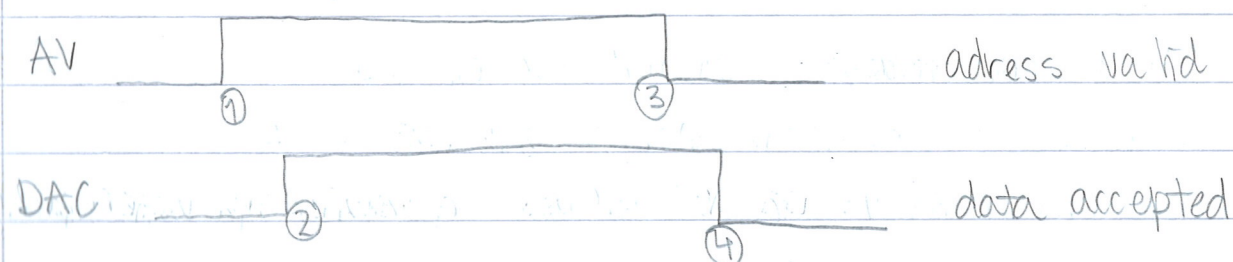
Prenosi potekajo v urinih taktih

↳ ko izvajamo v procesorju neke ukaze, potem ta ukaz tvori običajno večje število prenos podatkov

PROBLEM: vodilo ne ve kdaj je začetek in konec ukaza

② ASINHRONI PRENOS PODATKOV

Shema rokovanje: HANDSHAKE



- ① oddajnik postavi adresno na vodilo in v ① sporoči, da je adresa na vodilu pravilna.

Ko sprejemnik to vidi, prebere podatke iz vodila in v ② sporoči, da je sprejel podatke.

Ko oddajnik opazi, da so podatki sprejeti, umakne svoj signal (v ③) in ko sprejemnik to opazi umakne še svojega (v ④)

ODDAJNIK & SPREJEMNIK SE OBVEŠČATA O TEM

SLABOSTI: - dva dodatna kontrolna signala

- če oddajnik hoče komunicirati z neobstoječo napravo, oddajnik lahko v nedogled čaka

• Več. procesorjev = mojster Bussmeister

Protokol, ki v vsakem trenutku določi en mojster vodila in

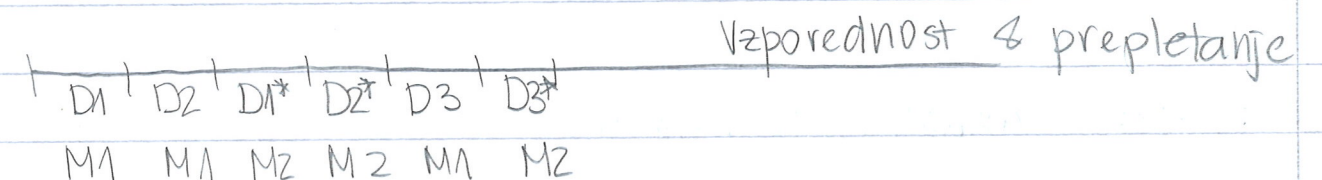
poseben del hardverja

ARBITRAŽA VODILA

V vsakem trenutku sprejema odločbe od mojstrov in odloči, da vsakemu dodeli procesor

- V vsakem trenutku določi enega mojstra vodila

- ne ozira se na to kdaj se izvaja en ukaz na enem in kdaj na drugem procesorju



del izvajanja na procesorju 1 (M1): D1, D2, D3

del izvajanja na procesorju 2 (M2): D1*, D2*, D3*

- Arbitražna vodila ne ve kateri proces se izvaja na katerem vodilu

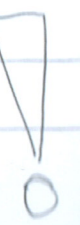
⇒ doseže, da prenos podatkov tečejo v redni

Lahko pride do prepletanja: Dva ukaza na enem vodilu

V določenih primerih do tega ne sme pridet

↳ da se določeni ukazi izvajajo neprekinjeno na vodilu

↳ da do prepletanja ne pride



14.10.12

Ukazi v pomnilniku: lahko binarni (0 ali 1)

↳ računalniki interpretirajo te ukaze (strojni jezik)

Včasih: računalniki so bili programirani v strojnem jeziku

Danes se tega ne počne

↳ za olajšanje so se razvili programski jeziki

↳ višji programski jeziki, ki so bližji človeku

⇒ Program, ki je napisan v višjem programskem jeziku je treba prevesti v nižji programski jezik

⇒ obstajajo prevajalniki, zbirniki, interpreterji, assemblerji

TRANSLATORJI

Največkrat jih računalnik izvaja, včasih pa tudi interpretira

Prvi programski jezik: Fortran: Formula translator

Ključno za višje programske jezike:

↳ vsak je namenjen svojemu področju in vsakemu ukazu pripada od 100-1000 strojnih ukazov

Jeziki algorskega tipa: Pascal, C

Večina programskih jezikov omogoča PROGRAMIRANJE ZAPOREDNO

IZVAJAJOČIH SE PROGRAMOV (UKAZ ZA UKAZOM): PROCEDURALNI

PROGRAMSKI JEZIKI: procesor interpretira naš ukaz

Lahko opišemo tudi sočasno izvajajočih se programov

↳ to omogočajo nekateri programski jeziki

↳ V C-ju in Pascal-u te možnosti ni

↳ mi si bomo izmislili neke dodatne ukaze

↳ V SDL-u je to možno

Obstajajo dogodkovno krmiljeni & dogodkovno -gnani programski jeziki

• Program, ko je vrstni red drugačen (sam program na bazi dogodkov): ko ima na voljo podatke

↳ Labview event driven programming

Vzporednega programiranja v večini programskih jezikov se ne da opisati (Java, SDL) ⇒ Naloga operacijskega sistema je da to omogoči.

Kak programski jezik omogoča uporabo storitve operacijskega sistema

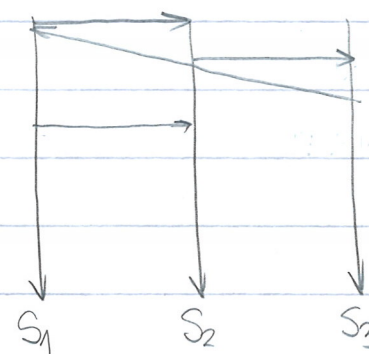
⇒ Sočasno izvajanje se opiše v programskem jeziku ali se to zahteva od operacijskega sistema (večji nadzor)

Dodatni konstrukti (programski) za opis vzporednega izvajanja

① cobegin $S_1/S_2/S_3$ coend

računalnik mora izvajati S_1, S_2, S_3 sočasno

enoprosesorski računalnik: program se ne more izvajati sočasno



prepletajoče izvajanje programov

↑ enoprosesorski: namišljena vzporednost

NAVIDEZNA VZPOREDNOST: Vsi programi bodo v izvajanju

↳ Večprocesorski: Vsak: S_1, S_2, S_3 izvajamo na ločene procesorje

DEJANSKA VZPOREDNOST

⇒ s stališča logične zgradbe programe je vseeno ali je dejanska ali navidezna vzporednost.

② Programi, ki se izvajajo v neskončnih zankah (krožno).

while true do S

for (i) S (v C-ju) $\equiv$ loop S end

NESKONČNA ZANKA

③ ČAKALNA ZANKA: program čaka na nek dogodek

while B do i dokler je pogoj izpolnjen čaka

DELOVNO
ČAKANJE

④ ŠTEVEC PO MODULU N

0 1 2 3 N 0 1 2
 $z = x + y \equiv \text{if } x + y < N \text{ then } z := x + y$
 $\uparrow$
 else $z := x + y - N$ (Pascal)

seštevam x & y
 po modulu N

$z \leftarrow x + y$

$N = N + 1$ (C)

$N := N + 1$ (Pascal)

↳ znak prirejanja

razlika:

$z = x - y \equiv \text{if } x - y > 0 \text{ then } z := x - y$
 $\uparrow$
 else $z := x - y + N$
 negativna vsota

⑤ Elementi množice L

za vsak element je treba nekaj narediti

for all $p \in L$ do S
 PEL

for all $p \in L$ such that B do S
 ki imajo lastnost B do S

⑥ Simboli

$\wedge$: AND

$\vee$: OR

$\neg$: NOT

C: AND: lahko logična
 operacija
 oper. nad spremenljivkami
 oper. nad biti

ASSEMBLER: ZBIRALNIK

↳ bodisi programski jezik ali translator

↳ mnemonična oblika strojnega jezika

Napišemo nekaj z nekimi razumljivimi izrazi in ASSEMBLER
 TO PREVEDE

Včasih so programirali v zbiralniku, vendar je to počasno

Vsak procesor ima svoj zbiralnik. Pri operacijskih sistemih se temu ne moremo izogniti

Vkaz v strojnem jeziku: sestavljen iz dveh delih:

A) OPERACIJSKA KODA: kaj mora ta ukaz napraviti

B) NASLOVNI DEL: kje so podatki, kam bodo šli rezultati (labela)

Oznaka adrese	
OC	ADR
1101..	101111

↳ Simbolično ime: seštevanje: ADD

MPI

JMP → preskoči

} mnemoničen del
 } nabor možnosti
 } fiksni del

↳ simbol: A1, A2 (vsebinska lokacije A1, vsebinska lokacije A2)

LAB 1

Enemu ukazu v zbiralniku pripada natanko en ukaz v strojnem jeziku.

PREDNOST: - imamo na voljo vse možnosti procesorja

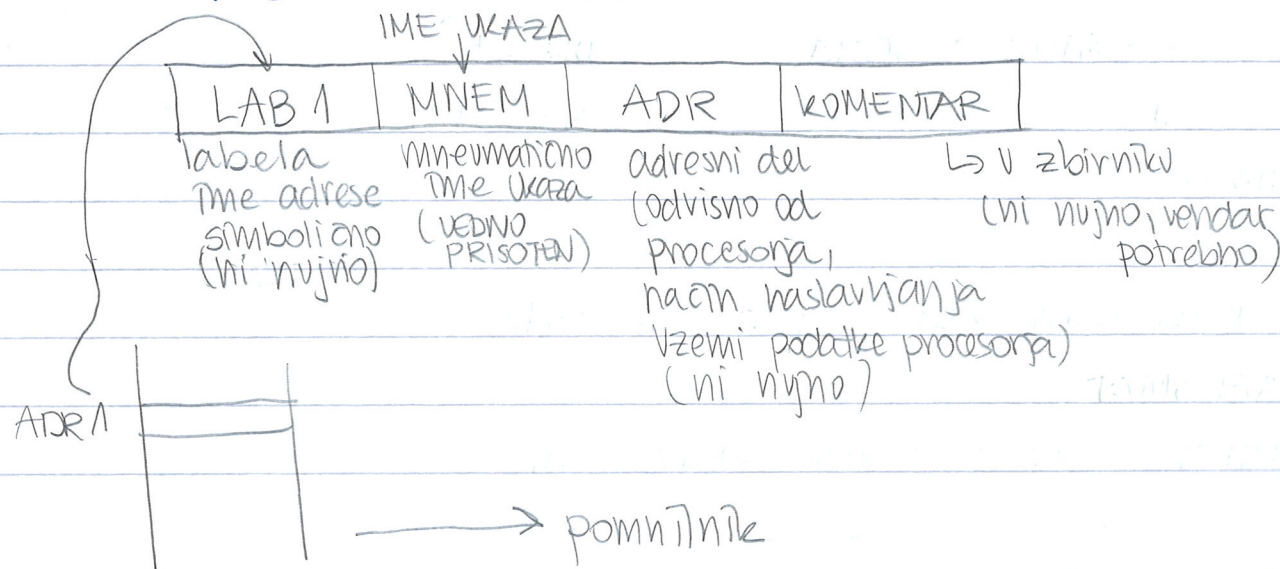
↳ v višjem programskem jeziku tega ne moremo

translator: prevaja iz višjega programskega jezika v strojni

↳ težko prevaja adres

Prevajalniki prevajajo v zbirnik, potem v strojni jezik.

Vkaz v zbirniku ima vedno isto strukturo.



Različni računalniki imajo različne zbiralnike!

Zgradba procesorja: pomembna za programiranje v zbirniku

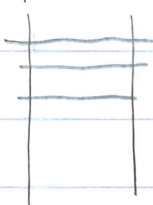
PROGRAMSKI MODEL PROCESORJA (ZBIRNIK)

① ZGRADBA POMNILNIKA

Sestavljen iz posameznih delov, ki jih lahko eksplicitno naslavljamo

A) podatek: dolžina besed: 16 bit, 32 bit, 64b.

↳ posamezni kosi pomnilnika so 16b, 32b, 64b.



B) na kakšen način adresiramo lokacije

pr: št. adres, št. glede na trenutno lokacijo
del adrese v nekem registru

② REGISTRI - pomnilne celice

Znotraj CPU je večje število specialnih pomnilnikov, ki so posebno dosegljivi ⇒ REGISTRI. ↳ uniformnih

procesor skuša spremenljivko čim dlje držati v registru, ⇒ ni treba brati iz pomnilnika

③ TIP PODATKOV

Kakšne vrste podatkov pozna procesor

Pomnilne celice v binarnih stanjih 0,1

↳ to vsebino lahko interpretiramo na različne načine (št: integer, floating (plavajoča vejica), ASCII, samo biti)

procesor ima vgrajene ukaze, ki to interpretirajo

↳ na isti lokaciji uporabljamo različne ukaze

④ UKAZI

kaj lahko počne procesor nad temi podatki

⑤ POSEBNOST

prekinitveni sistem, zaščitni mehanizmi, ...

POVEZOVALNI NALAGALNIK (Linking loader)

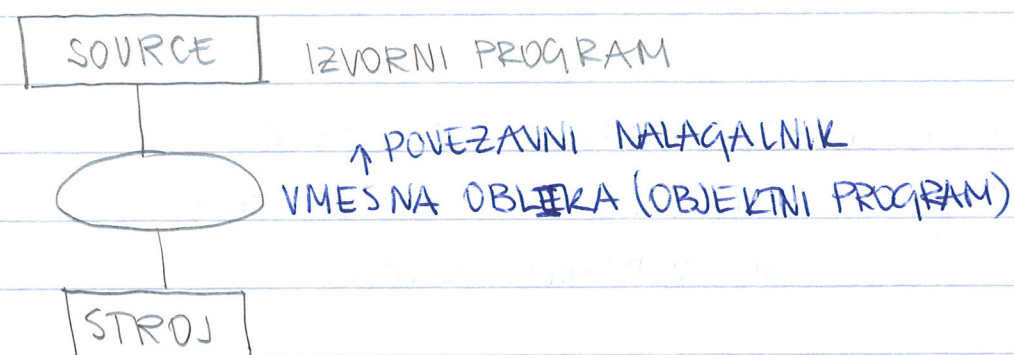
Običajno programi se pišejo v več kosih (modulih) in v različnih programih

⇒ te module sestavljamo z linking loader

POMEMBNO: program v programskem jeziku ⇒ faza prevajanja

KONČNI cilj: strojno prevajanje preko vmesne oblike

Povezovalni nalagalek vzame vmesno obliko kot svoj vhod



POMEMBNOST:

- Vsi prevajalniki prevedejo v vmesno obliko, ki ima standardni format in je opremljena s standardnimi podatki

Naloga vseh prevajalnikov: prevedejo vse module v vmesno obliko
↳ v Javi se uporablja za vmesni interpreter

Zakaj imamo ta povezovalni nalagalek?

Program v več modulih $\xrightarrow{\text{PREVOD}}$ strojna oblika
↳ moramo shraniti na procesor

ABSOLUTNI NALAGALNIK

ADR1					
ADR2					

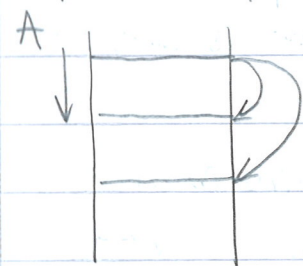
Najprej shrani ADR1 in vse ostalo po vrsti, potem ADR2 in vsa polja, ki sledijo

Programer mora specificirati v katerem delu se bo program nahajal
↳ fja dodeljevanja pomnilnika & fja nalaganja
⇒ del pomnilnika mora biti najprej definirana

Programi so sestavljeni iz več modulih: ne moremo povedati adreso

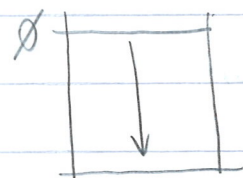
⇒ težko najprej napovedati kje se bo nahajal

Zato v trenutku, ko program naložimo v pomnilnik, vse module zapakiramo, da so en za drugim.



Ko prevajamo program, ne vemo kako dolgi so in koliko prostora potrebujejo ostali moduli

⇒ Naredimo tako, da jih pišemo in prevedemo na neko fiksno adreso.



Te module je treba sestaviti = POVEZOVANJE in

nekatero programske module je treba prestaviti/prenasloviti = PRENASLAVLJANJE

POVEZOVANJE: | ➡ moduli morajo biti povezani

Funkcije povezovalnega nalagalnika

- mora dodeliti pomnilnik (mora znati določiti koliko pomnilnika)

↳ dobi podatke o dolžini posameznega modula

- mora znati prenasloviti posamezne programske module

↳ podatki o posameznih lokacijah v modulu (to ima vmesna oblika)

- ta program mora imeti podatke za povezovanje

- vse to naloži v pomnilnik

VRSTE NALAGALNIKOV

A) Absolutni

B) Nalagalik/sistem tipa prevedi & izvedi (compile & go)

↳ imamo sočasno v pomnilniku prevajalnik in vse programske module. Prevajalnik hkrati prevaja vse te programske module, spravi nalaga v pomnilnik in začne z izvajanjem

SLABOSTI - prevajalnik mora biti v pomnilniku (zasede del pomnilnika)

- Vsakokrat ko skušamo ta program izvesti, ga je treba ponovno prevesti

ZBOLŠANJE ➡ program mora biti napisan v istem programskem jeziku
➡ vsak programski modul prevedemo v objektno obliko, shranimo na disk in pred izvajanjem vse to skupaj povežemo

POMEMBNO: PRENASLAVLJANJE: pred začetkom izvajanja napravimo te operacije. Ta program je sestavljen pred začetkom izvajanja.

1) STATIČNO PRENASLAVLJANJE: PRED ZAČETKOM IZVAJANJA

Programi se pogostokrat menjajo v pomnilniku.

↳ posamezne kose pišemo na disk; težko naprej rezervirati prostor

Računamo, da so se posamezni kosi pred izvajanjem "sprehajali" po pomnilniku.

⇒ zato uporabljamo DINAMIČNO PRENASLAVLJANJE

↳ dejanska adresa vsakega kazalca se določi MED IZVAJANJEM KAZALCA ⇒ posebna aparaturna oprema ⇒ fleksibilnost

OBJEKTHO PROGRAMIRANJE

Poskus kako obvladati vse kompleksnosti pri programiranju

Metodi za zmanjševanje števila programskih napak

1) s programskimi jeziki (Go - starki)

↳ prisili za PRAVILNO pisanje programe (pravilno strukturiranje)
programi morajo biti robustno napisani (testiranje)

- objektno napisani programi imajo zelo jasno strukturo

- Programski jeziki, ki podpirajo objektno programiranje:

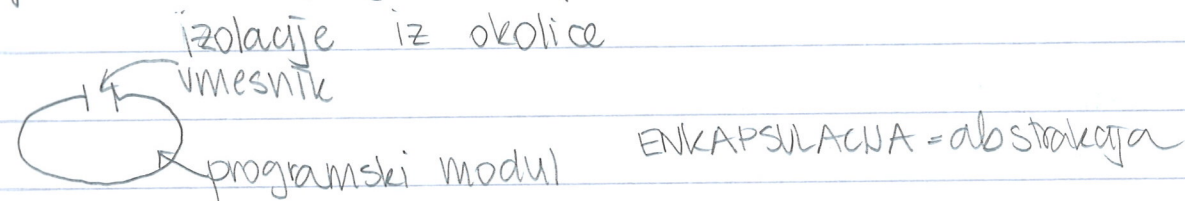
SmallTalk (čisto objektno orientiran), C++ (C ne podpira),

JAVA, Pascal

Operacijski sistem Windows je napisan v objektnem programiranju! !

NAČINI KAKO OBVLADOVATI KOMPLEKSNOŠTI PROGRAMIRANJA

-omejtev svobode programiranja = ABSTRAKCIJA: možnost,

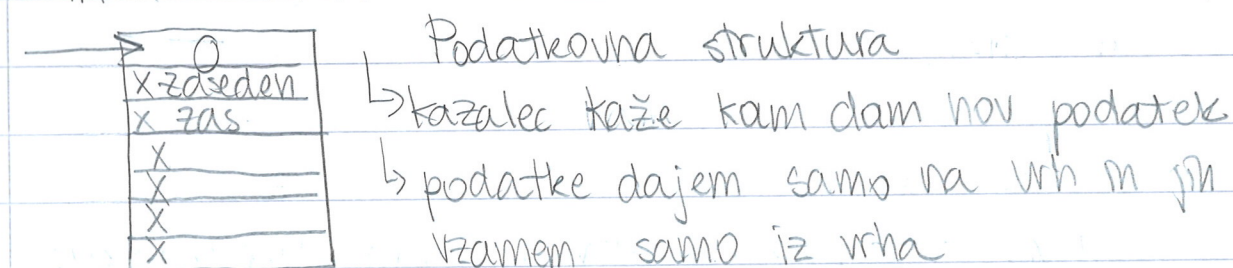


A) UPORABA PODPROGRAMOV / PROCEDURE

(programske module enkapsuliramo v procedure)
 → kos programa ki ga kličemo - to ni čisto jasno
 definiran vmesnik

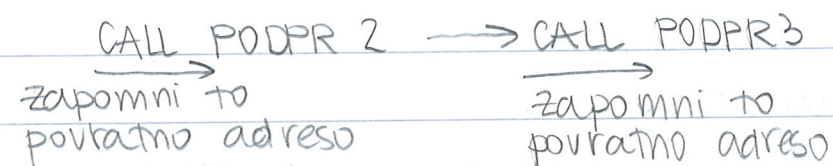
procedure omogočajo skrivanje informacije, vendar te
 procedure samo deloma rešijo skrivanje informacije

PRIMER SKLADA:



SKLAD: uporaben v klicanju podprogramov: moramo shraniti
 tudi povratno adresno

PODPR 1



SKLAD: idealna struktura za pomnjenje povratnih adres

- pri prekinitvenih sistemih
- pri prevajalnikih

REALIZACIJA SKLADA

Osnovni operaciji: -push: potisnemo podatek v sklad
 -pop: vzamemo iz sklada
 init: inicializacija
 top: vrh



int datastack [100]; // rezerviramo 100 lokacij

int datatop = 0;

void init ()
 {

datatop = 0;
 }

void push (int val) // vstavi podatek z vrednostjo value
 {

if (datatop < 100) // preveri če je prosto mesto
 datastack[datatop++] = val; // vstavi v top, potem
 } // premakne kazalec

int top ()

{

if (datatop > 0)

return datastack[datatop-1]; // ne smemo premakniti
 return 0; // kazalec; samo gledamo
 } // kje je kazalec

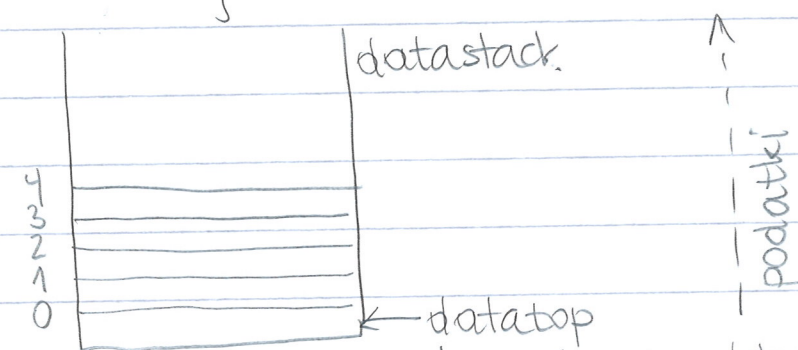
int pop ()

{

if (datatop > 0)

return datastack[--datatop];
 return 0;

}



datatop
 → kazalec na lokacijo kamor
 dajemo podatke

Ali smo dosegli skrivanje informacij

↳ Sklad si deli večje število procedur. Vse te procedure si delijo te operacije (push, pop)

Ali lahko uporabniki to storijo brez da bi vedeli za datatop & datastack. Ali so to lokalne ali globalne spremenljivke?

↳ Te podatke je treba modulirati v vseh moduli. Problem je, če programerji morajo te podatke vedeti (adrese?)

PROBLEM: večje število skladov

⇒ Imamo na nek način skrivanja informacij, samo preko teh deklaracij ⇒ datatop, datastack: ni popolna rešitev

MODUL: če imamo modul, potem prostor teh imen razdelimo na dve skupini:

A) JAVNI DEL IMEN: dostopen vsakemu push, pop

B) PRIVATNI DEL IMEN: dostopen samo znotraj modula

Načela skrivanja informacije

Uporabniku damo na voljo vsako informacijo, ki jo potrebuje in nič več

Problem z moduli: iz sklada pogostokrat imamo operacije na nekih podatkih

↳ ni nujno da so vse procedure na enem mestu, lahko na več mestih

naslednjih: abstraktni podatkovni tipi

ABSTRAKтни PODATKOVNI TIPI (ADT)

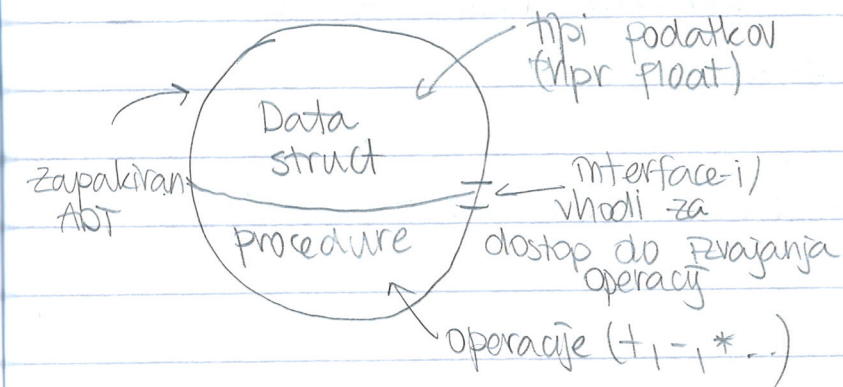
- imamo definirane osnovne podatkovne tipe

↳ nad njimi delamo operacije

↳ ne vemo kako je to noter → implementiramo

- lahko definiramo svoj podatkovni tip (iz osnovnih tipov)

↳ zapakiramo v _____, ki ga uporabimo enako kot osnovne



Znotraj ADT je lahko mnogo uporabnih nevidnih procedur

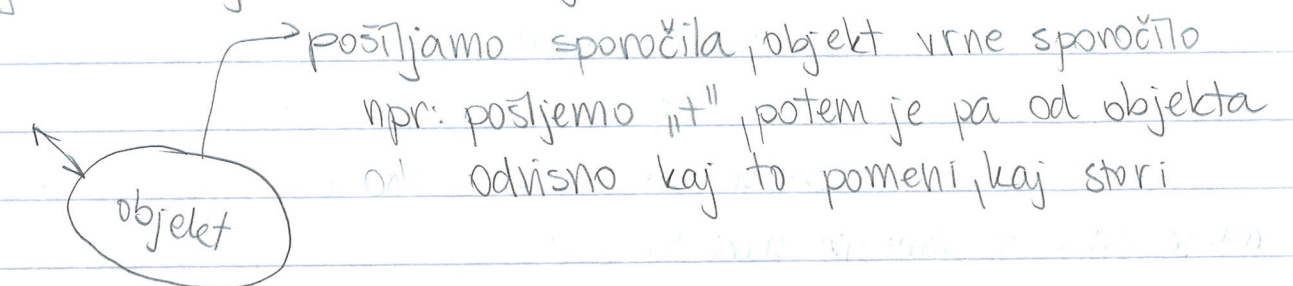
↳ Uporabnik lahko dostopa le do dovoljenih operacij

- smo že korak bližje objektnemu programiranju

- v Smalltalk-u lahko uporabljamo le ADT

- vsaka taka stvar stane (več pomnilnika), vendar se to na dolgi rok splača

- tu se metode še vedno kličejo kot procedure → pri objektnem je to drugače → na drugi strani



DELOVANJE (IN THE SITUATION)

- značilno za objektno programiranje

- nov objekt lahko podeduje lastnosti nekega drugega objekta

- lahko dodamo tudi nove lastnosti

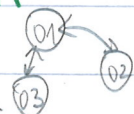
- da se narediti celo hierarhijo objektov

↳ npr: obek kompleksno ⇒ vektor kompleksnih

- Polmorfizen (raznolikost oblik): da se spremeniti dele, ki ne ustrezajo

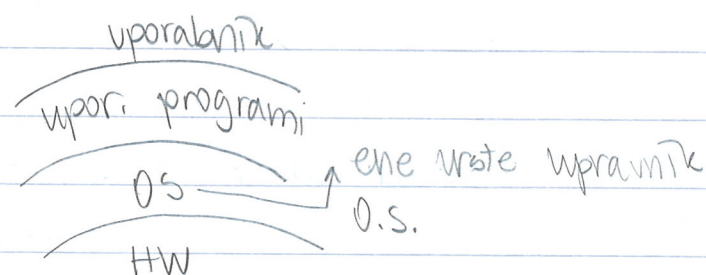
POVZETEK LASTNOSTI OBJEKTNEGA PROGRAMIRANJA

- 1) Vsaka stvar, ki jo obravnavate je objekt
- 2) Vse obdelave izvajajo objekti, ki komunicirajo s spornosti
- 3) Vsak objekt ima svoj lastni pomnilnik (notri so lahko tudi drugi objekti)
- 4) Vsak posamezen objekt je samo pomnilnik določenega razreda
- 5) Razred je shramba obnašanja metod objekta (cel razred se obnaša enako)
- 6) Objekte lahko organiziramo v neko strukturo (hierarhijo)



PREGLED RAZVOJA IN VRSTE OS

- OS: sloj podprogramov:



REZIDENČNI MONITOR

NALOGE

- skrbi za udobje uporabnika (npr. dostop do diska)
 - učinkovito upravljanje z viri
 - skrbi za varnost uporabnika in procesorskega sistema
- Zadnje čase so vmesniki med sloji standardizirani, da se lažje kombinira

- Sistemi so se razvijali skupaj s hardware-om
- ↳ učinkoviti in udobni uporabniku

Delo računalniškega sistema odvisno od močnega dela operaterja

↳ to je zelo drago

FAZE V RAZVOJU O.S.

- 1) AVTOMATIZACIJA: izločanje operaterja programi v posebnih paketih
- ⇒ POJAV REZIDENČNEGA MONITORJA

↳ paketna obdelava: nadzoruje avtomatski prehod od enega paketa na drugega

↳ povečanje učinkovitosti

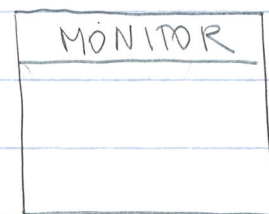
⇒ pojava KONTROLNIH UKAZOV

↳ ukazi monitorju

⇒ 1. PROBLEM: VARNOST

- uporabniški program lahko poveži monitor

- en uporabnik lahko prebere podatke drugega uporabnika



⇒ 2. PROBLEM: V/I ENOTE

- računalnik hitrejši od I/O enot.

↳ pr: VHODNA OPERACIJA: računalnik spoži to operacijo, potem pa čaka da se to vhodno operacijo izvrši

↳ računalnik tudi nadzoruje to operacijo.

KAKO REŠIMO TA PROBLEM?

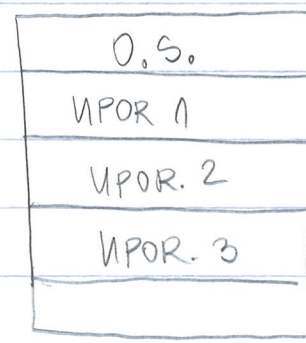
↳ sheme za sočasno izvajanje drugih operacij

↳ magnetni trak (podatke dajejo na mag. trake)

↳ spooler: sočasno se izvaja vnos in branje ali izpis

↳ procesor dela nekaj, hkrati pa se izvajajo V/I operacije

2) MULTIPROGRAMIRANJE: s pojavom večjih diskov in dovolj velikim pomnilnikom

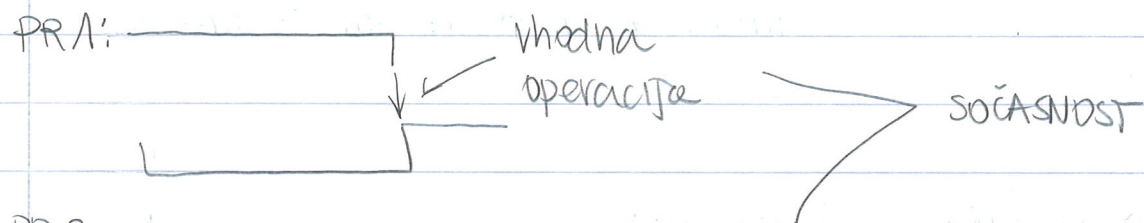


Večje število uporabnikov v centralnem pomnilniku.

Izvaja se en program enega uporabnika. Ta uporabnik hoče izvesti neko V/I operacijo $\Rightarrow$ zato procesor prenesemo na drug uporabnik in računalnik začne izvajati program drugega uporabnika. Tako naprej za tretjega, ...
Stopnja programiranja: število uporabnikov, velikost pomnilnika O.S. skrbi za vse te preklope

$\Rightarrow$ Uporabnik vpraša OS za V/I operacijo.

V izvajanju je večje število programov, čeprav dejansko se izvaja samo en $\Rightarrow$ večja učinkovitost in boljša izraba



OS postane bolj zapleten $\Rightarrow$ problemi varnosti

Kaj mora OS pri tem delati?

Procesor v dogodkih (zahteva za V/I operacije, ...) preklaplja med uporabniki. $\Rightarrow$ na istem procesorju se izvajajo več operacij

$\Rightarrow$ Skomin za vse podatke (kdo ga uporablja?)

Kdaj smo prekinili podatke?

Na začetku računalništva uporabnik ni imel dostopa do računalnika
Sedaj imamo INTERAKCijo z računalnikom (app)
 $\Rightarrow$ uporabnik rabi direktni dostop do računalnika

3) TIME SHARING: ČASOVNO PREPLETANJE

- dodelitev procesorskega časa procesorja uporabnikom po časovnih režimih



- Vsakemu uporabniku pripada časovna rezina
- Če uporabnik predčasno zaključi, takoj odda OS sistemu.

- Lahko uporabnik izrabi to časovno rezino in ne more nadaljevati, zato ker mu OS vzame to rezino in prekine
- OS to naredi s pomočjo prekinitvenega signala

$\Rightarrow$ Če to teče dovolj hitro, vsi imajo nek zagotovljen čas in uporabniki tega ne opazijo.

Ko je računalnik v izvajanju (kadarkoli = asinhrono) lahko mu vzamemo prostor in ga predamo drugemu uporabniku

$\Rightarrow$ Potem bo treba ta program zagnati iz iste točke, kjer je prej nehal $\Rightarrow$ INTERAKCIJA

4) VZPOREDNO DELUJOČI SISTEMI

Možnost, da imamo na istem vodiku večje število procesorjev in večje število uporabnikov v pomnilniku. Običajno: več uporabnikov kot procesorjev.

5) DISTRIBUIRANI SISTEMI

Medsebojno sklopljeni računalniki samo preko komunikacijskih kanalov, ki nimajo dostopa do istega pomnilnika. Tudi ti distribuirani sistemi potrebujejo OS.

6) DELO V REALNEM ČASU

Procesor se mora odzvati v določenih časovnih mejah (dovolj hitro) na določene signale iz okolja

Dva tipa dela v realnem času:

A) SOFT REAL TIME: v večini primerov se morajo odzvati v določenem času

B) HARD REAL TIME: strogo se morajo držati omejitev:

$\Rightarrow$ strogo v predpisanem času (letalo)

Kako dosežemo soft real time?

Odzivov dajemo višjo prioriteto. OS prekine program z

nižjo prioriteto in da procesu program z višjo prioriteto.
 ⇒ PREKLOP ZARADI VIŠJE PRIORITETE

TIP I OS

A) **PAKETNI OS**: Obdelujejo pakete

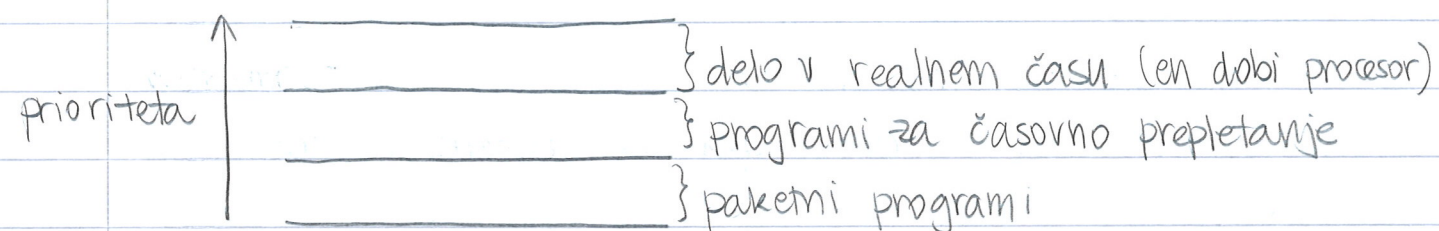
↳ ta način obdelave se še zmeraj uporablja

B) **OS S ČASOVNIM PREPLETANJEM**

C) **OS V REALNEM ČASU**

D) **KOMBINIRANI OS**

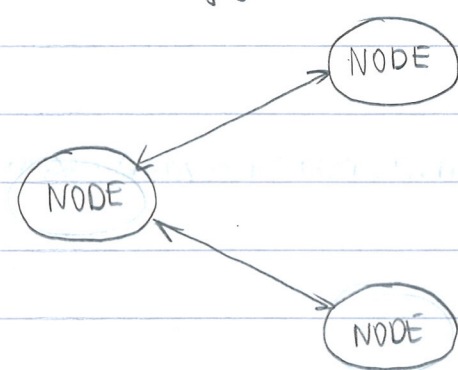
Programi z različnimi prioritetami



Z določenimi parametri se lahko doseže, da je računalniški sistem bolj ugoden za operacijske sisteme, ki jih je več.

E) **DISTRIBUIRANI OS**

OS, ki delujejo z distribuiranimi računalniki

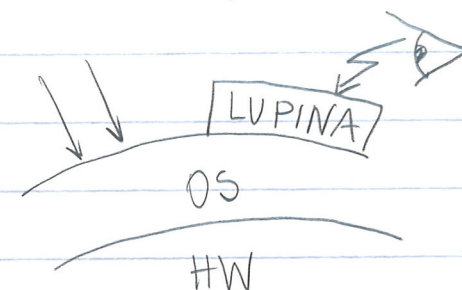


Tudi ti računalniki so avtonomni (vsak zase procesor)

Vsak uporabnik ima svoj OS in komunicira z drugimi procesorji

- Eksplicitna uporaba svojega računalnika in distribuirana uporaba drugih računalnikov.
- Ne vemo kje se kaj odvija
- Ena delujoča enota

KAKO LAHKO GLEDAMO NA OS?



A) **SISTEMSKI KLICI**

↳ z OS lahko komuniciramo s t.i. sistemskimi klici

↳ s pomočjo sistemskih klicov lahko izrabimo vse možnosti OS

↳ standardizirani klici

B) **UPORABA LUPINE**

↳ nadzor preko katerega uporabnik komunicira s procesorjem

↳ Lupina prevede v neko zaporedje sistemskih klicov.

STRUKTURA ZGRADBE OS

Začetek: monolitni sistemi

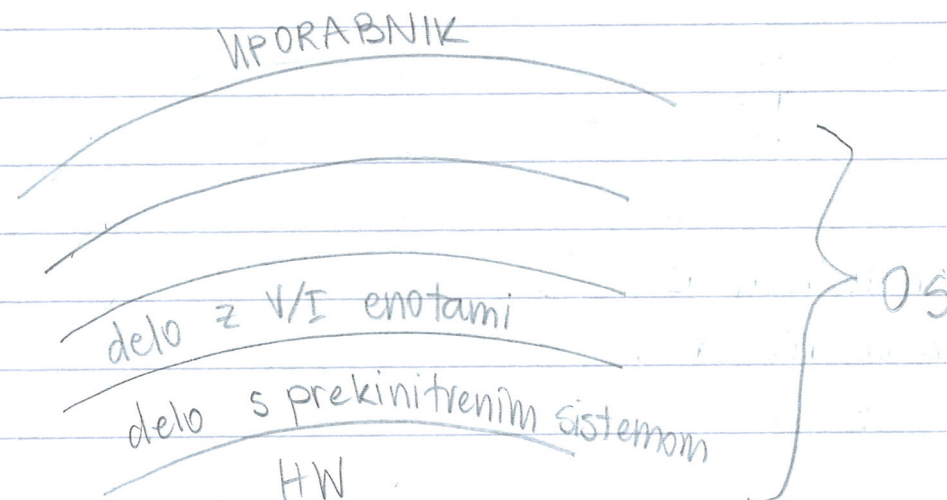
↳ ni nobene arhitekture, ni strukture

↳ programi, ki delajo pravilno

Metoda za strukturiranje:

LUPINASTA STRUKTURA OS.

- sestavljena iz večjega števila virtualnih (navideznih) strojev



Prva lupina: = jedro O.S.

Problem: to strukturiranje je težko narediti

- vse lupine delajo v privilegiranem načinu dela (ne da se ga prekiniti; uporabnik ne sme posegati notri)
- pogostokrat ne potrebujemo vseh teh lupin.
- ⇒ Privilegiran način ni dober.

⇒ zato se uporablja tako, da je jedro (MIKROJEDRO) je v privilegiranem načinu in se deklarira kot O.S., vse ostalo pa teče v uporabniškem načinu

OBJEKTHO STRUKTURIRANJE

Windows 7

SOČASNI / IZPOREDNI ALGORITMI

- delujejo istočasno ali se prepletajo (navidezno sočasno)
 - ↳ preklap med programi

Težek nadzor, pisanje programe

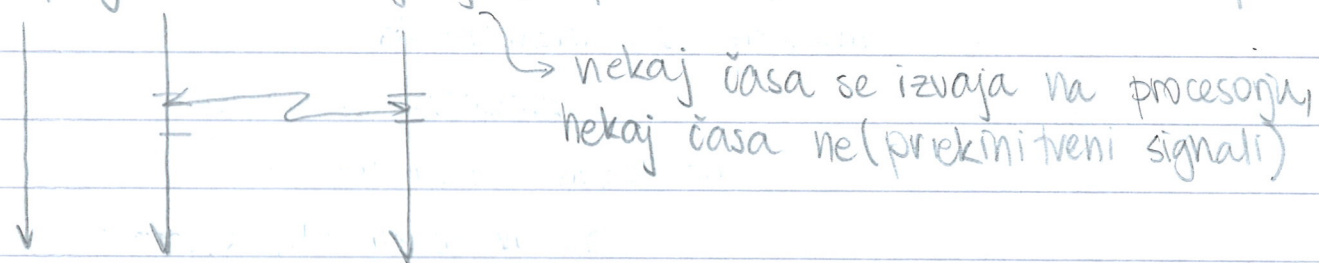
če niso neodvisne: sodelujejo ⇒ to je zelo težko nadzorovati

ABSTRAKCIJA: = za poenostavitev

↳ LOGIČNA: ne zanima nas na kakšen način se izvaja

PROCES = ena osnovnih abstrakcij (TASK)

↳ program v izvajanju ⇒ tipično v procesu: SEKVENČNI PROGRAM



Vsi programi^{so} sestavljeni iz procesov, ampak na določenih mestih bodo ti programi prišli v interakcijo. Ne sme biti preveč interakcijskih točk.

! Izvedba procesov je nekaj drugega kot zgradba procesov (logična) !

IZPOREDNOST

Proces: zaporedna (sekvenčna) aktivnost, ki se izvaja takrat ko jo obravnavamo; zato rečemo da je program v izvajanju procesi imajo interakcije med seboj: pride do sodelovanja procesov (izmenjava viru sporočila) ali tekmovanja (za vir, za sporočila)

Če procesi nimajo medsebojne interakcije ne more priti do težav oz. motenj

Problem: interakcija

↳ Stiki med procesi = SODELOVANJE: 2 načina

A) komunikacija preko skupnega pomnilnika

[RAM]: En proces nalozí podatke v RAM in drug prebere

B) pošiljanje sporočil: pošiljanje podatke prek sporočila (tudi na en procesor)

TEKMOVANJE (za vire): vir: tiskalnik, mag. trakovi, sporočila

Komunikacije: konflikt: ti viri morajo biti opisani tako, da ne pride do konfliktov oz. da se te konflikti razrešijo

INTERAKCIJA (kadar imamo opravke z viri)

vir: A) HW: tiskalnik, disk B) SW: prevajalnik, sporočilo

↳ vsaka stvar na katero je treba čakati

↳ vir opisan v RAM-u in nekdo (OS) mora to urejati

NESODELJUJOČI PROCESI: nepomembni različni časovni odnosi

INTERAKCIJA: ni vseeno kakšni so medsebojni časovni odnosi

PROBLEM SINHRONIZACIJE: upoštevanje časovnih pravit

PRIMER: P1, P2: nek dogodek v P1 se mora zgoditi preden se zgodi dogodek v P2

PRIMER: P1... Pn...: Pn1 sme začeti z delom, ko so vsi ti vključno s Pn zaključeni

PROBLEM: Določenih operacij več procesov ne bo smelo izvajati

↳ 'Istočasno' = MEDSEBOJNO IZKLJUČEVANJE

PRIMER: tiskalnik: Dva procesa ali več ne smejo uporabljati tiskalnik istočasno.

ARHITEKTURA SISTEMOV

- različna

1) Enoprocessorski sistem

MULTIPROGRAMSKO OKOLJE:

↳ več procesov v izvajanju na enem procesorju (dejansko se izvaja samo en proces)

- tak procesor je treba preklapljati

- preklap: kadar se zgodi nek dogodek.

↳ dogodek: odvisno kaj je določeno (stvar načrtovalca)

- na istem procesorju se izvaja tudi O.S. in to s pomočjo mehanizmov, ki O.S. dodelijo procesor

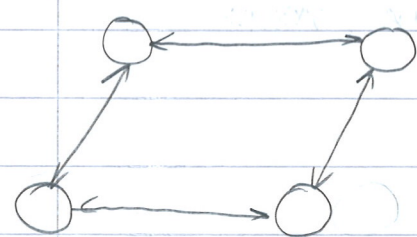
2) Večprocessorski sistem

↳ več krepko sklopljenih procesorjev, ki imajo dostop do skupnega pomnilnika preko skupnega vodila

PROBLEM: Kako vse te stvari upravljati? O.S. skrbi za to.

↳ imamo več procesorjev na voljo. Procese razdelimo procesorjem.

3) Arhitektura distribuiranih sistemov: posamezna vozlišča na voljo, ki so med seboj povezani s komunikacijskimi povezavami.



- nimajo skupnega pomnilnika, ki bi ga lahko hitro zasledili.

- zaporedno povezan komunikacijski procesor

DISTRIBUIRAN SISTEM Z ENIM O.S.

Porazdeljen preko strukture: V nobenem trenutku nima popolne

informacije v kakšnem stanju je sistem, ki ga obravnavamo.

PORAZDELJEN O.S.: V Vsakem vozlišču posebej imamo O.S.

Problem časa; izbiranje imen (ime mora biti globalno)

KOMPONENTE O.S.

V/I SISTEM PROCESORJA: veliko različnih V/I enot. Obravnava

vseh V/I enot na enoviti način. Kako?

Razlika: stavki za pisanje & stavki za branje v program. jeziku

1) TE V/I SO ASINHRONE ?

↳ dogodki, ki se dogajajo na teh V/I enotih niso sinhronizirani s processorsko uro.

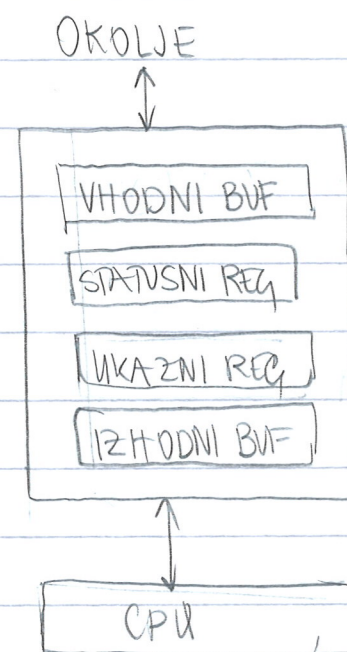
↳ dogodkov v zvezi z V/I enotami ne moremo vnaprej napovedati.

2) Obstaja velik hitrostni prepad med hitrostjo CPU in hitrosti V/I enot: CPU hitrejši ?

3) Performiranje podatkov: Različni formati podatkov na V/I enot kot tistih na CPU ?

V/I ENOTE (SPLOŠNO)

Shematski prikaz:



Notranjost V/I enote s stalisca programiranja opišemo z naborom štirih registerskih množic

REGISTRI:

1) Vhodni buffer

↳ lahko večji nabor vhodnih registrov za začasno shranjevanje podatkov.

↳ ko beremo podatke iz okolja (lahko se pojavijo kadarkoli), ni nujno pa da je tudi CPU na voljo takrat. Vhodni podatki se najprej vpišejo v vhodni izravnalnik, dokler jih CPU ne prebere.

4) Izhodni izravnalnik: shranjevanje za izpis podatkov, ki jih izhodna enota izpiše

2) Statusni registri

- V/I enota mora obveščati okolico. To naredi tako, da vse ugotovitve napiše v statusni register. Zapisan je tudi status V/I enote, da lahko uporabnik prebere.

BUZY BIT: poseben signal preko katerega V/I enota sporoča da je opravila z delom.

- ko je zasedena se pokaže ta bit.

↳ CPU ga lahko posreduje in ugotovi ali je operacija še v teku, ali pa zaključena.

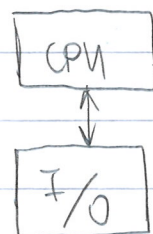
- V/I enota spremembo statusnih registrov lahko sporoči preko **PREKINITVENIH SIGNALOV**

3) Ukazni registri

↳ V/I enoto ob priklopu sistema moramo konfigurirati in konfiguracijske podatke vpišemo v ukazni register

- vpisujemo tudi ukaze

KOMUNIKACIJA MED V/I ENOTO & CPU



Dve osnovni metodi:

1) CPU preverja ali je V/I enota zasedena preko statusnega registra (BUZY BIT)

- če V/I enota ni zasedena, potem CPU vpiše v ukazni register: preberi podatek

- CPU je med tem prosta:

⇒ Najpreprostejša shema: CPU otipava BUZY BIT dokler ne ugotovi, da je V/I enota zaključila procesorsko kontroliranega prenosa

2) Po sproženi vhodni operaciji CPU dela vmes nekaj drugega in V/I enota pošlje prekinitveni signal = METODA S POMOČJO PREKINITVENEGA SIGNALA

↳ s prekinitvenim signalom obvesti CPU da je V/I enota zaključila z delom

PROBLEM: prekinitveni signal lahko pride kadarkoli

če imamo kodno operacijo, potem CPU sproži to operacijo, I/O enota izvede to operacijo in podatek, ki ga spremlja, ga postavi v izravnalnik ali če imamo DMA se direktno prenese v pomnilnik
⇒ V/I enota mora prenesti podatek v pomnilnik

Dve možnosti o obveščanju:

1) z otipavanjem

2) s prekinitvenim signalom

Potek prenosa:

Preberemo podatek, ga obdelamo (CPU), potem sproži branje drugega podatka. Vendar je to trošenje časa.

Ko V/I enota bere, CPU nima kaj početi in obratno.

DODATNI PROBLEMI: Vhodni podatki prihajajo v izbruhih in CPU nimajo dovolj časa za obdelavo.

REŠITEV:

Najprej preberemo vhodni podatek, takoj sprožimo branje naslednjega podatka; v tem času CPU obdeluje prvi podatek. Imamo dva izravnalnika za to



IDEALNO: Te dve operaciji trajata približno enako. V izravnalniku je vedno podatek.



HEMA DVOJNIH IZRAVNALNIKOV

PROBLEM: prenos podatkov je lahko dolgotrajen in samo zamenjamo kazalca na izravnalniku
- če imamo DMA

V idealnem primeru sta obe enoti vzporedno popolnoma aktivni

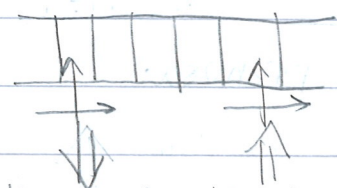
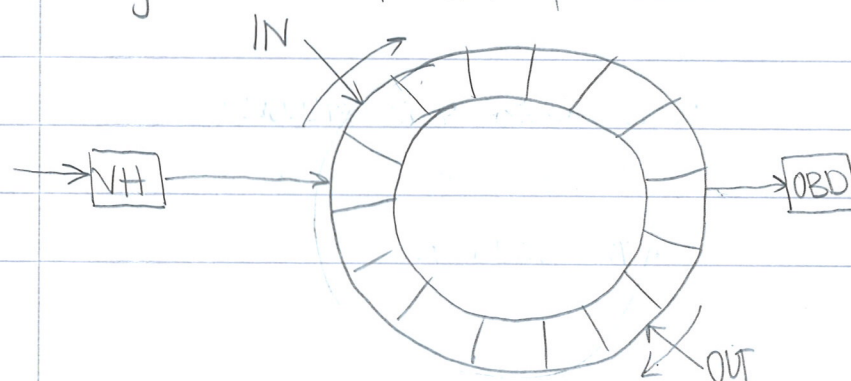
PROBLEM: - branje hitrejše

↳ čaka dokler se obdelava ne zaključi in obratna
- podatki prihajajo v izbruhih
↳ v tem primeru se poslužujemo s **HEMA KROŽNIH IZRAVNALNIKOV**

HEMA KROŽNIH IZRAVNALNIKOV

PRIMER ZA VHOD PODATKOV:

Na vходу je krožni izravnalnik. Vsi podatki so enako dolgi - enak format podatkov



na eni strani dodajamo podatke, na drugi strani jih jemljemo

Vhodni program, ki bere iz vhodne enote in jih vstavlja v čakalni vrsti. Na drugi strani program, ki to obdeluje. Če imamo dva procesorja, en skrbi za vnos, vhodno enoto, daje podatke v krožni izravnalnik. Drug procesor jemlje podatke iz izravnalnika in jih obdeluje. Različni hitrosti za polnjenje in obdelavo. Če imamo en procesor, bo ta procesor moral skrbeti za preskok iz ene aktivnosti na drugo oz. iz enega procesorja na drugega. Kako to napravimo? → Dve možnosti:

1) metoda z otipavanjem bita zasedenosti → nerodno

2) z uporabo prekinitvenega signala → ne dela

Tri vrste izravnalnikov

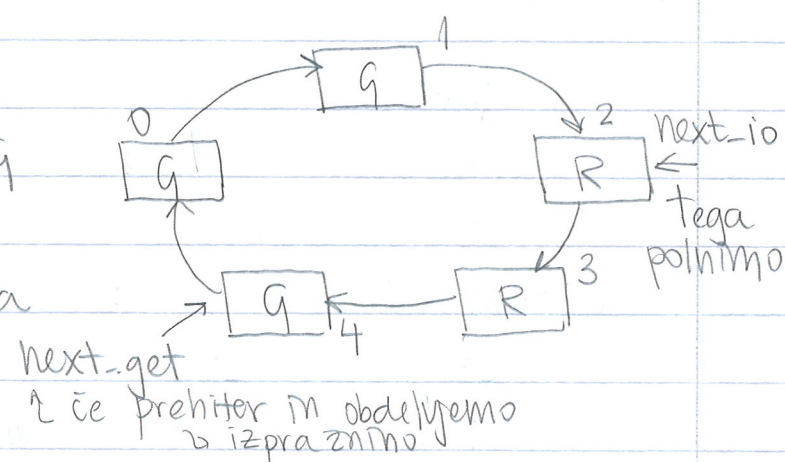
A) prazni: polnimo s podatki R

B) polni: obdelujemo podatke G

C) tega, ki obdelujemo

Get_buf: jemljemo podatke iz njega

Release_buf



buf[0], buf[1], ..., buf[n-1] // da bo kazalec krožil

repeat

compute - only;

Get_buf (current);

Compute (buf[current]);

Release_buf;

until all;

V tem času teče tudi vnos podatkov. Kako to naredimo?

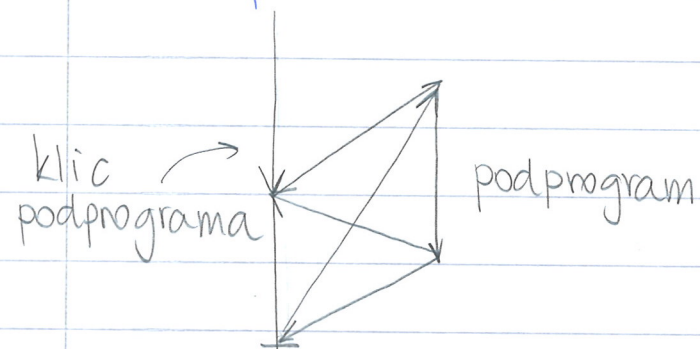
1) otipavanje: ko ta program PROSTOVOLJNO skoči na drugo aktivnost, ki bo delala z vhodnimi podatki.

PROSTOVOLJNO: struktura korutine: na tistem mestu kjer je preklc točno vemo kakšno je stanje.

2) **prekinitveni sistem**: tudi pride do takih preskokov, samo mehanizem je drugačen

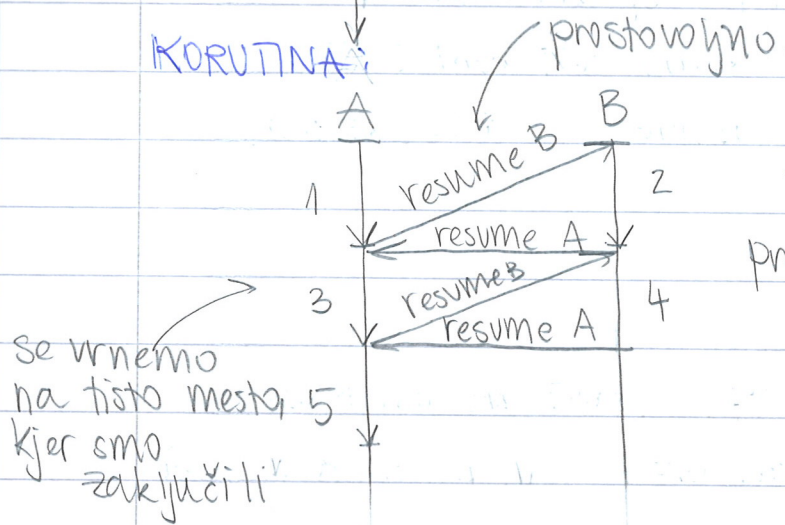
PRISILNI PREKIN: Aktivnosti z višjo prioriteto
 ↳ ne vemo kje pride do preskoka

PODPROGRAMI



začnemo izvajati
v isti točki

KORUTINA



Dva popolnoma enakovredna programa; se vrnemo na točko kjer smo prej zaključili

HEMA Z OTIPAVANJEM BITA ZASEDENOSTI

procedure: isto kot funkcija v C-ju

n1: sestevanje po modulu 1

procedure Get_buf[current];

n.. število vseh izvajalnikov
 r.. število praznih izvajalnikov

begin

repeat

if !busy[ch] then resume IOP

until r=n // če so vsi prazni nima kaj za obdelovati

current:=next-get; // kaže na sedanjega

next-get:=next-get+n1 // kaže na tistega ki je naslednji po n

end

procedure Release_buf;

begin

r:=r+1;

if !busy[ch] then resume IOP

end

IOP: loop // korutina za V/I kanale

while r=0 do resume CP; // če ni praznega buf, ne sme sprožiti nove obdelave

Read(ch, buf[next-io]); // če je izvajalnik prazen, sproži V/I opet ⇒ read

resume CP // vrne kontrolo na CP (obdelavi)

next-io:=next-io+n1

r:=r-1;

end

if !busy[ch] then resume IOP; // KAZ ZA TESTIRANJE BUSY BITA
 INIT: r=n; next-get:=next-io:=0 // na začetku vsi prazni in next-get in next-io na istem mestu

→ Po CPU posejemo. Kje postavimo ta ukaz?

Get_buf: spleča se ⇒ zato posejemo po CP

⇒ TO JE SLABOST: testiranje bita je treba postaviti na večih mestih

Imamo DVE KORUTINI: 1) tisti program prej = KORUTINA CP

2) korutina, ki skrbi za V/I kanal = KORUTINA IOP

Predpostavljamo da sta dva procesa

HEMA S PREKINITVENIM SIGNALOM

procedure Get_buf(current); // jemljemo naslednji polni izravnalnik

begin

while r=n do ; // nič ne dela če so vsi zasedeni, čaka

current := next_get; // javi da je poln

next_get := next_get + n1; // premakne za eno naprej

end

procedure Release_buf; // vrne polni obdelan izravn. kot prazni

② $r := r + 1$; // poveča št. praznih izravnalnikov

if !busy[ch] then read(ch, buf[next_io]); // to se zgodi ko se sproži

end

IO: Save State; // to se sproži ko se INIT zaključi

next_io := next_io + n1; // premakne kazalec

① $r := r - 1$; // zmanjša št. praznih izravnalnikov

* if $r \neq 0$ then Read(ch, buf[next_io]); // če je kakšen kazalec prazen sproži novo branje

Restore State; // sproži naslednjo branje

Init: $r = n$;

next_io := 0;

next_get := 0;

Read(ch, buf[next_io]) // tukaj sprožimo vhodno operacijo: RAZLIKA

Vsi ukazi in spremenljivke isti kot prej.

Prekinitveni sistem uporablja isti procesor kot obdelava v procesorju

PROBLEM: Imamo spremenljivke ① & ②, ki spreminjata isto lokacijo

$r := r + 1$: običajno jo prevajalnik mora prevesti v strojni ukaz

LDAA r preberemo iz pomnilnika v akumulator
 ADDA #1 prištejemo 1
 STAA r zapišemo v pomnilnik

PRIMER: Če izvede Release_buf mora biti 6, če izvede IO mora biti spet 5.

Acc (akumulator) 5

Se izvede prekinitveni IO

$r = 5$

$r = 4$

Acc(+1) 6

Zaradi prepleta piše, da je 6 praznih vrstic, ampak jih je samo 5

- to se zgodi samo, če pride do prekinitve med LDAA & ADDA in ADDA & STAA

Kritično: kadar imamo dve ali več vzporednih aktivnosti

Kako to preprečimo?

Dosežem da se ① izvaja kot enota in ne sme priti do prekinitve

↳ Izključimo prekinitveni sistem pred LDAA in ga vklopimo po STAA

↳ To zelo težko (ni dovoljeno, lahko samo OS, ...)

⇒ $r := r + 1$ & $r := r - 1$: ne smeta se zgoditi istočasno, ne smeta se prepletati

⇒ Določeni deli programa se ne smejo zgoditi istočasno

↳ KRITIČNI ODSEKI

↳ eden tipičnih problemov sinhronizacije

Kako to dosežemo?

OSNOVNI MODEL PROCESA

Proces: program v izvajanju

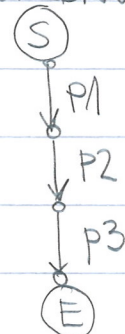
Kako so procesi med seboj strukturirani in kako napišemo v računalnik? Kako se bo to izvajalo?

↳ nobene interakcije

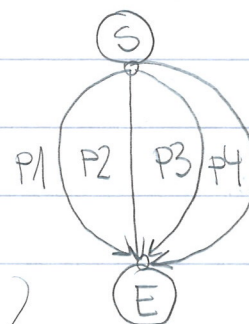
Izvajalni graf procesa

proces

A) ZAPOREDNO IZVAJANJE



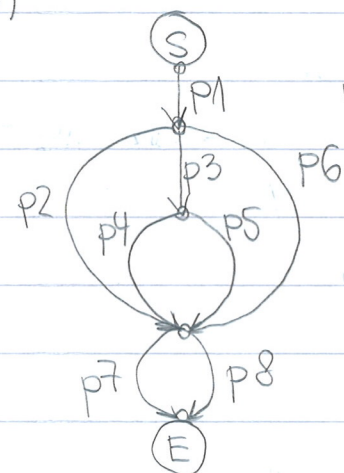
B) IZPOREDNO IZVAJANJE



Ko se vsi 4
zaključijo
pridemo do
točke End

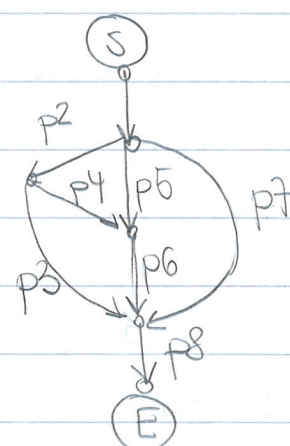
izvajajo se sočasno

C)



Opiši s
konstrukcijo

D)



Kako to opišemo? KONSTRUKTA:

$S(p1, p2)$... serial: zaporedno izvajanje ukazov

$P(p1, p2)$... parallel: vzporedno izvajanje ukazov

↳ izvajalni grafi, ki se dajo opisati s tema dvema konstrukcijo:

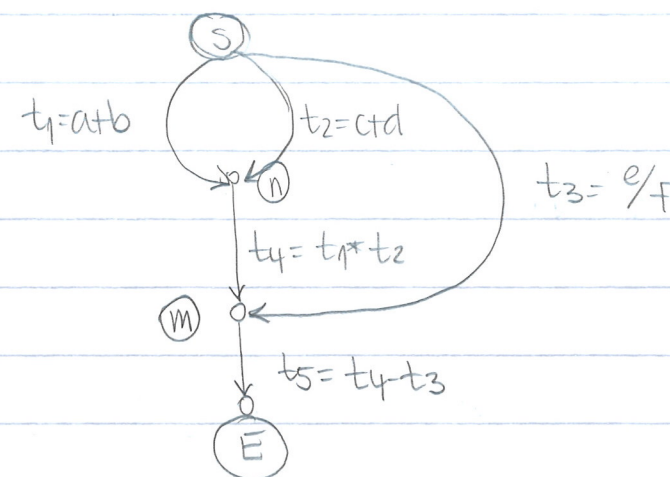
↳ VGNJEZDENI IZVAJALNI GRAFI

$S(p1, S(p2, p3))$

Nekateri programski jeziki imajo konstrukte za opis serijskega in paralelnega izvajanja ukazov

↳ ne pa za primer D)

$$(a+b) * (c+d) - (e/f)$$



IZPOREDNOST: cobegin p1/p2/p3.../pn coend

↳ vzporedno izvajanje procesov p1, p2, ..., pn

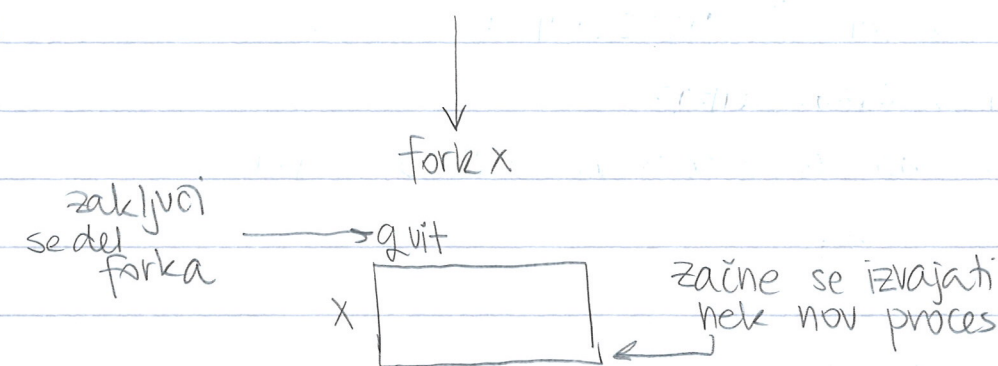
```

cobegin
  [ cobegin t1:=a+b | t2:=c+d coend;
    t4:=t1*t2;
    t3:=e/f;
  ]
coend
t5:=t4-t3;
  
```

IZPOREDNOST

~~NEVZNEZDENI IZVAJALNI GRAFI~~ LINUX fork, join, quit

fork x



join t1, y

NEDELIVAN ENOTA

$t_i := t - 1$

if $t = 0$ then go to y

Vsak proces ima to M s tem se zmanjšuje n. če izvajamo n procesov, ki se morajo zaključiti v neki točki, neko spremenljivko t

ta ukaz vedno
na koncu procesa

postavimo na vrednost n . Ko proces zaključi, zmanjša n (en proces se je zaključil), štejemo koliko procesov se je zaključilo. Ko se vsi zaključijo gremo na y .

glej ☉

n, m .. štejeta koliko je tistih procesov, ki se niso zaključili

$n := 2;$

$m := 2;$

$\left\{ \begin{array}{l} \text{fork } p_3; // \text{požene } p_3 \\ \text{fork } p_2; // \text{požene } p_2 \end{array} \right.$
 $t_1 := a + b; \text{ join } n, p_4; \text{ quit}; // \text{hkrati s } p_2 \text{ \& } p_3 \text{ računamo } p_1$

→ $p_2: t_2 := c + d; \text{ join } n, p_4; \text{ quit}; // \text{join: zmanjša } n \text{ za } 1$

⇒ $p_4: t_4 := t_1 * t_2; \text{ join } m, p_5; \text{ quit};$

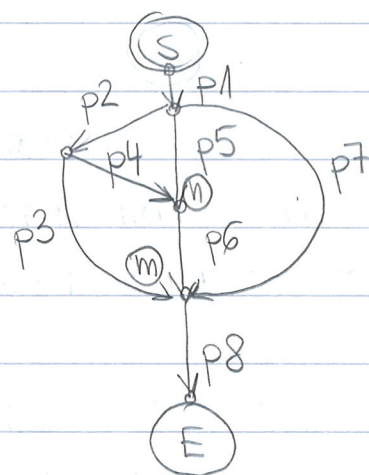
→ $p_3: t_3 := e / f; \text{ join } m, p; \text{ quit};$

⇒ $p_5: t_5 := t_4 - t_3$

Kadar se bodo izvajali več ukazov, se ti ukazi izvedejo zaporedno v nekem n . Vsak ukaz zmanjša n za 1. Kadar $n=0$ skočimo na p_4 . ⇒ **NEDELJIVI PROCESI**

NEKONEČNI GRAF

↳ ne moremo ga opisati cobegin & coend



S_i ..koda procesa

P_i ..labela procesa kjer se proces začne

$n := 2;$

$m := 3;$

$S_1; \text{fork } p_2; \text{fork } p_5; \text{fork } p_7; \text{quit};$

$p_2; S_2; \text{fork } p_4; \text{fork } p_3; \text{quit};$ // izvede se koda S_2 startam p_3 & p_4

$p_5; S_5; \text{join } n; p_6; \text{quit};$

$p_7; S_7; \text{join } m; p_8; \text{quit};$

$p_3; S_3; \text{join } m; p_8; \text{quit};$

$p_4; S_4; \text{join } n; p_6; \text{quit};$

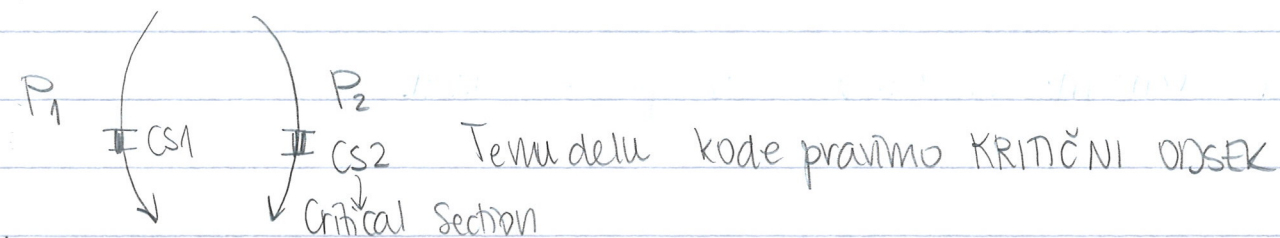
$p_6; S_6; \text{join } m; p_8; \text{quit};$

$p_8; S_8; \text{quit};$

Procesi med katerimi obstaja interakcija (sodelovanje ali tekmovanje)

↳ V takem primeru lahko nastane problem če vzporedni sočasno izvajajoči se procesi imajo dostop do istega procesorja (lokacije, ki jo dva procesa lahko spreminjata)

interakcija = obstajajo deli kode, ki jih procesi ne smejo izvajati istočasno (Bodisi ko imamo dejansko vzporednost ali prepletanje (navidezna vzporednost))



Moramo doseči, da se ti CS izvajajo kot nedeljive enote, lahko je tudi cel proces kritični odsek ⇒ vsilimo zaporednost
 Če so CS dolgi ⇒ Problem za vzporednost ⇒ Predpostavljamo, da so CS kratki deli kode.

Kako dosežemo, da se v nekem trenutku ne bosta izvajala sočasno?

1) **IZKLOP PREKINITIVNEGA SISTEMA** (pri enoprocesorskem sistemu)

↳ slaba rešitev

pri večprocesorskih sistemih tudi to ne pomaga.

2) lahko se zgodi, da noben ne more vstopiti v CS

↳ dva procesa čakata, pa noben ne more = DEADLOCK

3) Če imamo več CS in v nekem hipu vsi procesi želijo vstopiti v svoj CS → dobra rešitev: en vstopi, ostali čakajo (kateri vstopi = rešitev brez prioritete)

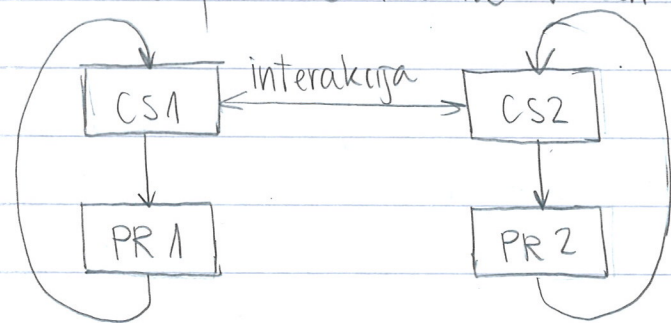
ZAHTEVE →

A) DEADLOCK - problem zapore

B) Proces se lahko ustavi izven CS ⇒ to ne sme vplivati na druge procese

C) Za CS predpostavimo, da se ne bo nikoli ustavil

ZGLED: oba procesa tečeta v zankah



Ne sme priti v sočasnost CS in tudi do zapore ne sme priti.

REŠITEV 1:

var turn: integer; // spremlj. turn
turn := 2;

cobegin

[p1: loop
while turn = 2 do;
CS1; turn := 2; PR1
end

| // znak med dvema procesoma

[p2: loop // najpej ta vstopi

PR1, PR2: dva dela procesa med katerima ni interakcije

while turn = 1 do;

CS2; turn := 1; PR2

coend

⇒ Obe ne moreta vstopiti istočasno

P1, P2, P1, P2

TA REŠITEV NI DOBRA: procesa lahko izmenično vstopata, vendar če je en hitrejši bo moral čakati na počasnega; Če se eden od procesov ustavi izven CS ⇒ tudi drugi prenos se ustavi

REŠITEV 2: Vsak prenos ima svojo spremenljivko s katero javlja da se drugi proces ne spreminja; drugi prenos samo gleda

var C1, C2; boolean; // Booleovi spr. true ali false

C1 := C2 := true; // ni v CS; obe true

cobegin

p1: loop

C1 := false // vsak proces preden vstopi v CS postavi svojo spremenljivko na false & pogleda drugega če je v CS
while !C2 do;
↳ če je drugi v CS čaka

CS1; C1 := true; PR1;

end

|

loop

C2 := false

while !C1 do;

CS2; C2 := true; PR2

end

coend

Če oba sočasno hočeta vstopiti v CS, oba bosta postavila C1 & C2 na false ⇒ oba čakata in noben ne bo vstopil

↳ ZAPORA

⇒ REŠITEV: najprej pogleda če je drugi v CS in potem postavi
 ↳ oba ugotovita, da drugi ni v CS ⇒ oba vstopita v CS
 ↳ SOČASNI DOSTOP

CELOTNA REŠITEV: Kombinacija teh dveh rešitev.
 ↳ Tukaj je problem simetrija

REŠITEV 3: Petersonova rešitev (asimetrija)

var C1, C2: boolean; turn: integer;

C1 := C2 := true;

cobegin

P1: loop

C1 := false;

turn := 1

while ¬C2 ∧ turn = 1 do;

CS1; C1 := true; PR1

end

P2: loop

C2 := false;

turn := 2

while ¬C1 ∧ turn = 2 do;

CS2; C2 := true; PR2

end

coend

Tadva procesa tekmujeta kdo bo vstopil v CS

SEMAFORJI

- kadar imajo procesi dostop do skupnega računalnika-pomnilnika
- če je več jedrov, procesorjev, morajo biti krepko sklopljeni
- dosti v praksi

SEMAFOR: nenegativna celostevilna spremenljivka S.

S, V(s), P(s) S ≥ 0
 operaciji

binarni semafor: S=1 ali S=0

splošni semafor: S=ostalo

V(s): operacija, ki se mora izvajati kot NEDELJIVA ENOTA,
 ↳ poveča S za 1

S := S + 1

P(s): tudi kot NEDELJIVA ENOTA

↳ če S > 0: se zmanjša za 1

↳ če S = 0: proces mora počakati dokler ni S > 0.

↳ if S > 0 then S := S - 1

else wait

⇒ REŠITEV MEDSEBOJNEGA IZKLJUČEVANJA (za n-procesov)

var mutex: semaphore

mutex := 1;

cobegin loop .. end

↓
 kritični odsek med operacijama
 P8 V

P1 loop

P(mutex); CSi; V(mutex); PRi

end

↳ ko proces izstopi iz CS postavi
 mutex na 1

Pn loop .. end // ko je mutex 0 počaka
 coend

BINARNI SEMAFOR

var S: semaphore

S := 0

cobegin

P1: begin

:

P(S)

:

end

P2: begin

:

V(S)

:

end

coend

Proces P1 ne sme nadaljevati v tej točki, če proces P2 predhodno ni dosegel te točke

Ko se P izvede, poveča S na 1

Semafors lahko uporabljamo tudi za štetje

bnpr: koliko praznih izravnalnikov je še na voljo. Ko je to na 0 mora počakati

var e, f, b: semaphore

e := n; f := 0; b := 1

// e.. empty - n praznih

cobegin

// f.. full - nič polnih

producer: loop

Produce_next;

P(e); P(b); Add_to_buffer; V(b); V(f)

end

poveča f za 1,
⇒ en poln vec

consumer: loop

P(f); P(b); Take_from_buffer; V(b); V(e)

Process_next;

end

coend

vrne izravnalnik med prazne
↳ struktura kritičnega odseka
↳ s P(b) & V(b) to zaščitimo

IMPLEMENTACIJA SEMAFORJEV

Če imamo enoprocesorski sistem s časovnim prepletanjem

↳ izklopimo prekinitve (dosežemo nedeljivost izvajanja). s tem izklopimo tudi najbolj prioritete sisteme.

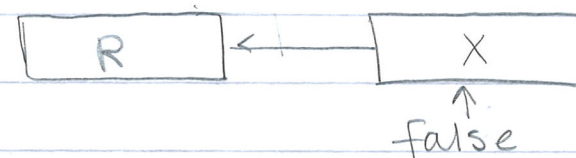
Tega ne moremo narediti pri večprocesorskih sistemih, ki so povezani preko skupnega pomnilnika. Tu obstaja ukaz test-and-set, ki se izvaja kot nedeljiva enota na vodilu

Ukaz Test and set (TS(x)) - se zgodi v resnici Test and clear

R := X

x := false } nedeljivo

OPERACIJA



vidimo, kakšen je bil prej x
x (true ali false)

Binarni:

Semaforsji z delovnim čakanjem (busy wait)

Pb(Sb) .. while TS(Sb) do;

Vb(Sb) .. Sb := true

↳ ko je TS true in Sb false
se vrti v zgornji zanki
in čaka

↳ vrednosti semaforja so 0 ali 1 (Bobleau)

Realizacija funkcije P(s)

P(s): Inhibit interrupts;

Pb(mutex);

S := S - 1; // S je lahko negativen → šteje koliko procesov čaka (-1, -2, ...)

if S < 0 then begin Vb(mutex); Pb(delay) end;

Vb(mutex);

Enable interrupts

tu čaka čakajoči proces

V(s): Inhibit interrupts;

Pb(mutex);

S := S + 1;

if $s \leq 0$ then Vb(delay) else Vb(mutex);

Enable interrupts

INIT:

mutex := true

delay := false

SINTAKSA: "slovnika" pravila

SEMANTIKA: pomen

KOMUNIKACIJA

send(p, msg) // proces, ki izvede send pošlje procesu p sporočilo msg
receive(q, msg) // proces, ki izvede receive pričakuje da bo dobil
od procesa q sporočilo msg

Predpostavka: Sporočilo je prenešeno

1) Če nek proces pošlje sporočilo ali čaka da je sporočilo sprejeto
(BLOKIRAJOČI PROCES) ali ne (NEBLOKIRAJOČI PROCES)

2) Če proces izvede operacijo receive ali je sporočilo na voljo ali ne
če ni ali čaka da bo sporočilo sprejeto = BLOKIRAJOČI SINHRONI SPREJEM

3) Proces lahko pošlje enim ali večim naslovnikom = EKSPlicitNO ali
IMPLICITNO IMENOVANJE

4) Sprejemnik lahko pričakuje sporočilo od določenega procesa ali
od kogorkoli (npr: tiskalnik)

8 Možnosti:

A) Oddajanje z eksplcitnim imenovanjem

1) blokirajoči proces

2) neblokirajoči proces

B) Oddajanje z implicitnim imenovanjem

1) blokirajoči proces

2) neblokirajoči proces

C) } Isto pri sprejemu
D) }

Pr: Blokirajoči z eksplcitnim: Pošlje kataunko enemu sprejemniku in
ta sprejemnik ga pričakuje. Sprejemnik
obvesti oddajnika o sprejemu
→ SINHRONIZACIJA

Kako s temi konstrukti izvedemo vse sinhronizacijske operacije?

PROBLEM

Problem producent - potrošnik (blokirajoča oper. z eksplcitnim imen.)

① Primer prenosa enega podatka

producer:

loop

produce - data;

send(consumer, data); *

end

consumer:

loop

receive(producer, data); **

consume - data;

end

* in ** se morata ujeti: veliko časovnih možnosti

consumer mora počakati da jih producer ^{pošlje} dobi in producer
mora čakati, da jih consumer potrebuje


```

process producer;
loop
  produce_data;
  send (buffer_manager, data);
end

process consumer;
loop
  send (buffer_manager, request); // zahteva: lahko nobenega
  receive (buffer_manager, data); // polnega izvajalnika ni
  consume_data;
end

process buffer_manager; // upravitelj izvajalnikov
loop
  if full_cnt = 0 then // nobenega izvajalnika ni prejel
  begin
    receive (producer, data);
    Buf [nextin] := data // krožni izvajalnik
    nextin := nextin + 1 // šteje po modulu n
    full_cnt := full_cnt + 1; // šteje koliko izvajalnikov je praznih
  end else
    if full_cnt = n then
    begin
      data := Buf [nextout]
      receive (consumer, request);
      send (consumer, data);
      next_out := next_out + 1;
      full_cnt := full_cnt - 1;
    end else

```

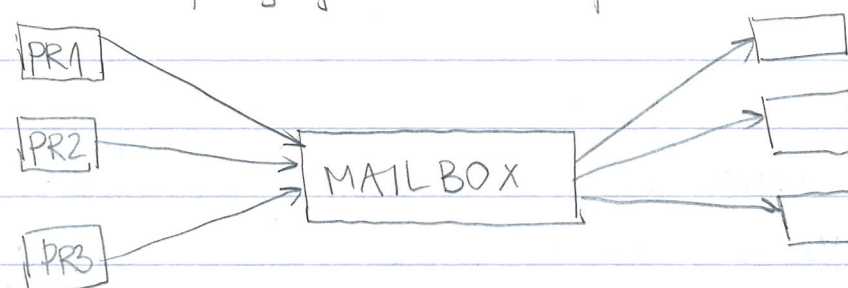
```

begin
  receive (*, temp); // receive z implicitnim imenovanjem
  // *-lahko producent ali potrošnik čaka na
  // katerokoli sporočilo
  if * = producer then // od producenta
  begin
    Buf [nextin] := temp;
    nextin := nextin + 1;
    full_cnt := full_cnt + 1;
  end else --> sporočilo od potrošnika
  begin
    data := Buf [nextout]
    send (consumer, data);
    next_out := next_out + 1;
    full_cnt := full_cnt - 1;
  end
end
end
end

```

Druge možnosti:

② POŠTNI PREDALI (MAIL BOXES): shranjevanje sporočil, v kateri na eni strani pošilja različni procesi

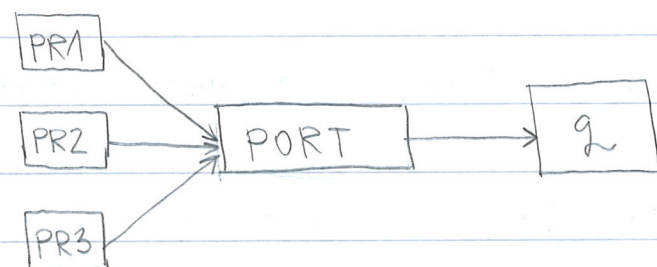


Nastavljanje: FIFO, ... , veliko možnosti

Oddajni procesi na en računalnik, sprejemni pa na drugi
Kje bo mailbox? →

↳ Varianta: vrata (PORT)

↳ sprejema iz večjega števila procesov, oddaja samo enim.



Pazi: OSNOVNI PARAMETRI: blokir, neblok, ekspl., impl.

VIŠJI NIVO

podprogram

RPC MEHANIZEM Remote Procedure Call

↳ Klicanje oddaljenih podprogramov: Pišemo nek program in ta program kliče drug podprogram (lahko na istem ali drugem procesorjem). Če je na istem procesorju → vemo kako to napravimo. Kaj pa če je na drugem procesorju?

REŠITEV: RPC: Vzvezi z vsakim takim programom se ustvari proces. ↳ Ko pokličemo podprogram, potem naš procesor pošlje sporočilo tistemu, ki ima podprogram. Na oddaljenem računalniku se podprogram izvede, rezultati se spakirajo in pošljejo nazaj na naš procesor.

↳ to lahko zelo transparentno napravljeno

PROBLEM:

Kako ta naš proces ve kje je ta podprogram? Ta podprogram se lahko seli, ne vemo kaksno me ima, ...

O.S. upravlja z vsemi temi procesi, upravlja s sporočili.

Vse te naloge, ki ustvarjajo OKOLJE PROCESA

⇒ UPAVNA O.S. OZ JEDRO O.S.

JEDRO OPERACIJSKEGA SISTEMA

- skrbi za vire (koliko jih je, kdo ga uporablja, ...)

PROGRAM

PROCES: program v izvajanju (nekje med začetkom in koncem)

↳ Ali lahko čaka v eni točki?

1) procesor ni na voljo (preveč procesov, ...)

2) proces zahteva nek vir, ki ni na voljo (tiskalnik, lahko ga uporablja on sam proces in to od začetka do konca)

- če procesu nasilno vzamemo procesor lahko prekinemo kjerkoli

↳ potem moramo pograti ta proces → podatkovne strukture

⇒ beleži podatke o okolju, v katerem se izvaja proces.

Vse te podatke združimo v **DESKRIPTOR PROCESA**

Programu dodelimo deskriptor da dobimo proces

DREVESNA (HIERARHIČNA) STRUKTURA PROCESOV

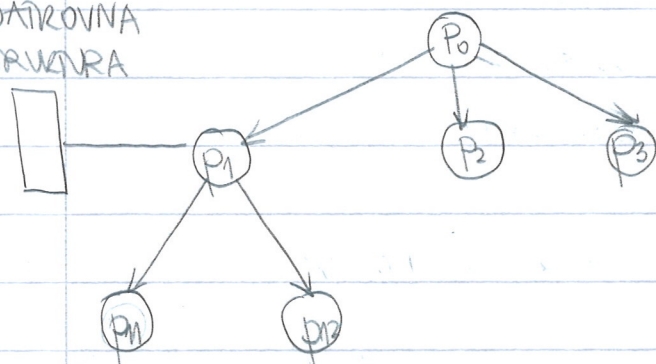
VR: deskriptor, ki opazuje stanje vira; če ni na voljo, bo treba čakati za vsak vir.

VIR: vsaka stvar na katero je treba čakati

PODATKOVNA STRUKTURA = deskriptor: vir ima tudi deskriptor, ki opisuje stanje tega vira.

Vsi procesi so shranjeni v podatkovni strukturi = DREVO PROCESA

PODATKOVNA STRUKTURA



Nad temi procesi izvajamo operacije s pomočjo primitivnih operacij jedra

Ko izvajamo operacije nad temi procesi, jedro OS mora tudi izvajati operacije nad ujimi. Vedno bo treba izvajati operacije tudi nad deskriptorji

Koliko časa trajajo te operacije?

Včasih preklon traja dolgo. To stane. Treba je shraniti notranje stanje procesorja

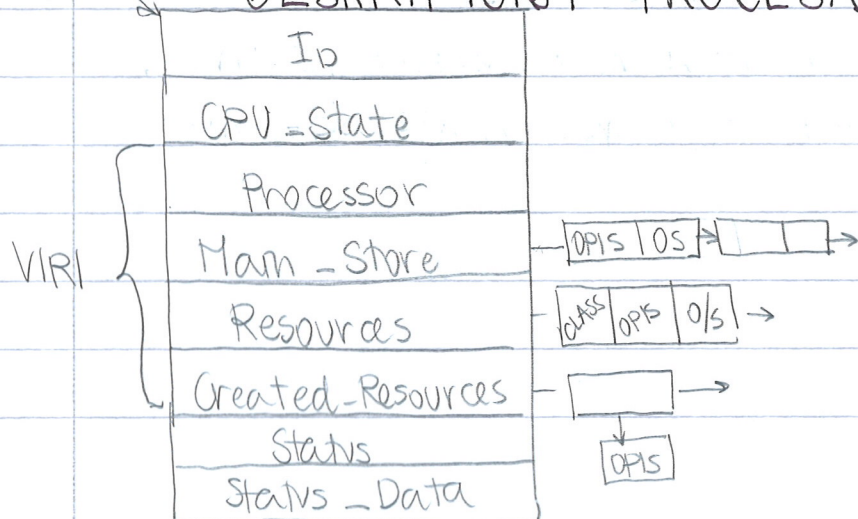
⇒ Različne vrste procesor:

A) v osnovni - hierarhični odnos

B) NITI (thread) - ko ni treba vsega okolja spremeniti

↳ skupni pomnilnik ↳ hitrejša preklona, isto okolje

P kazalec
DESKRIPTORJI PROCESA



p.Id

1) **Id**: s kazalcem p ($p := \text{Get-Pointer}(ep)$) določimo proces.

Uporabnik pozna proces po imenu (string). V ID je ime procesa

⇒ jedro OS vedno najde ta proces.

2) **CPU-state**: proces ima na voljo procesor. Ta procesor ima notranje registre. Ko se proces izvaja, se ti notranji registri spreminjajo. Prihajajo iz enega v drugo stanje. To je tudi notranje stanje procesa. Spreminjajo se pogostni biti.

indexi, programski števec ⇒ kazalec, ki kaže na naslov naslednjega ukaza, ki se bo izvajal

→ Zapomni notranje stanje procesorja. Lahko kompleksna struktura

3) **Processor**: celoštevilčna spremenljivka, ki pove, na katerem procesorju se izvaja proces.

4) **Main-Store**: seznam opisov centralnega pomnilnika, ki ga ta proces uporablja. Lahko iz večjega števila kosov.

Opis: začetna adresa, dolžina, opis strani

Boolevska spremenljivka: O/S: proces ima ta pomnilnik sam ali si ga deli s kom Owned/Shared

→ ti podatki so tudi osnova za zaščito pomnilnika

5) **Resources**: Dve vrsti virov:

A) POTROŠNI - proces jih ne more zbrisati (tiskalnik)

B) NEPOTROŠNI - proces jih lahko zbriše

6) **Created-Resources**: viri, ki jih proces lahko kreira ali izniči

↳ potrošni viri: primer: sporočilo, ki ga kasneje uničimo.

↳ vir, ker ga če en pošlje, ga drug čaka

Semafor tudi vir: en ga kreira, drug čaka

Viri, ki niso potrošni, ohranimo in vrnemo v inventurni seznam.

7) **Status**: proces prehaja med različnimi statusi

↳ pet različnih statusov.

A) **Running**: kadar se proces zares izvaja, ima procesor na voljo.

B) **Ready: a (active)**: takrat kadar bi se lahko izvajal če bi mel procesor na voljo.

pr: 50 procesov, 1 procesor → časovno prepletanje →

→ 1 v izvajanju, 49 v stanju Ready: a

↳ take procese imamo shranjene v čakalni vrsti

RL (Ready List): čakalna vrsta

Ready: s (suspended): če rabimo proces, ga začasno umaknemo

C) **Blocked: a**: če proces želi vr, ki v tistem ni na voljo

pr: tiskalnik bo zaseden, proces se ne bo mogel

izvajati tdi če ima procesor

↳ proces čaka na vr; vsakemu vrvi priredimo čakalno vrsto.

Blocked: s: suspended

8) **Status Data**: kazalec na čakalno vrsto v kateri se proces nahaja

9) **Parent**: pr: p1 kazalec na p0 {podatka o

10) **Progeny**: naslednji kazalec na p1 & p2 {drevesni strukturi

11) **Priority**: podatek (celoštevilska spremenljivka 0..n) pove koliko je proces pomemben

↳ Ta podatek se določa statično (naprej), če proces uporablja

Y/I ali CPU - tiskalnik uporablja Y/I enoto damo višjo

prioriteto, določena s predvideno dolžino obdelave.

di namučno (časovno prepletanje) → to stane

DESKRIPTOR VIRA

ld	
Sr	
Creator	
Avail	
Waiters	→ [] → []
Allocator	

1) **ld**: zunanje ime vira

r := Get-Pointer(er) → da kazalec na deskriptor vira

2) **Sr**: Booleova spremenljivka: pove ali je vr potrošen ali zaporedno uporaben: Serial reusable

3) **Creator**: kazalec na proces, ki je kreiral ta vr.

Sr viri: bo kreiral po (osnovni proces); potrošne vire kreira katerikoli proces; funkcija Garbage collection: pospravi (očisti) vse funkcije

4) **Avail**: inventurni seznam vseh virov tega tipa, ki so na voljo

↳ zasežemo in vračamo vire

↳ pr: tiskalnik: različne karakteristike; laserski, iglični

5) **Waiters**: čakalna vrsta tistih procesov, ki čakajo pred tem

virom → za vsak od teh čakajočih procesov je uavajen s podrobnostmi tega procesa; prostor za odgovor lahko FIFO, ni pa uujno

6) **Allocator**: metoda kako ta vr dodeljujemo.

↳ strategija prvega najboljšega prilagajanja

Problem: različnim virom pripadajo različni algoritmi

↳ za vsak vr posebej

26. 12 (2ura) ∇
○

→ zunajne me procesa
procedure Destroy(ep);

begin

Sched := false; // ni treba klicat Schedulerja

p := Get-Pointer(ep); // kazalec na deskriptor tega procesa

kill(p);

if sched then Scheduler; // če je sched na true, kličemo Scheduler ← globalna sprem.

end;

procedure kill(p);

begin // če najdemo proces, ki je v stanju running, ga ustavimo

if p↑Status = 'running' then begin Stop(p); Sched := true end;

Delete(p↑Status-Data, p);

for all g in p↑Progeny do kill(g); // tudi vse naslednike odstranimo

for all r-unit in (p↑main-store ∪ p↑Resources) do

if owned(r-unit) then

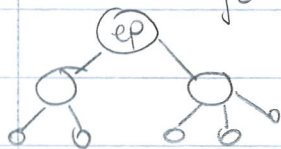
♥ begin r := Find-Descriptor(r-unit), Insert(r↑Avail, r-unit) end;

for all r in p↑Created-Resources do Destroy-RC(r);

Delete-Process-Descriptor(p)

end

tu notri je še, da zbrisemo tega otroka staršev, ki ga je kreiral



Ko zbrisemo proces zbrisemo tudi vse njegove naslednike.

Najlažje bi bilo, zbrisati proces, če bi zbrisali deskriptor, vendar tega ne moremo narediti, ker ima proces v lasti neke zmožljivosti (npr. tiskalnik). Če bi zbrisali proces, bi te zmožljivosti

ostale zasedele in nedostopne.

Zbrisati moramo VSE povezave in to storimo po zgornji kodi

Rekurzivni klici: ko neka procedura, program kliče samega sebe

Procedura ^{kill} kliče samo sebe za prekinitev naslednikov

če jih uporabimo narobe, gre lahko to $\rightarrow \infty$. Zato moramo imeti znotraj teh procesov nekaj, s čem lahko to konamo. V našem

primeru imamo končno drevo.

Rekurzivne klice sestavimo s pomočjo sklada.

♥ Za vse procese, vire, pogledamo, če so v lasti tega procesa.

Te vire je treba vriniti nazaj v njihove inventurne sezname.

Zopet moramo poiskati deskriptorje.

Potem pogledamo, potrošne zmožljivosti, ki jih je proces kreiral. Teh ne rabinu vriniti, ampak jih samo izboljšamo, uničimo deskriptor te zmožljivosti.

Ko odstranimo vse povezave, zbrisemo še deskriptor procesa

OPERACIJE

1) Create-RC

2) Destroy-RC

3) Request

4) Release

1) Kreiramo deskriptor te zmožljivosti. Napolnimo vse postavke v deskriptorjih. To se zgodi enkrat v življenju zmožljivosti. Če jih uporabljamo zaporedoma se to zgodi ob inicializaciji

2) Ko zbrisemo deskriptor, je treba vse te procese opustiti. Če ne, se lahko zgodi da en proces neskončno dolgo čaka eno zmožljivost

4) S to operacijo proces vrne zmogljivost

→ zmagajo me zmogljivosti, ki jo želi proces
povratni parameter, podobnosti
izpolnitve
zahtev

3) procedure Request (er, Reg-details, Alloc-details)

begin

r := Get-Pointer(er); me procesa

Insert (r↑.waiters, (* Reg-details, Alloc-details);

Allocator (r, L); naključno dodeljevanje zmogljivosti. Ta obravnava vse procese v čakalni vrsti na ist način

self := on-L; seznam procesov, ki jim je bila dodeljena zmogljivost. Potem rabišmo ta seznam pogledati

negacija,
predpostavimo,
da
v čakalni vrsti

while L ≠ NIL do

begin

if L↑.proc ≠ * then begin pregledamo en proces za drugim

p := L↑.proc;

Insert (RL, p); proces, ki je bil v čakalni vrsti ga damo v Ready List, ki čaka? je na procesor

p↑.Status-Data = RL; kazalec procese damo na Ready List

if p↑.Status = 'blocked-a'

then p↑.Status := 'ready-a' else p↑.Status := 'ready-s'

end else self := on-L; če je na seznamu mu ni treba spreminjati vseh teh podatkov če je v 'running'

L := L↑.next;

end

if self := on-L then begin

*↑.Status := 'blocked-a';

*↑.Status-Data := r↑.Waiters;

Process [*↑.Processor] := NIL;

Delete (RL, *);

end; zbrisemo ga iz čakalne vrste in na koncu pokličemo še Scheduler

Scheduler

end

proces, ki nekaj izvaja *

če L proces ni naš klicni, potem je to eden, ki je čakal in je v startu blokirani. Temu je treba spremeniti status in jih dati v drugo čakalno vrsto. Za našega pa to ni potrebno

4) procedure Release (er, Rel-details);

begin

r := Get-Pointer(er); podrobnosti o enotah, ki jih vračamo

Insert (r↑.Avail, Rel-details);

Allocator (r, L); dodeljevalec zmogljivosti

if L ≠ NIL then begin

repeat

p := L↑.Proc;

Insert (RL, p);

p↑.Status-Data = RL;

if p↑.Status = 'blocked-a' then

p↑.Status := 'ready-a'

else

p↑.Status := 'ready-s';

L = L↑.next;

until L := NIL;

Scheduler

end

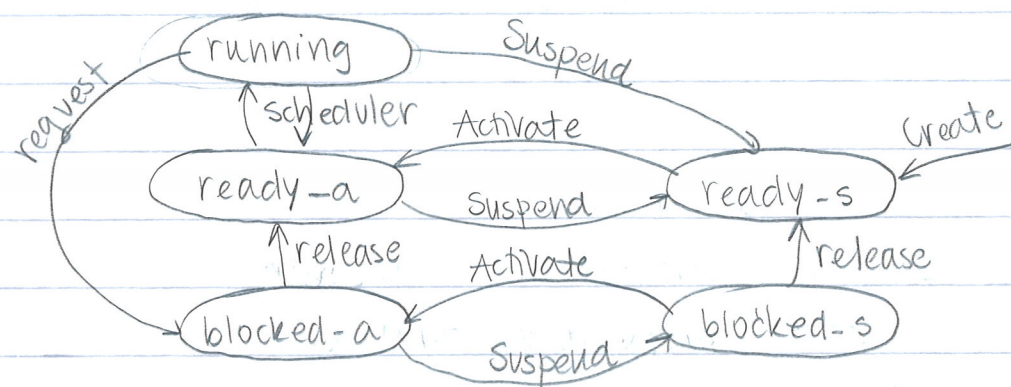
end

↳ proces vrne zmogljivost

Takrat vrnemo Allocator, če se kakšna zahteva ne da izpolniti. Na tem seznamu. (Seznam vseh procesov, ki jim je bil zahtev izpolnjen)

Procesu spremenimo status in ga damo v RL. To naredimo do konca, dokler seznam ni prazen. Na koncu pokličemo Scheduler.

Kako se spreminja status procesa znotraj sistema?



Ko proces pride v sistem, ga nek proces kreira z operacijo Create in proces pride v Ready-s. Če bi bil v ready-a bi mu Scheduler lahko takoj dodelil proces. Potem ga mora aktivirati. Nato pride v ready-a. Takoj lahko dobi procesor. Scheduler lahko klica z več mest

↳ dodeljevalec procesorja procesom

Procesor mu lahko scheduler tudi vzame. Lahko iz strahu suspendiramo proces. Najprej ga moramo zopet aktivirati.

Lahko se zgodi, da proces sam da request in pride v blocked-a (če je zahteval več kot je bilo na voljo)

Lahko drug proces izvede release in zopet naš proces pride v ready-a.

→ KRATKOROČNI: kateri proces bo dobil procesor med tistimi, ki so že v pomnilniku
 PROBLEM: mora se izvajati na istem procesorju kot vsi procesi iz daj jedro
 OS na istem procesorju

rešitev
 problema

IZVEDBA SCHEDULER-JA

Lahko je en sam Scheduler, ki ga vsi kličejo, lahko je več kopij Schedulerja

PRIORITETNI SCHEDULER

Imamo neko število procesorjev N_p

Sr... množica procesov, ki se izvajajo na procesorjih

Sa... procesi, ki so v statusu Ready-a ali Running

Schedulerji skušajo doseči, da so na procesorjih, ki so na voljo, procesi z najvišjo prioriteto.

Pomembno je, kdaj pokličemo Scheduler. Tudi če so notri procesi z najvišjo prioriteto se ne bo nič zgodilo, če ga ne bomo klicali.

```

procedure Scheduler; // mora pregledati RL (čakalna prioriteta vrsta)
begin

```

```

  cur_prio := n; // tekoča prioriteta
                  // n... najvišja možna prioriteta
  deferred := NIL; // takrat, kadar določen procesor, na katerem bo treba
                  // namestiti nek proces. Na ta procesor se izvaja
                  // tudi scheduler

```

čakalno
 do konca,
 da se
 scheduler
 zaključ

```

loop
  Find-Highest(cur_prio, p); // tekoča prioriteta
  if p = NIL then Wrap-Up(deferred); // zaključek cele rutine
  Allocate-ble-CPU(p, found, g-mm, deferred); // dodamo procesu procesor,
  ki je mirujoč.

```

```

  → if 7 found ^ g-mm ^ Priority < p ^ Priority
  then Preempt(g-mm, p, deferred);
  end

```

```

end

```

če nismo našli praznega procesorja in če mm prioriteta poiskaneja manjša od prioritete od našega, naredimo Preempt; zamenjamo te dva procesa
 ↳ namesto tistega, ki se izvaja namestimo našega

Λ logični in
V ali

Find Highest.. gre skozi čakalno vrsto in najde proces z
najnižjo prioriteto

```
procedure Find-Highest (cur-prio, p);  
begin
```

p := NIL; // tekoči prioritetni nivo

found := false; // ni prioritete nivoja

while (p = NIL) ∧ (cur-prio > 0) do // dokler ne najdemo nič in prazno

begin

cur := RL↑. [cur-prio] first; // postavimo kazalec na prvega

while 7 found ∧ (cur ≠ NIL) do // dokler nismo na spodnjem nivoju

if cur↑. proc↑. status = 'ready-a' then // če status ready-a ga
bomo izbrali

begin

p := cur.proc;

found := true;

end else cur := cur.next;

cur-prio := cur-prio - 1 // če nismo nobenega našli
spremenimo (znižamo) prioriteto

end

end

Če ga najde javi p (kazalec na procesorja), če ne najde pa javi NIL

Če je NIL prazen gremo v zaključno fazo Wrap-Up. Normalno

gremo v Allocate-Idle-CPU. Dodelimo procesor, ki je mirujoč

```
procedure Allocate-Idle-CPU (p, found, g-min, deferred);
```

begin

nismo našli nobenega
praznega procesa

found := false; cpu = 1, g-min := p; // na proces z najnižjo prioriteto

while 7 found ∧ (cpu ≤ np) do // dokler nismo našli
prazni proces

begin

if Process [cpu] = NIL then // če proces, ki se izvaja na tem procesorju
prazen proces, potem naš proces
namestimo na ta procesor.

begin

found := true; // našli smo prazen prostor

Process [cpu] := p // v deskriptor napišemo, da se izvaja proces p

p↑. Processor := cpu; // na katerem procesorju se izvaja

p↑. Status := 'running';

if cpu = *↑. Processor // če ta proces enak na katerem se izvaja
Scheduler ne smemo narediti preklopa

then deferred := p else Load State (cpu, p↑. cpu-State);

end else // zakasnjeno, ga vključimo → na drugem pa lahko nalozimo
na koncu status procesorja

if Process [cpu]↑. Priority < g-min.Priority // če procesor ni prazen
pogledamo samo prioritete

then g-min := Process [cpu]

cpu := cpu + 1

end

end

Če je prioriteta nižja od tega, ki kliče, poišče procesor na katerem
se izvaja proces z nižjo prioriteto. V tem primeru kasneje
zamenjamo procesor.

Da proces namestimo na procesor, moramo namestiti vse
stativne podatke procesa (zadnji se namesti programski števec).
To se ne sme zgoditi na istem procesorju na katerem se izvaja Scheduler.

Preempt → zamenjava

```
procedure Preempt (g-min, p, deferred)
```

begin

g-min↑. Status := 'ready-a'; // je imel status running, zdaj bo
imel status ready-a

p↑. Status := 'running'; // je imel status ready-a, zdaj bo
imel status running

p↑. Processor := g-min↑. Processor; // p se izvaja na istem procesorju
kot se je izvajal g-min

Process [g-min↑. Processor] := p;

if g-min := * // če se izvaja na istem procesorju kot se izvaja
Scheduler bo treba čakati

then deferred := p

else Switch (g-min, p) // če ne, napravimo switch

end

g-min: vzamemo proces in gledamo p. če se na tem procesorju izvaja tudi scheduler, moramo počakati da se to zaključi in šele na koncu lahko to izvedemo.

procedure Switch(p, g)

begin

c := p↑.Processor

if p ≠ * then Interrupt(c)

Store-State(c, p↑.CPU-State);

if p = * then Adjust-Instr-Counter(*↑.CPU-State);

Load-State(c, g↑.CPU-State); // naloži stanje novega procesa v notranje registre procesorja

end

Procesu p vzamemo procesor in na njem nastavimo proces g.

Enostavno če se p izvaja na drugem procesorju

Problem, če se to izvaja na istem procesorju kot scheduler

Poskrbeti moramo, da se shranijo statusi procesor.

Že ko pokličemo Scheduler, si moramo zapomniti status procesa

Čim nastavimo programski števec, se začne izvajati proces.

procedure Wrap-Up(deferred)

begin

if *.Status ≠ 'running' then Switch(*, deferred)

else Scheduler

end

PREKINITVE

→ časovni prekinitveni signal (periodično _____ signal)

→ v sistemu se lahko zgodi nek dogodek ob katerem pridemo do Schedulerja

Do Schedulerja pridemo:

• proces prostovoljno to napravi

↳ zaključi z delom in zahteva vir, ki ni na voljo

nonPreemptive strategy

• vedno poskrbimo za to, da so na procesorju procesi z najvišjimi prioritetami

Preemptive strategy

V tem primeru moramo vedno klicati Scheduler

↳ sistem je ODZIVEN (lahko imamo sistem v mehkem realnem času)

• Samo določeni procesi imajo prednost. To predpišemo s tabelami vmesne strategije

Po: loop

Request(Dio-queue, 2, (p, ioprogram));

Request(Path, D, path)

Generate-ID-instr;

Imitate-IO;

Request(IOint, path, message);

Decode-Int;

Produce-Status;

Release(Path, path)

Release(Dio-end, i, p, Complete-Status);

end

D: !

```
Release(Dio-queue, ioprog); // pošlji zahtevo za IO operacijo  
Request(Dio-end, 2, Complete-Status); // čaka status
```

← interrupt handler

```
H: Store-State(*↑.Processor, *↑.CPU-State);  
  *↑.Status := 'ready-a';  
  Process[*↑.Processor] := NIL;  
  Determine-Servise-Process-Pb;  
  Release[Interrupt-Class, (Pb, int-details)];
```

POMNILNIK

Pomnilniki običajno počasnejši kot procesorji

Čim hitrejši pomnilnik je dražji

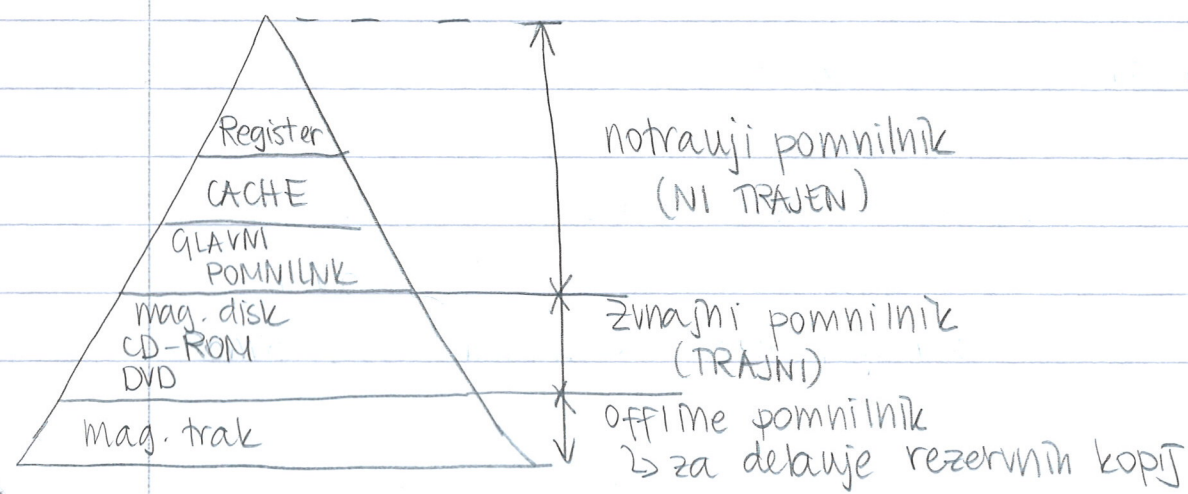
Pomnilniki z veliko kapaciteto imajo počasnejši dostop

Hitrega pomnilnika uporabimo manj in je dražji

Potrebujeemo velike in hitre pomnilnike

⇒ trajni .. zadržijo podatke če izpade napajanje

⇒ ne trajni .. izgubijo podatke če izpade napajanje



Notraujni pomnilniki najhitrejši. Najhitrejši so REGISTERI (so najdražji). Lahko bi celotni pomnilnik naredili na tak način, ampak bi bilo predrago.

Cache.. razdeljeno na posamezne zloge

Glavni pomnilnik.. počasnejši od glavnega procesorja. To stanje se ne izboljšuje

Cache.. medpomnilnik, je hitrejši od glavnega pomnilnika

Pri zunajnih pomnilnikih gre prenos na različne načine. So počasnejši od notraujnih pomnilnikov.

→ cena pada od vrha proti dnu.

→ kapaciteta pada od vrha proti dnu

→ čas dostopa narašča od vrha proti dnu

→ frekvenca dostopov do podatkov pada proti dnu.

Če imamo nek podatek, se na ostalih nivojih pojavljajo kopije. Na nekaterih delih se podatek spreminja → problem RAZNOLIKOST KOPIJ PODATKOV

Neuskladenost.. podatkov na različnih nivojih

CACHE → delovanje s cache-om

NIVO 1 1000 B $0.1 \mu s = T_1$

NIVO 2 100 000 B $1 \mu s = T_2$

Če vzame proces podatek iz prvega nivoja, zasede ga v $0.1 \mu s$. Če podatka ni na prvem nivoju, moramo najprej ugotoviti kje je podatek in ga iz spodnjega nivoja prenesemo na zgornji.

Če bi bili vsi podatki na drugem nivoju bi bilo to $1.1\mu s$.
 Lahko se zgodi, da jih je nekaj na drugem, nekaj na prvem
 nivoju. Čas dostopa odvisen od pogostosti zadetkov
 (frekvenca zadetkov)

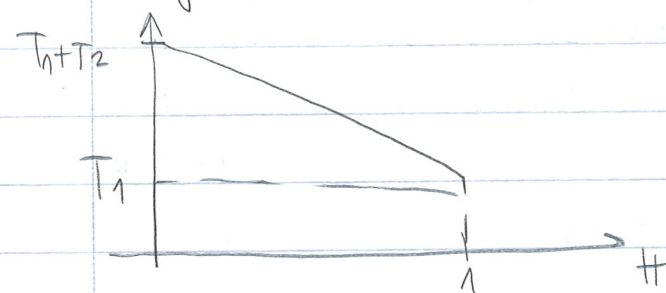
H.. delež podatkov, ki jih dobimo v zgornjem pomnilniku.
 št. med $0 < H < 1$

1-H.. podatki na spodnjem nivoju

⇒ čas dostopa:

$$\text{povprečni čas dostopa} = H \cdot T_1 + (1-H)(T_1+T_2)$$

Če jih je 95% na prvem nivoju, je čas dostopa $0.15\mu s$



Cache.. NIVO 1; Glavni pomnilnik.. NIVO 2

Če frekvenca zadetkov v cache-u zelo visoka,
 potem s cache-om pospešimo delovanje
 NAČELO LOKALIZACIJE REFERENC < če to ne drži, bi cache še vse poslabšal

⇒ frekvenca podatkov v nižjih nivojih nižja

V praksi je res frekvenca podatkov v cache-u višja

Cache: haren hardware-sko

V zadnjem času se uporabljajo večnivojski cache-i:

npr: Intel i7 ima 3 nivojski cache:

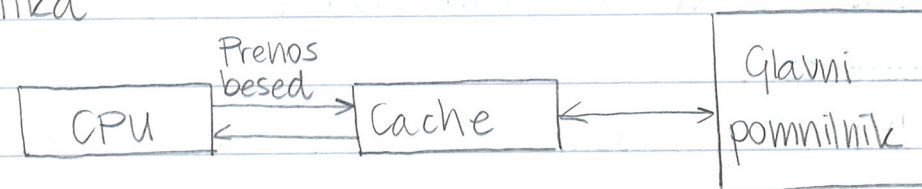
32 kB

256 kB

8 MB → skupen za vsa 4 jedra

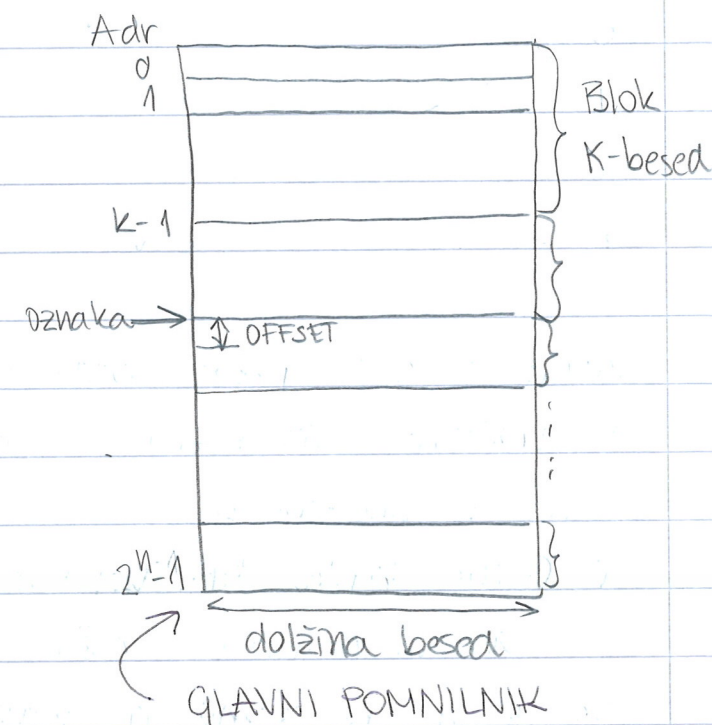
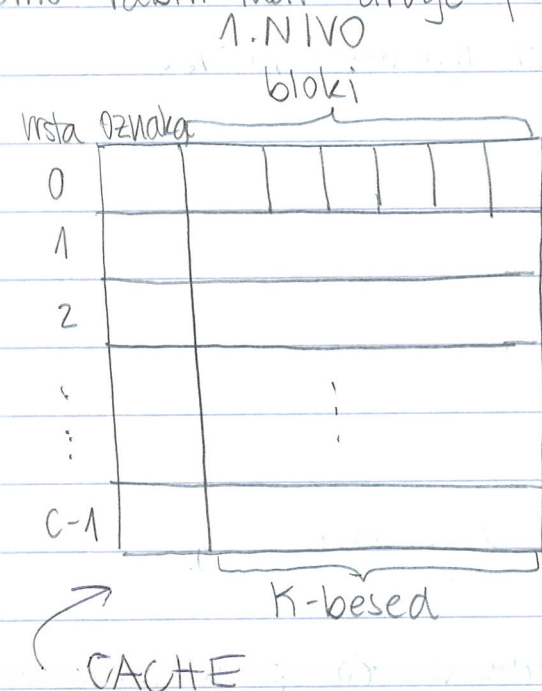
CACHE

delovanje procesorja omejujo s hitrostjo zaseganja podatkov iz pomnilnika



V cache je kopija nekega dela podatkov v pomnilniku

Če smo potrebovali en podatek iz bloka, je zelo verjetno, da bomo rabili tudi druge podatke iz tega bloka.



cena pomnilnika

$$K = 2^m$$

$$M = \frac{2^n}{K} = 2^{n-m}$$

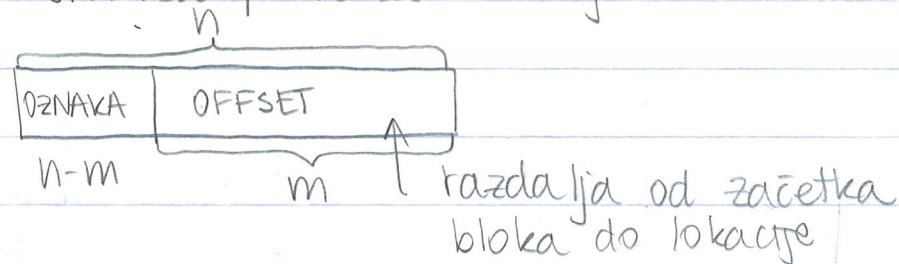
število blokov, ki imajo K besed

Cache sestavljen iz C blokov: $C \ll M$ v cache manjše število blokov iz pomnilnika

Adresa predstavlja vsako posamezno besedo v pomnilniku

$n-m \rightarrow$ številka bloka

$m \rightarrow$ številka podatka znotraj bloka



Oznako v cache prenesemo posebej. Offset določi položaj podatka v bloku

Pomembno koliko je velik cache. Že majhen bistveno izboljša delovanje sistema

Velikost bloka vpliva na načelo lokalizacije referenc. Če ga večamo blok se frekvence zadetkov večajo. Če ga večamo čez neko mejo se frekvence zadetkov spet zmanjšuje. Če ni praznega mesta v cache, moramo iz cache odstraniti eno rezino. Katerno?

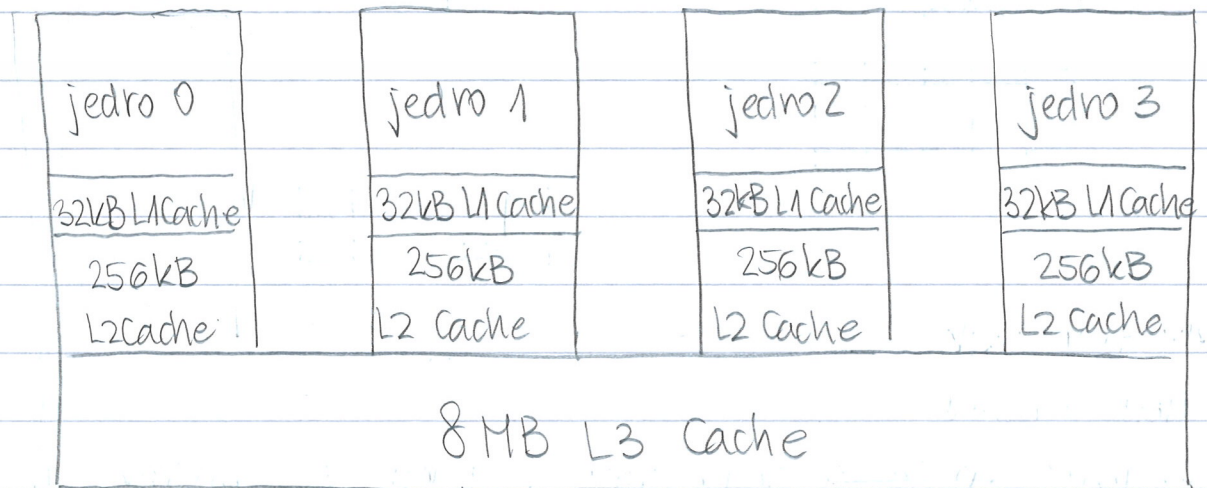
→ izločimo tisto rezino, ki vsebuje blok, ki ga najdlje časa nismo uporabljali

V cache lahko tudi vpisujemo podatke. Ko ga vpišemo v cache, ga moramo prenesti (vpisati tudi v glavni pomnilnik)

Če takoj prenesemo nimamo različnih kopij podatkov. (to je problem o večprocesorskih sistemih). To je malo zamudno.

Če v glavni pomnilnik zapišemo samo tedaj, ko blok ne potrebujemo več, imamo različne kopije podatkov.

Intel Core i7 4jedrni procesor $\rightarrow$ multiprocesor, ki ima 4 procesorje



2 različna "memory controllerja"

UPRAVLJANJE POMNILNIKA (MEMORY MANAGEMENT)

Upravitelj pomnilnika:

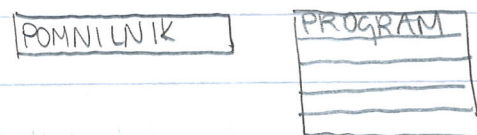
- skrbi, da so procesi med seboj izolirani (se ne motijo)
 - skrbi za avtomatsko dodeljevanje in upravljanje pomnilnika
 - uporabnik dobi pomnilnik po potrebi
 - transparentno za uporabnika
 - podpira modularno programiranje
 - skrbi za zaščito in kontrola dostopa do pomnilnika
- Vse to lahko uresničimo z virtualnim pomnilnikom in datotečnim sistemom

ORGANIZACIJA POMNILNIKA

Pomnilnik mora znati prenaslavljati dinamično

Deskriptor	
Program	reference med različnimi kosi programa program se sklicuje na podatke
Podatki	
Skład	Za spremenljivke tipa automatic OS mora za vsak proces/program poznati kazalce na deskriptor, program & sklad

PREKRIVKE (OVERLAY) Lahko samo en kos programa damo na pomnilnik. V tem primeru razdelimo program na več kosov. V nekem trenutku v pomnilniku je samo en kos. Tehnika prekrivkov omogoča, da programer lahko sprogramira kateri del programa se bo prenesel v pomnilnik.



Na kakšen način opravljamo razkosanje pomnilnika?
↳ Da se narediti tudi brez virtualnega pomnilnika

HEMA FIKSNEGA RAZKOSANJA

-pomnilnik razdelimo na fiksne segmente (statične kose)
do časa generiranja sistema

HEMA ENAKIH PARTICIJ

8M	0.5
8M	
8M	
8M	
8M	
8M	

①

-lahko so ti kosi različnih dolžin

HEMA NEENAKIH PARTICIJ

III	2M
III	2M
	4M
II	8M
	16M

① Če nek program zahteva določen kos se mu pri prvi shemi dodeli eno particijo;

Več možnosti:

Interna fragmentacija: programu manjši od particije
če program večji od particije programer uporabi
tehniko prekrivkov.

Če en proces ni aktiven, ga OS umakne in natoži drugega

HEMA neenakih particij

Rešimo oba problema iz sheme enakih particij. Majhnemu programu se dodeli majhen del. Vendar se pojavijo dodatni problemi. Če je program majhen in so vse majhne particije zasedene. Lahko vsakemu programu predpišemo eno particijo in ta program čaka. Moramo pred vsako particijo narediti takšno vrsto.

HEMA DINAMIČNEGA RAZKOSANJA

OS
P1
P4
P2
P3

Če se vmes P2 izloči iz pomnilnika
damo P4

Pomnilnik obravnavamo kot eno samo enoto, ki jo dodeljujemo po potrebi.

Zunajna fragmentacija pomnilnika:

Kako se temu izognemo?

KOMPAKTIFIKACIJA: če OS oceni, da je to potrebno, premakne programe tako, da jih pomnilnik združi.

pr: P3 premakne gor $\Rightarrow$ Ta shema mora omogočati dinamično naslavljanje

Poskrbimo da je teh lukenj čimmanj.

Kakšne algoritme razvrstitve programe uporabimo?

- 1) strategija najboljšega prilagajanja
- 2) strategija prvega prilagajanja $\rightarrow$ najboljša ∇
- 3) strategija naslednjega prilagajanja

ena ura manjka ∇

TRANSAKCIJE

ko se transakcija zaključi morajo biti njeni podatki (rezultati) trajni, sicer pa ne smejo biti rezultati.

Operacije, ki se smejo izvajati: (nobene druge se ne smejo)

• **Read(X)** .. prebere vrednost podatka X } izvajata se kot

• **Write(X, val)** .. v X se upiše vrednost value } nedeljivi enoti

X je lahko podatek ali blok, ki se izvaja kot nedeljiva enota.

Če se več operacij izvaja na istem podatku, se izvajajo kot nedeljiva enota (v neznanim vrstnem redu)

• **Start** .. transakcija se začne

• **Commit**

• **Abort**

Začne z operacijo Start. Ob Start dobi transakcija identifikacijo id zaključiti s Commit. Če transakcija izvede Abort, je prekinila to delo in je treba odstraniti vse posledice te transakcije.

Problem: Vnaprej je treba predvideti vse možnosti!!

Vse operacije transakcij morajo iti preko sistemske podatkovne baze. Transakcije ne smejo komunicirati direktno med seboj. Nadzor nad nekaterimi operacijami

Efekti transakcij

- Efekti, ki jih je transakcija sama zapisala

$\rightarrow$ Ta transakcija ima vpliv na druge transakcije

Če izvedemo Abort, je treba posledice (efekte) transakcije razveljaviti. Medtem je lahko nekdo drug prebral to in uporabil. Treba je razveljaviti tudi vse druge transakcije. To je takšna razveljavitev. Potem moramo obnoviti predhodno stanje. Zato si ga tudi moramo zapomniti.

Če transakcija izvede Commit šele potem, ko vse transakcije od katerih je odvisna izvedejo Commit, je to transakcije, ki se lahko okrepi (recovery)

Problem: transakcija komunicira z drugo transakcijo preko V/I enote.

Algoritme podatkovnih baz naredimo tako, da se izognemo kaskadnih abortov:

- transakcija lahko bere samo tiste podatke, ki so jih zapisale že zaključene transakcije

Če hočemo narediti Abort, je treba obnoviti predstliko. To predstliko je težko ugotoviti

- zapis v podatkovne baze zakasujemo toliko časa, dokler se transakcija ne zaključi (Commit)

Pri izvajanju operacij nad podatki nismo povsem svobodni. Striktne operacije: z njimi se izognemo kaskadnim abortom.

Ne smemo izvesti Abort

Zanesljiva podatkovna baza: noben podatek se ne sme zgubiti
↳ zahteva po času in prostoru

Niinkonta podatkovna baza: hitrost

Problem transparency: določene algoritme mora uporabnik sam narediti. Lahko jih pa podatkovna baza sama naredi

Problem izpada komunikacije: računalnik bolj zanesljiv kot komunikacije (šum, napaka, izpad, ...)

Bodisi ko se transakcija zaključi morajo biti njeni podatki (rezultati) trajni, sicer pa ne smejo biti rezultat.

Operacije, ki se smejo izvajati (nobene druge se ne smejo)

- Read(X): prebere se vrednost podatka X } Izvajata se ko
 - Write(X, val), ... v X se vpiše vrednost valve } nedeljiva enota
- X lahko podatek ali blok; blok se izvaja kot nedeljiva enota
Če se več operacij izvaja na istem podatku, se izvajajo kot nedeljiva enota (v neznanim vrstnem redu)

- Start: transakcije se začne
- Commit
- Abort

začne z operacijo Start, ob Start dobi transakcija id identifikacijo. Vsi programi se bodo izvajali nad to trans. z id. Trans. se zaključi s Commit. Če trans. izvede Abort, je prekinila to delo in je treba vse posledice te trans. odstraniti

Problem: Unaprej je treba predvideti vse možnosti !!

Vse operacije transakcij morajo iti preko sistemske podatkovne baze. Transakcije ne smejo komunicirati direktno. Nadzor nad nekaterimi operacijami

Effekti transakcij

efekti, ki jih je transakcija sama zapisala

Problem: ta transakcije ima vpliv na drugih transakcijah

Če izvedemo Abort je treba posledice (efekte) transakcije razveljaviti

Medtem je lahko, nekdo drug prebral to in uporabil. Treba je razveljaviti vse druge transakcije. Kaskadne razveljavitev. Potem moramo obnoviti predhodno stanje, zato si ga tudi moramo zapomniti

Če transakcija izvede Commit, šele potem ko vse transakcije od katerih je odvisna je to okrepite transakcije

Problem: transakcija komunicira z drugo transakcijo preko V/I enote

Algoritme podatkovne baze naredimo tako da se izognemo kaskadnim abortom
- transakcije lahko bere samo tiste podatke, ki so jih zapisale zaključene transakcije

Če hočemo narediti Abort, je treba obnoviti predstliko. To predstliko je težko ugotoviti.

- zapisujemo/zapisi v podatkovne baze zakasujemo toliko časa, dokler se transakcija ne zaključi (Commit)

Jej zdravo, jej v menzi na faksu!

Poglej si kaj ponujamo danes: <http://menza.fri.uni-lj.si>



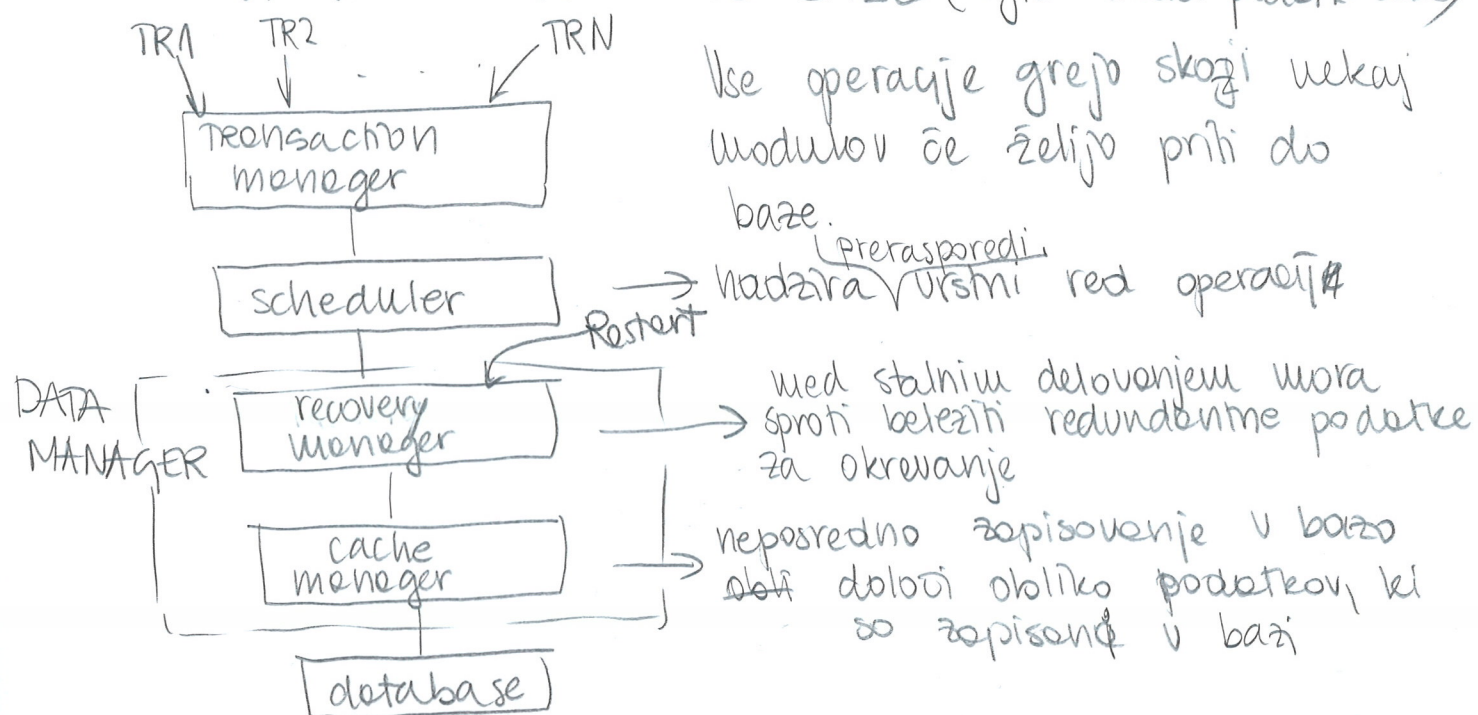
Pri izvajanju operacij neod podatki nismo povsem svobodni.
 Striktne operacije: z njimi se izognemo tiskadnim abortom. Ne smemo izvesti abort.
 Zanesljiva podatkovna baza: noben podatek se ne sme zgubiti.
 zahteva po času & prostoru

Učinkovita pod. baza: hitrost

Problem transparence: določeni algoritmi so tako urejeni, da jih mora uporabnik sam narediti. Lahko jih po podatkovna baza samo naredi

Problem izpada komunikacije: Računalnik bolj zanesljiv kot komunikacija (šum, upraka, izpad)

KONSTRUIRANJE PODATKOVNE BAZE (logični model podatkovne baze)



Distribuiran sistem: v vsakem vozlišču mora biti tak sistem. Upravljalne transakcije komunicirajo med seboj

TRJE V RSTF PODATKOVNIH BAZ

1)

Sistem v enem samem vozlišču. Do njega lahko dostopa večje število uporabnikov preko komunikacijskega sistema. Izvajanja se med seboj prepletajo. Temu se izognemo tako, da zaporedno izvajamo. Tega ne moremo. Zato prepletamo izvajanja in naredimo, da so ta izvajanja ekvivalentna zaporednemu. To je zaporedniško izvajanje.

Ekvivalentno izvajanje: konfliktna ekvivalence: dve konfliktni operaciji, če je vsaj ena operacija operacije zapisa
 branje/branje branje/zapis zapis/zapis
 NI DA DA

ker je na koncu izvajanja transakcij je pomemben samo vrstni red konfliktnih operacij

Izvajanje konfliktnih operacij → protokoli za zaporedniško izvajanje:

- pesimistični protokoli: vsilijo tako izvajanje, ki zagotavlja zaporedniškost
- optimistični: predpostavka: tudi če pride do nezaporedniškosti, je to redko. Potem uporabimo Abort

PESIMISTIČNI: 4 skupine

- z uporabo ključev (locks)
- z uporabo časovnih oznak
- testiranje grafa zaporednosti
- z uporabo več verzij podatkov

Z UPORABO KLJUČOV

Ključ je neka spremenljivka, ki nadzoruje dostop do podatkovnega elementa (podobno semafor). Ključ lahko zasedemo za branje (več transakcij) ali pisanje (ena transakcija). Pred vsakim ključem je zapisovanje je čekalna vrsta

Ne zagotavlja zaporedniškost. Zaporedniškost izvaja protokol dvofaznega zaklepanja (two faced locking): Pravila:

- vse operacije zaklepanja ključev morajo biti izvedene pred operacijami odklepanja ključev.

Kdaj smemo odkleniti ključ

→ PROTOKOL STROGEGA DVOFAZNEGA ZAKLEPANJA

- razporejevalec sproti zaklepa podatke in jih odklene šele po operaciji commit ali abort.

Slobost: podatki so večina časa zaklepani

- lahko pride do zapora deadlock. Ustrezne transakcije je treba abortirati.

→ REZULTAT: ZAPOREDNIŠKO IZVAJANJE

Podatkovne nestete: predpostavka: računalniške ne odpravljamo, (robustni nos podatkov)

Jej zdravo, jej v menzi na faksu!

Poglej si kaj ponujamo danes: <http://menza.fri.uni-lj.si>

1) Transakcijske napake: ko mora izvesti Abort. Če je + (Napaka zapored.)

Trans. sama ugotovi, da je treba narediti abort

Lahko se zanesemo na vsebino obstojnega (disku) ali neobstoječega (centralnega) posrednika.

2) Diskovne napake: pride do okvare vsebine neobstoječega posrednika (centralni). Zanesemo se na vsebino trajnega posrednika (diska)

3) Diskovne napake: uničen del diska. Delamo s podvojenimi kopijami.

Skozi upravnika okrevanja grejo vse operacije razen start. Pri vsaki operaciji si ta upravitelj vse zapiše v dnevnik. Ni dovolj, da te spremembe zapiše v centralni posrednik. Mora tudi na disk. Kakšen je vrstni red zapisovanja? Takrat kadar se izvede operacija Restart mora upravitelj okrevanja ali bo transakciji uničil vse efekte (na podlagi osnovi podatkov dnevnika), popolno razveljaviti ali jih dokončati. Spravi bazo v konsistentno stanje.

4) Diskovne napake: redundanca (arhiviranje) diskov, stalno zrcaljenje diskov. Pranita so več vzporednih diskov (v različni različni diskij.)

2) PORAZDELJENE PODATKOVNE BAZE

Porazdeljena baza, ampak se podatek nahaja na enem mestu ne replirano podatkov. baze

↳ replirane podatkovne baze na različnih mestih

A) Nereplirane

Transakcije se izvajajo na motornem računalniku ali na drugih konsistentno transakcije: v celoti

Na vsakem vozlišču je je sistem pod. baze. Komunikacija preko upravitelja transakcij

- uporaben protokol strogega dwo-faznega zaklepanja

Problem pri zaporednosti: kadar želimo zaključiti transakcijo
↳ ugotavlja se deadlock: globalno na ravnanje

Problem pri commit/abort: ACP (Atomic Commitment Protocol)
Kadar hoče upravitelj zaključiti transakcijo najprej vpraša vse računalnike, ki so bili vključeni v transakcijo ali se izvede commit ali abort. Vsi odgovorijo centralnemu računalniku. Tisti, ki zahteva Abort, ga naredi in se zaključijo sporocila. Ostalim pa pošlje sporočilo bodisi zaključiti ali razveljaviti transakcijo.
Dolgotrajna & težavna zadeva: če en računalnik izpade čaka na odgovor, razpad sistema se zaključek transakcije: zadnja usklejevalna faza

B) Replirane

Isti podatek se nahaja na različnih mestih

Problem: redundanca nečeloma dobro, vendar če so podatki porazdeljeni se branje pospeši. Pri zapisovanjem v bazo moramo spreminiti tudi vse kopije. Uporabimo vse protokole. Zelo zamudno. Dodaten problem: lahko pride do pado. Poseben primer: sistem razpade v otroke, ki med seboj ne komunicirajo. Lahko preprosto delo na vseh lokacijah, (ena lokacija odpade, vse odpade)