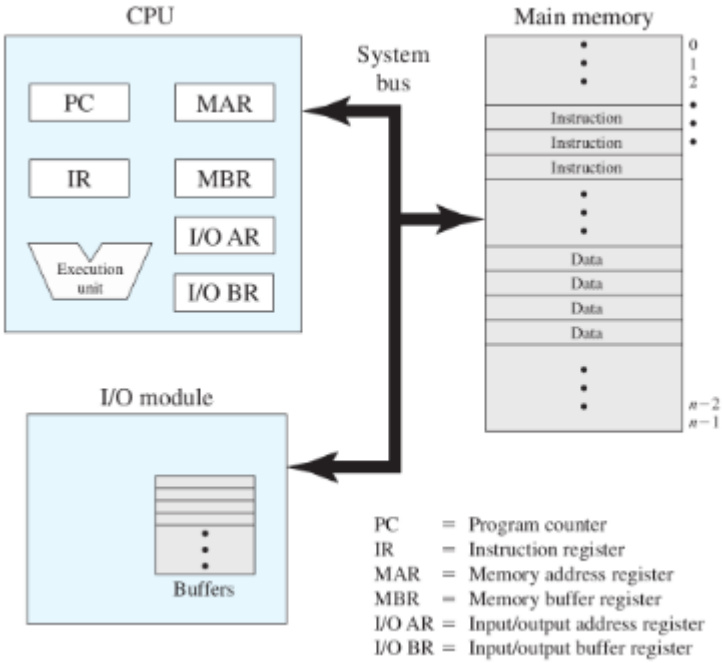
Operativni sistemi 1 - ispitna pitanja

I. Pregled računarskog sistema

1. Delovi računara



Osnovni delovi računarskog sistema su: **procesor**, **ulazno-izlazni moduli**, **memorija** i **sistemska magistrala** koja povezuje ove tri komponente. Sistemska magistrala se sastoji od magistrale podataka i adresne magistrale. U glavnoj memoriji se nalaze i podaci i instrukcije.

Procesor se sastoji od osnovnih registara kao što su:

- **PC (Program Counter)** – sadrži adresu instrukcije koja se trenutno izvršava
- **IR (Instruction Register)** – sadrži operaciju koja treba da se izvrši sa adrese koja se nalzi u PC
- **MBR (Memory Buffer Register)** – registar za memoriju u koji se smešta izračunati rezultat namenjen za upis u memoriju
- **MAR (Memory Address Register)** – sadrži adresu u memoriji na koju treba da se upiše rezultat iz MBR
- **I/O BR (Input/Output Buffer Register)** – registar u koji se kopiraju podaci iz U/I modula tj. U/I bafera
- **I/O AR ((Input/Output Address Register)** – sadrži adresu u memoriji na koju treba da se upiše rezultat iz I/O BR.

2. Procesorski registri i instrukcijski ciklus

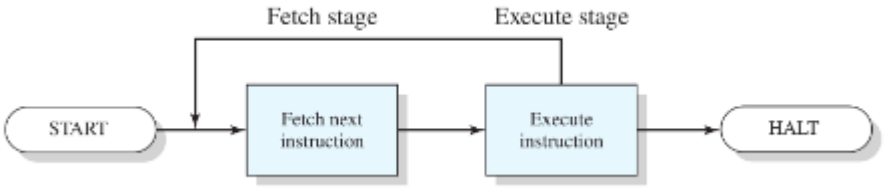
Procesorski registri su:

- **Registri vidljivi i registri nevidljivi korisniku**
- **Upravljački i statusni registri**
- **Indeksni registri** – koristi se za podatke koji se čuvaju redom tj. za nizove
- **Pokazivač segmenta**
- **Pokazivač steka** – čuva lokalne promenljive, argumente funkcija, PSW i PC
- **Program Counter** (PC) i **Instruction Register** (IR)
- **Processor Status Word** (PSW)

Instrukcijski ciklus se sastoji iz dva koraka:

- **Faza donošenja (fetch)**
- **Faza izvršavanja (execute)**

Donešene instrukcije se učitavaju u IR, a PC se povećava tako da pokazuje na sledeću instrukciju.



Vrste instrukcija su:

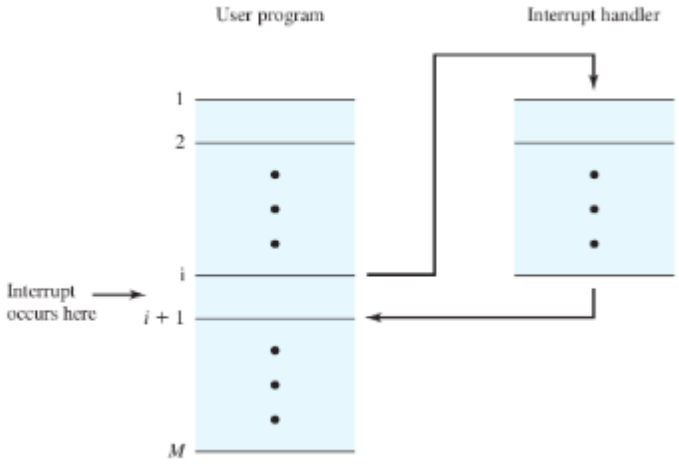
- Transfer **CPU – memorija** i obrnuto
- Transfer **CPU – U/I** (prenos podataka između CPU i U/I modula)
- **Obrada podataka** (aritmetičke i logičke operacije nad podacima)
- **Upravljanje** (instrukcije koje mogu da dovedu do promene redosleda izvršavanja)

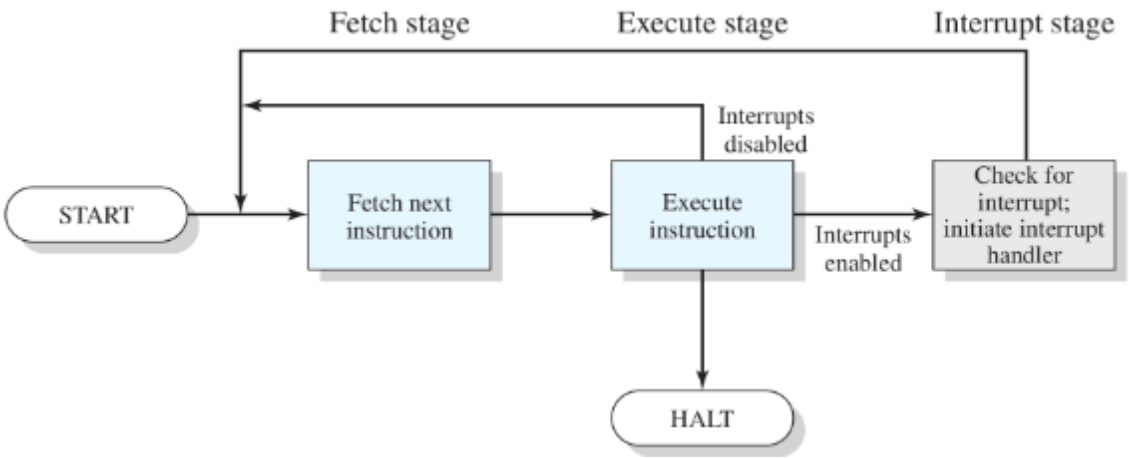
3. Prekidi (Interrupts), višestruki prekidi

Osnovna namena prekida je da se poboljša iskorisćenost procesora.

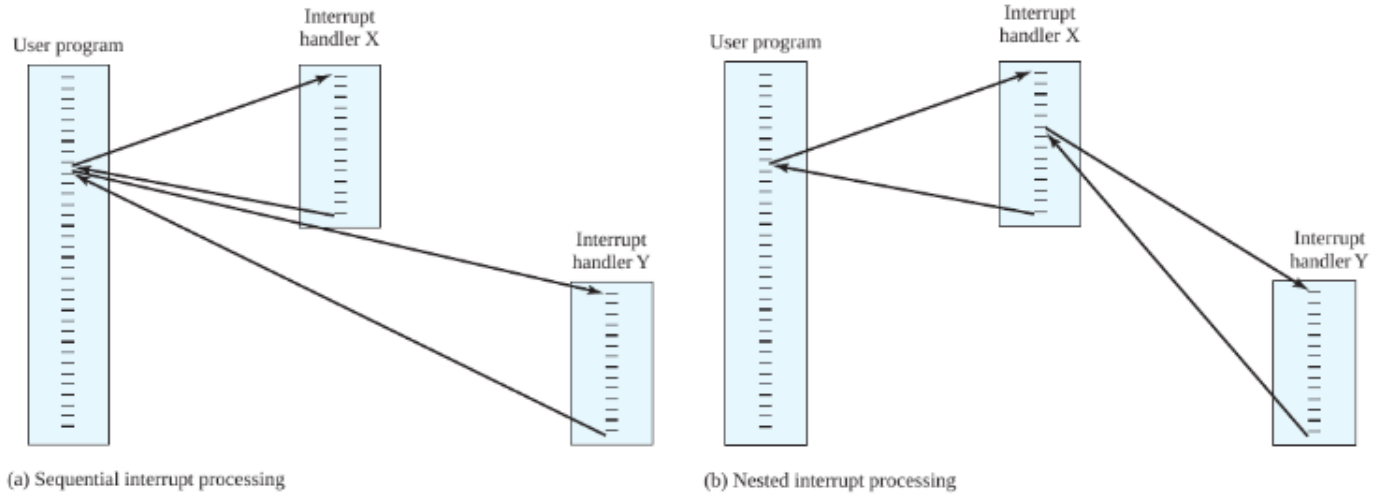
Klase prekida:

- Program – prekoračenje, deljenje nulom, segmentacija (segmentation fault)
- Tajmer – Generiše ga tajmer unutar CPU i na ovaj način OS može redovno da izvršava neke instrukcije.
- UI – generiše ga UI kontroler
- Otkaz hardvera (npr. greška pariteta memorije)
  Na slici je prikazana obrada prekida i programski ciklus sa prekidom





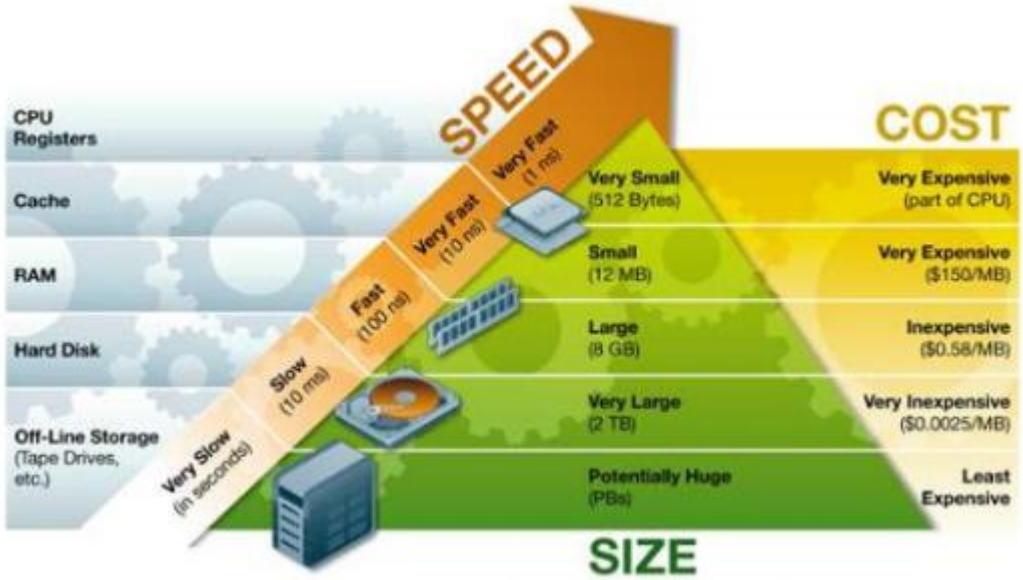
Do višestrukih prekida dolazi kada dođe do prekida tokom izvršavanja neke prekidne rutine. Ovako nastali prekidi se mogu rešiti na ***sekvencijalni*** ili ***ugnježđeni*** način.



Sekvencijalni način funkcioniše tako što se prvo obradi jedan prekid, pa se onda obrađuje drugi prekid.

Ugnježđeni način funkcioniše tako prekid višeg prioriteta može da izazove prekid rada prekida sa nižim prioritetom.

4. Hijerarhija memorija



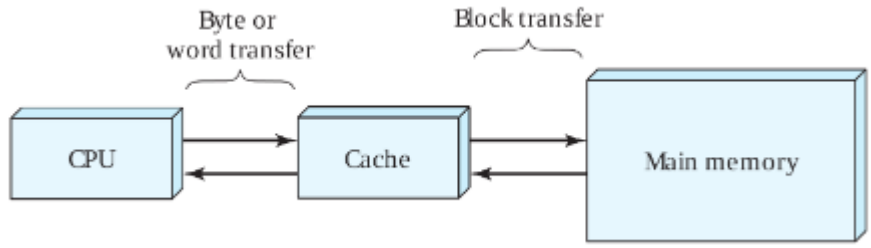
Odnosi koji uglavnom važe kod memorije su sledeći:

- Što je brže vreme pristupa, veća je cena po bitu
- Što je veći kapacitet, troškovi su manji po bitu i manja je brzina pristupa

Na osnovu spuštanja niz hijerarhiju možemo zaključiti sledeće:

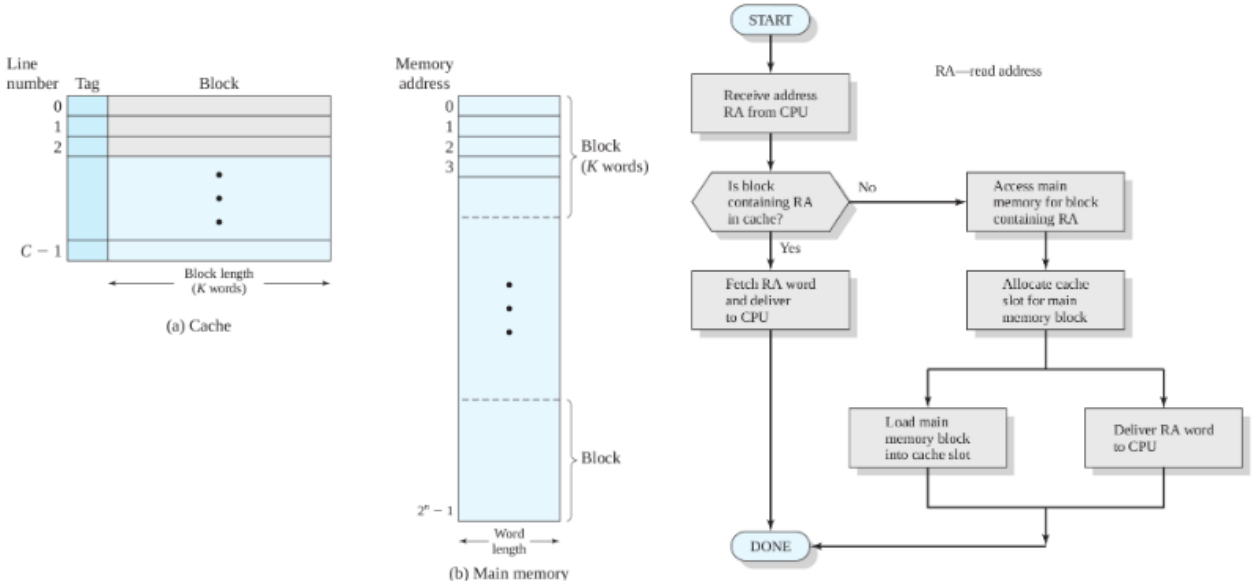
- Smanjuje se cena po bitu
- Povećava se kapacitet
- Povećava se vreme pristupa
- Smanjuje se učestalost pristupa memoriji

5. Princip rada keš memorija



Glavna memorija se sastoji od 2^n reči gde svaka ima n-bitnu adresu. Sastoji se od blokova fiksne dužine K reči. Postoji $M = 2^n/K$ blokova

Keš memorija se sastoji od C slotova koji imaju po K reči, C<<M
Svaki slot sadrži tag koji ukazuje na to koji je trenutno blok smešten u njega. Tag se odnosi na sve adrese koje počinju tom sekvencom bitova.
Osnovna jedinica transfera između keša i CPU je **reč**, a između keša i memorije **blok**.



Objašnjenje algoritma:
Keš kontroler uzima adresu iz procesora i proverava da li se blok koji sadrži ovu adresu nalazi u kešu. Ako se blok nalazi u kešu, keš kontroler uzima taj blok i vraća podatak procesoru. Ako se blok ne nalazi, keš kontroler pristupa glavnoj memoriji i treba da dovuče blok sa tom adresom. Ukoliko su svi slotovi popunjeni, keš kontroler će donešeni blok da upiše na mesto najmanje skoro korišćenog bloka

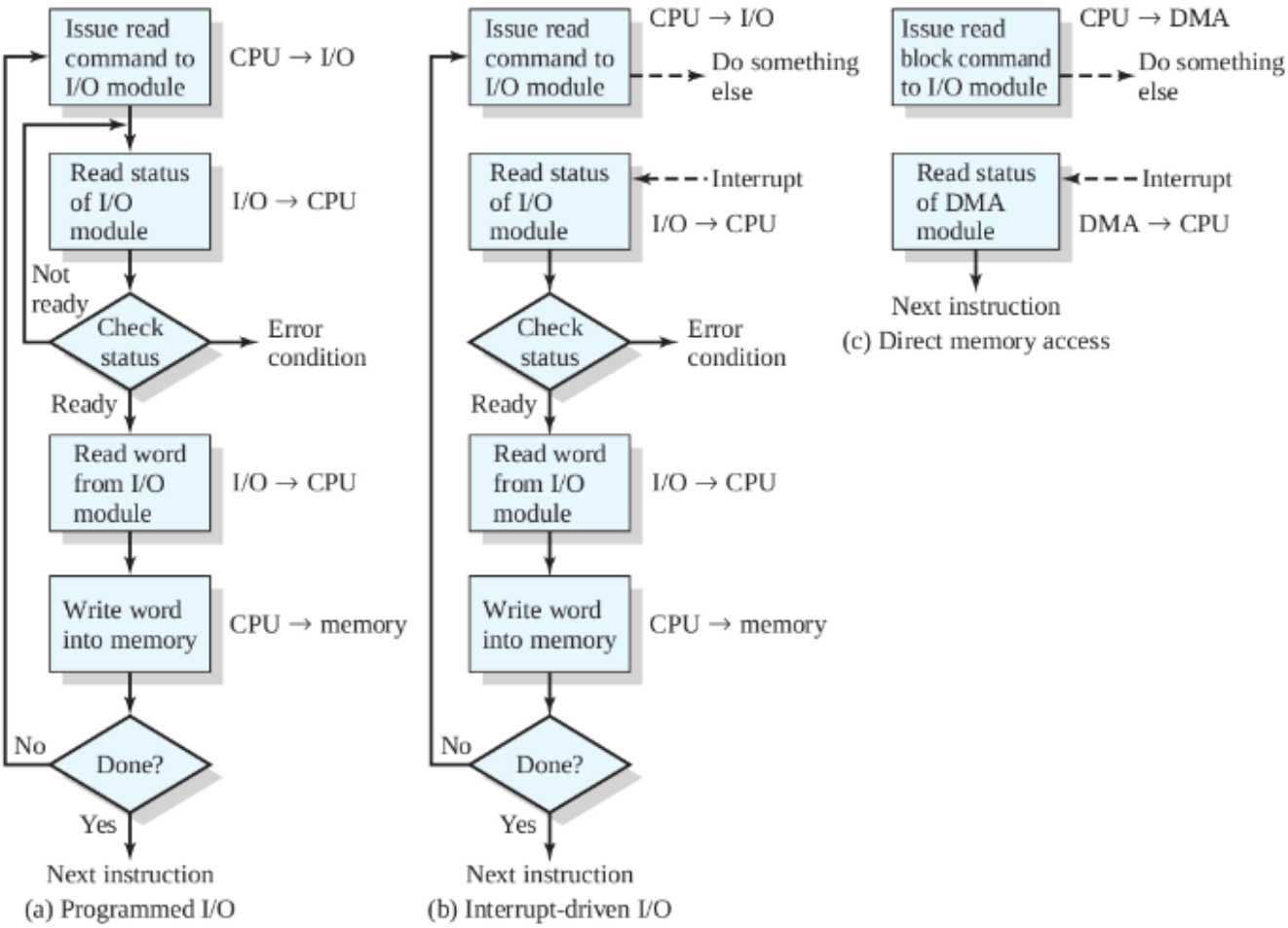
6. U/I tehnike, DMA pristup

Tehnike koje se koriste za U/I operacije su:

- Programirani U/I
- U/I koji se upravlja prekidima
- Direktan pristup memoriji (DMA)

Skup instrukcija U/I su sledeće:

- Upravljanje – aktiviranje uređaja i davanje instrukcija
- Status – testiranje stanja UI modula i njegovih periferala
- Prenos – čitanje/upis podataka između registara CPU-a i eksternih uređaja



DMA kontroler je komponenta koja omogućava da se neko parče memorije prebaci direktno sa U/I modula u memoriju.

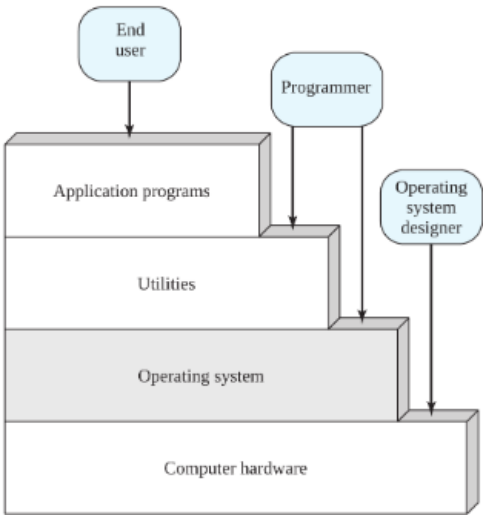
Objašnjenje algoritma pod c): Da CPU ne bi gubio na svojoj efikasnosti, on se obraća DMA kontroleru za neki podatak u memoriji i za to vreme on odlazi da izvršava neki drugi posao sve dok mu DMA ne pošalje prekid da je podatak koji je tražio spreman.

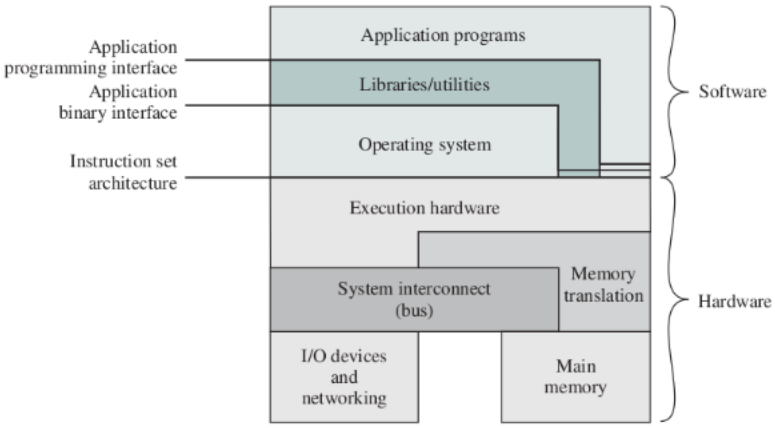
7. Ciljevi operativnog sistema

- Praktičnost – čini računar udobnim za korišćenje
- Efikasnost - pomoću OS-a se računarski resursi efikasnije iskorišćavaju
- Mogućnost evolucije – OS treba da bude konstruisan tako da omogućava uvođenje novih usluga bez ometanja postojećih

II. Pregled operativnih sistema

8. Osnovni nivoi i usluge OS-a, uzroci evolucije OS-a





Usluge OS-a:

- Razvoj programa
- Izvršavanje programa
- Pristup U/I uređajima
- Kontrolisan pristup fajlovima
- Pristup sistemu
- Obrada grešaka i odgovori
- Vođenje evidencije

Kernel je jezgro OS-a. OS predaje upravljanje CPU da bi izvršio neki koristan rad, a zatim OS preuzima upravljanje dovoljno dugo da bi pripremio CPU za sledeći deo posla. U memoriji se uvek nalati Kernel kao i drugi delovi OS-a koji se trenutno koriste.

Uzroci evolucije OS-a su:

- Razvoj hardvera – grafički terminali, prozori, straničenje, mobilni uređaji...
- Nove usluge – zahtevi korisnika, novi trendovi i sl.
- Ispravke grešaka

9. Istorijat – serijska obrada, prosti sistemi paketne obrade

Ne postoji operativni sistem i programi su pisani u mašinskom jeziku koji su se učitavali pomoću čitača bušenih kartica

Glavni problemi serijske obrade su bili:

- Raspoređivanje vremena – operator je zakazivao vreme za računarom koje je moglo biti neiskorišćeno ili nedovoljno
- Vreme uspostavljanja posla – obično duže od vremena pravog računanja

10. Istorijat – multiprogramirani sistemi paketne obrade

Poavljuje se prvi OS nazvan IBSYS kompanije IBM. Uvodi se poseban program tzv. **monitor** i korisnik više nema direktan pristup računaru.

Monitor se sastoji od **rezidentnog monitora** i **pomoćnih programa.** Na ulaz se postavlja ceo paket poslova i programi se učitavaju jedan po jedan i sukcesivno se izvršavaju. Takođe, uvodi se poseban jezik za upravljanje monitorom **JCL (Job Control Language)**.

Prosti sistemi paketne obrade čija je osnovna komponenta monitor koji je preteča operativnog sistema.

Monitor je jednostavan program koji zavisi od sposobnosti procesora da izvršava instrukcije iz različitih delova memorije kako bi privremeno preuzeo ili predao upravljanje.

Hardverske osobine naprednijih prostih sistema paketne obrade:

- Zaštita memorije
- Tajmer
- Privilegovane instrukcije
- Prekidi

11. Istorijat – sistemi sa deljenjem vremena

Nastali su sa uvođenjem transakcionih sistema i interaktvnih terminala. Osnovni motiv je smanjenje odgovora sistema korisniku.

12. Dostignuća – procesi i upravljanje memorijom

- Izolacija procesa
- Automatsko dodeljivanje i upravljanje
- Podrška modularnog programiranja
- Zaštita i kontrola pristupa
- Dugotrajna memorija

13. Dostignuća – upravljanje resursima, zaštita i bezbednost

Raspoređivanje i upravljanje resursima

- Nepristrasnost – fer pristup resursima
- Distinkcija odgovora – npr. davanje prednosti procesima koji čekaju U/I
- Efikasnost – uvećanje propusne moći, smanjenje vremena odziva, prihvatanje što većeg broja korisnika

Zaštita i bezbednost

- Raspoloživost - zaštita od prekida rada
- Poverljivost – ko sme da čita podatke, a ko ne
- Integritet podataka – zaštita od nedozvoljene izmene
- Autentičnost – identitet korisnika i ispravnost podataka

14. Dostignuća – distribuirani sistemi, klasteri, virtuelizacija

Distribuirani sistemi

Klasteri predstavljaju međusobno povezane računare koji imaju više jezgara, a međusobno komuniciraju preko mreže

Postoje 3 vrste virtuelizacije: puna virtuelizacija, paravirtuelizacija, virtuelizacija nivoa OS-a

Paravirtuelizacijom se postižu performanse virtuelne mašine koje se približavaju performansama prave.

15. Dostignuća – savremeni OS-evi, Windows NT, Linux, sličnosti i razlike

Savremeni OS:

- Savremeni OS umesto velikog monolitnog kernela, ima mikrokernel sa najosnovnijim funkcionalnostima, a ostatak funkcionalnosti se premešta u posebne servete. Portabilnost na račun performansi. Mikrokernel je bolji iz razloga što omogućava da se promeni željeni modul a da se pritom ne kompajlira ceo kernel (ušteda vremena).
- SMP (Multiprocesorski sistemi)
  U sistemu postoji više procesora koji dele RAM i UI veze i svi procesori mogu da vrše iste operacije. Prednosti SMP u odnosu na jednoprocesorski sistem su: performanse, raspoloživost i skalabilnost.
- Višenitna obrada (Multithreading) – deljenje procesa na delove (niti)

Windows

- Izmenjena arhitektura mikrokernela
- Mnoge sistemske funkcije koje se nalaze izvan kernela se izvršavaju u režimu kernela zbog performansi

Komponente Windows-ovog kernel režima

1. **Executive** - upravljanje memorijom, procesi, niti, bezbednost, keš, PnP...
2. **Kernel** - jedini deo koji ne može da se prekida ili *swap*-uje
3. **Hardware Abstraction Layer (HAL)** - sloj apstrakcije hardvera
4. **GUI** - na Windows-u je deo kernela

Linux

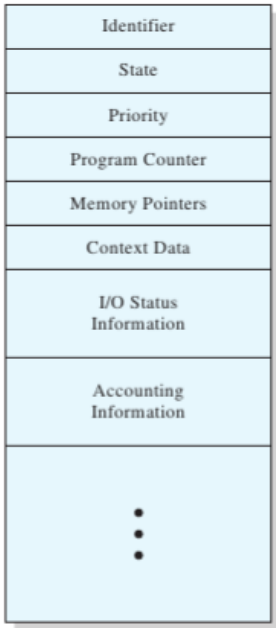
- Open-source
- Linux ima modularnu arhitekturu tj. moguće je dodavanje novog modula bez ponovnog kompajliranja kernela
- Moduli se mogu vezivati dinamički (svi moduli su spakovani u jednu povezanu listu)
- Svaki modul je određen tabelama: tabelon modula i tabelo simbola
- Tabela simbola definiše simbole koje kontroliše dati modul, a koriste se na nekom drugom mestu

III. Procesi

16. Definicije procesa, struktura upravljačkog bloka

Proces je program u izvršavanju.

Struktura upravljačkog bloka



Svaki proces mora da ima PCB (Process control block tj. upravljački blok).

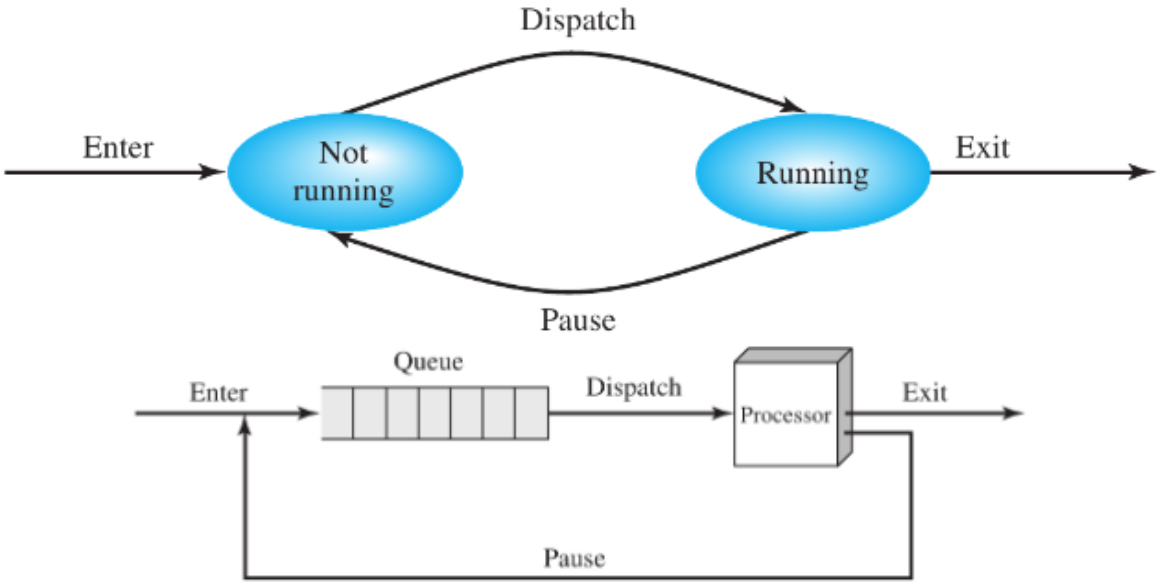
U upravljačkom bloku se nalazi:

- Identifikator procesa (Process id - PID)
- ID procesa roditelja (Parent process ID - PPID)
- Stanje procesa (Running, not running, blocked...)
- Prioritet
- PC (Program Counter)
- Pointeri na memoriju
- Konteksni podaci
- Ulazno-izlazni status
- Accounting information pamti koliko je proces do sad trošio procesor

17. Stanja procesa - model sa 2 stanja

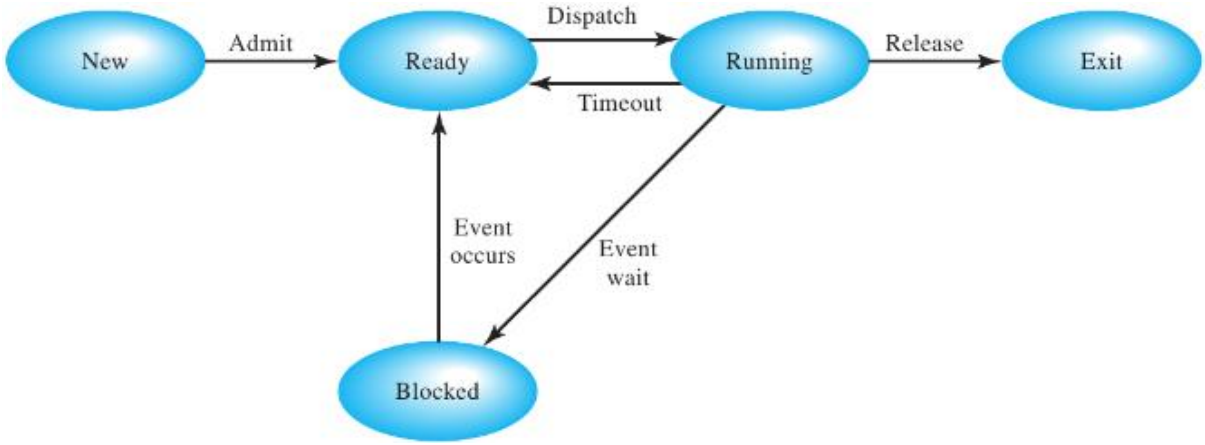
Trag procesa - Ponašanje procesa se može predstaviti sekvencom instrukcija koje se izvršavaju za taj konkretan proces

Dispečer – relativno mali program koji je deo OS-a zadužen za prebacivanje procesora sa procesa na proces. Može biti **prekidni** i **neprekidni**. Neprekidni dispečer ne može da prekine proces dok se izvršava, što nije slučaj kod prekidnog.



Postoje dva stanja procesa: Running i Not running. Kada proces uđe u sistem on je prvo u stanju ***not running***, a onda dispečer odlučuje da ga stavi u stanje running tj. da se izvršava. Kad mu istekne vreme predviđeno za rad, proces se vraća u stanje not running. Onog trenutka kada se proces završi, on izlazi i ide napolje.

18. Stanja procesa - model sa 5 stanja



1. **Running** - Proces koji se trenutno izvršava
2. **Ready** - Proces koji je spreman za izvršavanje kada dobije procesor na korišćenje
3. **Blocked** - Proces koji ne može da nastavi sa izvršavanjem dok se ne pojavi neki događaj, npr. kraj U/I operacije
4. **New** - Proces koji je upravo kreiran, ali ga OS još nije stavio na red spremnih. Obično su to procesi kojima je napravljen upravljački blok, ali još uvek nisu učitani u memoriju.
5. **Exit** - Proces koji je na normalan način prekinuo s radom ili ga je OS skinuo sa reda iz nekog drugog razloga.

19. Kreiranje i terminacija procesa, redovi blokiranih procesa

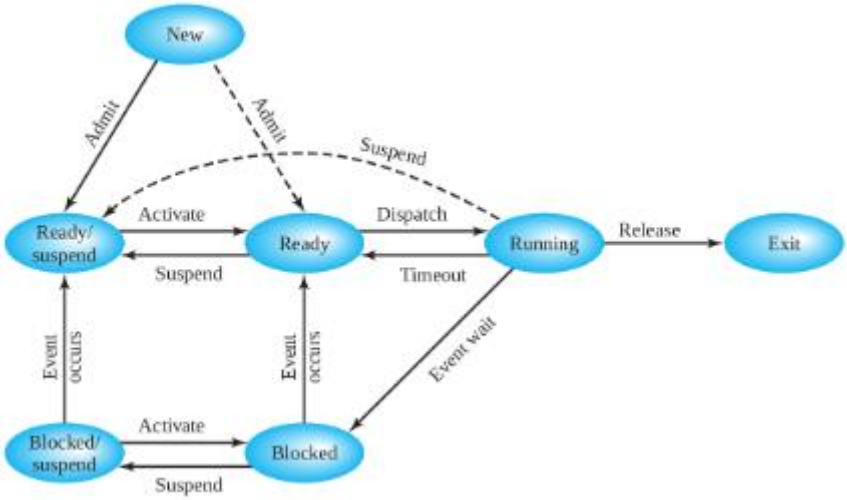
Kreiranje procesa:

- Dodeljuje mu se jedinstveni identifikator
- Dodeljuje mu se prostor za elemente slike procesa
- Inicijalizuje se PCB – sve se setuje na 0 sem PC koji se setuje na početnu adresu tekst segmenta
- Postavljaju se odgovarajuće veze npr. veza u redu Ready
- Kreiranje ili proširivanje ostalih struktura – npr. obračunske informacije

Terminacija procesa:

1. Normalna terminacija
2. Prekoračenje vremenskog limita (*CPU time* ili *Wall Clock time*)
3. Nedostupnost memorije
4. Kršenje granica memorije
5. Greška zaštite
6. Aritmetička greška
7. Prekoračenje vremena (proces čekao predugo na neki resurs)
8. Otkaz U/I
9. Pogrešna instrukcija
10. Privilegovana instrukcija (proces pokušao da koristi kernel mode instrukciju)
11. Pogrešna upotreba podataka
12. Intervencija operatora ili OS-a
13. Terminacija procesa roditelja
14. Zahtev procesa roditelja

20. Stanja procesa - model sa 2 suspendovana stanja



Stanja procesa

1. **Ready** - Proces je u RAM-u i raspoloživ za izvršavanje
2. **Blocked** - Proces je u RAM-u i čeka na događaj
3. **Blocked/Suspended** - Proces je na disku i čeka na događaj
4. **Ready/Suspended** - Proces je na disku ali će biti raspoloživ za izvršavanje čim dođe u RAM

Suspend stanje je stanje u kom se proces prenosi sa RAM-a na disk kako ne bi bespotrebno zauzimao RAM s obzirom da već dugo čeka na neki događaj.

Kada proces dugo čeka na neki događaj, on biva prebačen u Blocked/Suspend. Čim se dogodi događaj koji čeka, prelazi u stanje Ready/Suspend jer ne može odmah da uđe u trku za procesor pošto je na disku.

Novi, pristigli proces možemo da stavimo u stanje Ready/Suspend ukoliko nemamo dovoljno RAM memorije, a ukoliko imamo on će odmah ući u trku za procesor.

Proces može da pređe iz stanja Running u stanje Ready/Suspend ako dođe proces visokog prioriteta mada je ovo retka situacija.

21. Elementi PCB-a

Identifikacija procesa

- Identifikator procesa
- Identifikator procesa roditelja
- Korisnički identifikator

Informacije o stanju procesa

- Registri vidljivi korisniku
- Upravljački i statusni registri (PSW, PC, ...)
- Pokazivači steka - jedan ili više dodeljenih sistemskih stekova

Informacije za upravljanje procesom

- **Raspoređivanje i informacije o stanju**
  - Stanje procesa (*Ready*, *Blocked*, ...)
  - Prioritet
  - Informacije vezane za raspoređivanje - recimo broj do sada izvedenih instrukcija
  - Događaj - Identitet događaja koji proces čeka da bi nastavio s radom
- **Strukturiranje podataka** - recimo pointerska veza sa sledećim procesom u listi
- **Međuprocesna komunikacija** - markeri, signali i poruke
- **Privilegije procesa**
- **Upravljanje memorijom** - pokazivači na delove tabele stranica virtuelne memorije
- **Vlasništvo nad resursima i iskorišćenje** - npr. otvoreni fajlovi

22. Komutiranje moda i promena stanja procesa, izvršavanje funkcija OS-a

Komutiranje moda je kada proces A bude prekinut i treba da se izvršava proces B, odnosno da procesi promene svoja stanja (Running->Ready, Ready->Running).

Mehanizam	Uzrok	Upotreba
Prekid (*Interrupt*)	Eksterni u odnosu na tekuću instrukciju	Reakcija na asinhroni spoljni događaj
Zamka (*Trap*)	Pridružen izvršenju tekuće instrukcije	Rukovanje greškom ili izuzetkom
Poziv supervizora	Eksplicitni zahtev	Poziv funkcije operativnog sistema

Prekid – reakcija na događaj koji se dešava kao npr. prekid sistemskog sata, UI prekid

Zamka – greška npr. segmentation fault, deljenje nulom

Poziv supervizora – pozivanje funkcija OS-a npr. otvaranje fajla (fopen)

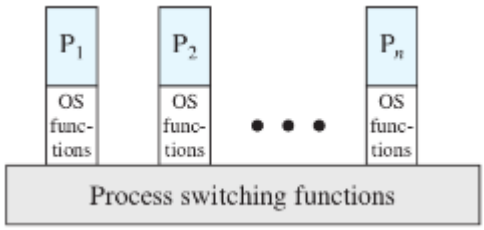
Komutiranje moda

1. PC registar se setuje na početnu adresu rukovaoca prekidom (*interrupt handler*)
2. Procesor se prebacuje iz korisničkom moda u kernel mod, pošto rutina za obradu prekida može sadržati privilegovane instrukcije

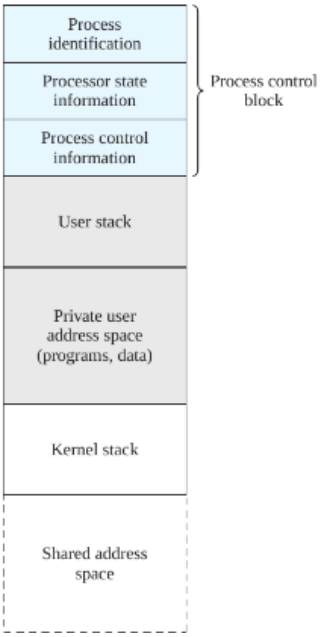
Promena stanja procesa

1. Sačuvati kontekst proceora, PC, PSW i druge registre
2. Ažurirati PCB procesa koji je trenutno u *Running* stanju. Šta se sve menja?
3. Pomeranje PCB-a u odgovarajući red (Ready, *Blocked on Event i*, ...)
4. Odabir nekog drugog procesa za izvršavanje
5. Ažuriranje PCB-a odabranog procesa
6. Ažuriranje memorijskih struktura
7. Učitati stanje procesora koje je bilo aktuelno kada je poslednji put radio

Izvršavanje funkcija OS-a



Kernel stack služi za funkcije OS-a.



IV. Niti, SMP i mikrokerneli

23. Koncept i nadležnosti niti, primeri upotrebe

Operativni sistem izvršava više procesa u isto vreme. Svaki proces dobija po deo procesorskog vremena i to nam stvara utisak da se oni izvršavaju paralelno. Često svaki odprocesa izvršava jednu radnju u jednom trenutku. Sa više niti u okviru jednog procesamožemo postići da jedan proces izvršava više radnji u jednom trenutku, svaka nit posebnuradnju. Niti dele deskriptore i memoriju procesa u ovkiru koga su pokrenute.

Koncepti:

1) Vlasništvo nad resursima – u datom vremenskom trenutku, procesu može biti dodeljena odgovarajuća memorija, stek, UI uređaj...
2) Raspoređivanje/izvtšavanje – izvršavanje procesa prati trag kroz jedan ili više procesa. Njegovo izvršenje može da se prepliće sa izvršenjem ostalih procesa u sistemu.

Nit:

Ukoliko se koncept izvršenja u potpunosti odvoji od procesa vlasništva nad resursima, onda je proces u mogućnosti da poseduje jedan ili više konkurentnih tragova izvršenja.

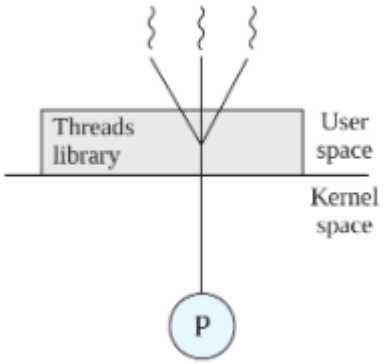
Nadležnosti niti

1. **Stanje izvršenja niti** (*Running*, *Ready*, itd.)
2. Sačuvan **kontekst niti** kada se ne izvršava (PC, registri, ...)
3. **Zaseban stek**
4. **Prostor za lokalne varijable**
5. **Pristup memoriji svog procesa**, zajedno sa svim ostalim nitima tog procesa

Primeri upotrebe

- **Pozadinske niti u GUI programima** - postiže se utisak o bržem odzivu sistema
- **Asinhrono procesiranje** - primer je nit koja u kancelarijskim paketima automatski u određenim intervalima snima *backup* fajla koji se obrađuje
- **Povećanje brzine izvršenja** - recimo jedna nit može da čita podatke sa diska i da često ulazi u blokirano stanje, dok druga može da obrađuje podatke. Na multiprocesorskim sistemima, ovo posebno dolazi do izražaja.
- **Modularna struktura programa** - Različite vrste aktivnosti mogu se implementirati putem niti.

24. Niti korisničkog nivoa



Prednosti

1. Prebacivanje niti ne zahteva privilegije kernela
2. Raspoređivanje niti može se uvesti specifično za datu aplikaciju
3. ULT-ovi se mogu izvršavati na svakom OS-u

Mane

1. Ako se blokira jedna nit, blokirane su sve niti unutar datog procesa
2. Ne mogu se iskorisititi prednosti multiprocesiranja

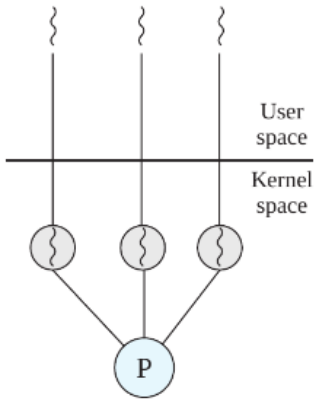
To i nisu baš prave niti. Kernel vidi samo jedan proces, a u tom procesu je korišćena biblioteka *Thread library*. Korisnik je napravio program koji radi iz više niti i time je organizovao svoj program.

Ovim se ne postiže nikakvo dodatno ubrzanje jer kernel vidi samo jedan proces

Sled koraka

1. Proces B izvršava nit 2 koja se blokira čekajući na U/I
2. Kontrola se predaje kernelu koji pokreće UI akciju, i daje kontrolu drugoom procesu
3. Kada proces B potroši svoj vremenski isečak, postaje *Ready*
4. Dolazi se do tačke kada biblioteka niti predaje kontrolul Niti 1 procesa B, pri čemu Nit 2 ostaje u stanju *Blocked*

25. Niti kernel nivoa



Prednosti

1. Bilo koja aplikacija se može kreirati kao višenitna (primer je *Windows*)
2. Kernel može istovremeno rasporediti više niti jednog procesa na više procesora
3. Blokiranje jedne niti ne znači blokiranje procesa

Mane

1. Zahteva se prebacivanje u režim kernela i nazad za komutiranje niti

Svaki proces unutar kernela ima niti i kernel sve to vidi. Korisnik i kernel vide isto u ovom slučaju.

26. SMP, Flynn-ova taksonomija, arhitekture paralelnih procesora

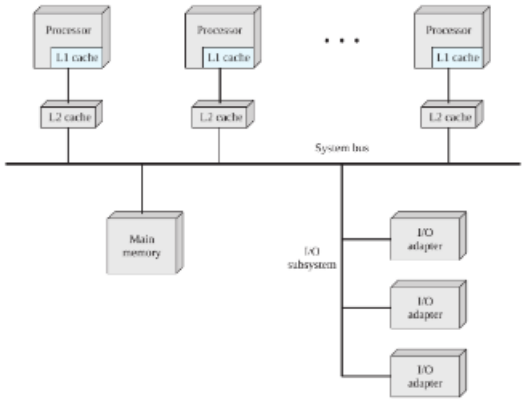
SMP – Symmetric MultiProcessing

- Sva jezgra procesora su ista tj. ravnopravna.
- SMP je jedna od tehnika paralelizacije.
- Naziva se još i ***multiprocesor sa deljenom memorijom***.
- Procesori komuniciraju preko deljene memorije

Flynn-ova taksonomija

1. Single Instruction Single Data (SISD)
2. Single Instruction Multiple Data (SIMD)
3. Multiple Instructions Single Data (MISD)
4. Multiple Instructions Multiple Data (MIMD)

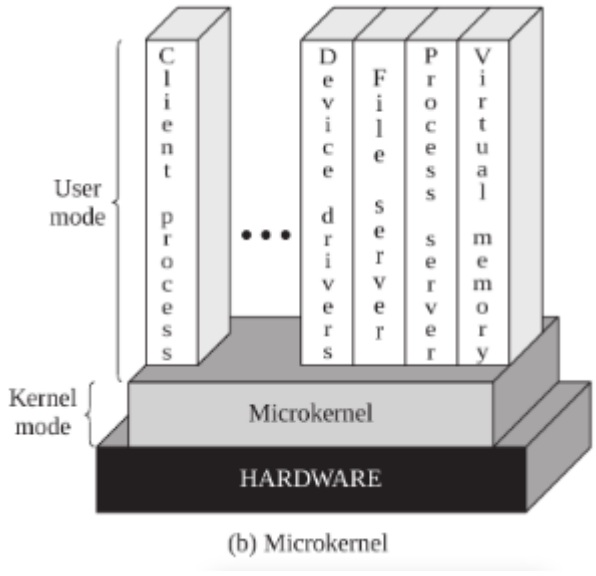
Arhitektura paralelnih procesora



Na savremenim procesorima postoji bar jedan nivo keš memorije kao provatni keš procesora. Ovo može dovesti do problema ***koherentnosti keša*** koji se obično rešava hardverski.

Pr. Procesor B uzima vrednost sa neke adrese iz glavne memorije i dovlači je u svoj keš. Istu tu adresu i vrednost dovlači i procesor A kod sebe. Ako procesor B promeni donešenu vrednosti i upiše je nazad u memoriju, procesor A neće imati validnu vrednost tj. dolazi do ***koherentnosti keša***. Ovaj problem se rešava hardverski tako što čim neki procesor spusti blok keša na sistemsku magistralu, svi ostali keševi koji sadrže promenjeni podatak se poništavaju i dovlače novi blok sa izmenjenim podatkom.

27. Koncept mikrokernela



(b) Microkernel

Mikrokernel služi kao komunikaciona linija između različitih delova OS-a. Ukoliko se menja neki od modula, nije potrebno ponovno kompajliranje celog kernela već je moguće samo zameniti željeni modul

Prednosti mikrokernela:

- Uniformni interfejsi – nema razlike između zahteva procesa i korisničkog i kernel nivoa
- Proširivost
- Fleksibilnost
- Prenosivost
- Pouzdanost
- Podrška distribuiranim sistemima
- Podrška za objektno orijentisane tehnologije

V. Konkurentna obrada

28. Ključni pojmovi konkurentnosti

- Multiprogramiranje – upravljanje sa vše procesa u jednoprocesorskom sistemu
- Multiprocesiranje – upravljanje sa više procesa unuta multiprocesora
- Distribuirana obrada – procesi na više računarskih sistema

Ključni pojmovi

1. **Kritična sekcija** - Deo koda koji zahteva pristup deljenim resursima i ne može se izvršiti dok je drugi proces u svojoj kritičnoj sekciji.
2. **Uzajamna blokada** (*deadlock*) - Zastoj jer svaki proces čeka da onaj drugi nešto uradi
3. **Živa blokada** (*livelock*) - Dva ili više procesa konstantno menjaju svoje stanje kao odgovor na akcije drugog procesa ne radeći ništa korisno
4. **Uzajamno isključivanje** (*mutual exclusion*) - Zahtev da nijedan proces ne može biti u svojoj kritičnoj sekciji kada je jedan proces u svojoj krit. sekciji
5. **Stanje trke** (*race condition*) - Rezultat zavisi od relativnog vremena izvršavanja
6. **Gladovanje** (*starvation*) - Raspoređivač (dispečer) uporno preskače dati proces iako se on nalazi u *Ready* stanju

29. Primeri stanja trke, uzajamno delovanje procesa, kritična sekcija

Primer stanje trke

```
void echo()
{
  chin = getchar();
  chout = chin;
  putchar(chout);
}
```

```
Process P1
•
chin = getchar();
•
chout = chin;
putchar(chout);
•
•
```

```
Process P2
•
•
chin = getchar();
chout = chin;
•
putchar(chout);
•
```

Rezultat ovog koda će biti vrednost karaktera koja se unese kad se izvršava proces P2 jer je chin globalna promenljiva i funkcija echo je dostupna svima. Ovaj problem može da se reši tako što će se funckija echo proglasiti kritičnim regionom. Kritičnom regionu može da pristupa samo jedan proces u jednom trenutku.

Uzajamno delovanje procesa

1. **Procesi nisu svesni drugih procesa** - Postoji odnos nadmetanja. Mogu se javiti problemi uzajamnog isključivanja, uzajamne blokade i gladovanja (*stravation*).
2. **Procesi su indirektno svesni postojanja drugih procesa (npr. deljeni objekti)** - Rezultati jednog procesa zavise od informacija koje daje neki drugi. Problemi su uzajamno isključivanje, blokada, gladovanje i povezanost podataka.
3. **Procesi su direktno svesni drugih procesa** - Procesi su projektovani tako da kooperiraju na istom zadatku. Problemi koji se mogu javiti su uzajamna blokada i gladovanje.

Načini kooperacije procesa

1. **Deljenje** - Preko deljenih promenljivih (objekata) i fajlova
2. **Putem komunikacije** - Procesi međusobno komuniciraju putem poruka

Kritična sekcija

```
do {
     entry section
         critical section
     exit section
     /* ostatak koda */
} while (1);
```

Kako se kritična sekcija može realizovati?

1. softverski
2. hardverski
3. pomoću semafora
4. pomoću OS-a (sistemskim pozivima)
5. višim programskim strukturama za sinhronizaciju (monitorima)

Mehanizam koji se bavi kritičnom sekcijom mora da zadovolji sledeće zahteve:

1) Međusobno isključenje – samo jedan proces može da se nađe u kritičnoj sekciji
2) Proces koji je izvan svoje kritčne sekcije ne može sprečiti druge procese da uđu u svoju kritičnu sekciju
3) Proces ne sme neogrničeno dugo da čeka na ulazak u svoju kritičnu sekciju
4) Proces ne sme neograničeno dugo ostati u svojoj kritičnoj sekciji

30. Algoritam striktne alternacije

Proces 0:

```
do {
    while (turn!=0);    // ulazna sekcija
    /* kriticna sekcija */
    turn = 1;
    /* ostatak koda */
} while (true);
```

Proces 1:

```
do {
    while (turn!=1);    // ulazna sekcija
    /* kriticna sekcija */
    turn = 0;
    /* ostatak koda */
} while (true);
```

Turn je globalna promenljiva koja služi kao identifikator koji će za odgovarajuću vrednost dozvoliti određenom procesu da uđe u svoju kritičnu sekciju.

while (turn!=1) se naziva zaposleno čekanje, odnosno busy wait

Nedostaci ovog algoritma su:

- Zaposleno čekanje troši procesorsko vrene
- Moguća je blokada ako neki od procesa zaglavi u ostatku koda

Što se tiče zahteva koji moraju biti zadovoljeni, zahtev 2) neće biti zadovoljen jer ukoliko se nakon kritične sekcije proces 1 zaglavi npr. čeka na scanf(), proces 0 će jednom u svoju kritičnu sekciju, ali drugi put neće jer je proces 1 ostao zaglavljen i nije imao ko da promeni vrednost promenljive turn na 0.

31. Kritična sekcija bez stroge alternacije

```
int flag[2];
flag[0] = flag[1] = 0;

do {
    flag[i]=1;
    while (flag[j]);   /ulazna sekcija
    /* kriticna sekcija */
    flag[i] = 0;
    /* ostatak koda */
} while (true);
```

flag[i] – predstavlja identifikator da li proces \eli da u]e u svoju kriti;nu sekciju ili ne

while(flag[j]); - čeka se na signal od drugog procesa koji treba da kaže da ne želi da uđe u svoju kritičnu sekciju.

Nedostatak ovog algoritma je mogućnost uzajamnog blokiranja koja procese ostavlja u beskonačnoj petlji tj. može da dođe do situacije da i jedan i drugi proces žele da uđu u svoje kritične sekcije, ali prvo moraju da sačekaju da signal od suprotnog procesa da ne želi da uđe u svoju kritičnu sekciju. Ovo je primer blokade jer će i jedan i drugi beskonačno čekati.

32. Dekker-Petersenov algoritam

Za ovaj algoritam se uvodi promenljiva **turn** i radiće samo za DVA procesa

```
do {
    flag[i] = true;
    turn = j;
    while (flag[j] && turn==j);
    /* kriticna sekcija */
    flag[i] = 0;
    /* ostatak koda */
} while (true);
```

flag[j] – proces želi da uđe u svoju kritičnu sekciju

turn=j – proces proverava da li je njegov redč ako je turn=0 onda mora da čeka drugi proces, a ako je turn=1 onda on može da uđe u svoju kritičnu sekciju

33. Pekarski algoritam

Pekarski algoritam predstavlja uopštenje Dekker-Petersenovog algoritma i ovaj algoritam će raditi za proizvoljan broj procesa. Ovaj proces radi na principu dodeljivanja brojeva procesima u rastućem poretku. Procesi se opslužuju od najmanjeg do najvećeg dodeljenog broja, a ukoliko dođe do situacije da procesi imaju iste dodeljene brojeve, onda će prvo biti uslužen proces sa manjim PID-om

Zajednički podaci za pekarski algoritam

- `int biranje[n]; // logicka promenljiva za medjusobno iskljucivanje`
- `int broj[n]; // za smestanje dodeljenih brojeva`

```
do {
    biranje[i] = 1;
    broj[i] = max (broj[0],broj[1],...,broj[n-1])+1;
    biranje[i] = 0;
    for (j=0; j<n; j++) {
        while (biranje[j]);
        while ( (broj[j]!=0) && ((broj[j],j) < (broj[i],i)) );
    }
    /* kriticna sekcija */
    broj[i] = 0;
    /* ostatak koda */
} while (true);
```

Kada proces dođe, prvo otvara ciklus biranja broja, a onda se procesu dodeljuje njegov redni broj i zatvara se ciklus dodeljivanja broja

Proces ostaje van kritične sekcije sve dok postoji proces koji je ispred njega. Kada završi, proces setuje svoj broj na 0 i on se iše ne takmiči za kritičnu sekciju.

34. Hardverska realizacija k.s. - testset() i exchange()

Instrukcija testset() je atomska, tj. izvršava se u jednom komadu i ne može se prekinuti.

```
/* program mutualexclusion */
const int n = /* number of processes */;
int bolt;
void P(int i)
{
   while (true) {
     while (!testset(bolt))
          /* do nothing */;
      /* critical section */;
      bolt = 0;
      /* remainder */;
   }
}
void main()
{
   bolt = 0;
   parbegin (P(1), P(2), ... ,P(n));

}
```

```
boolean testset(int i) {
    if (i==0) {
        i=1;
        return true;
    } else
        return false;
}
```

Promenljiva bolt je identifikator za to da li je neki proces u kritičnoj sekciji.

```
int const n = /* number of processes**/;
int bolt;
void P(int i)
{
   int keyi = 1;
   while (true) {
      do exchange (keyi, bolt)
      while (keyi != 0);
      /* critical section */;
      bolt = 0;
      /* remainder */;
   }
}
void main()
{
   bolt = 0;
   parbegin (P(1), P(2), ..., P(n));
}
```

Funkcija do excange je funkcija koja je realizovana na Intelovim procesorima. Ova funkcija menja vrednosti dve promenljive bez uvođenja treće. Bolt predsravlja bravu, a keyi ključ.

Nedostaci ovih hardversih realizacija su:

- Upotreba stanja busy wait tj. zauzet na čekanju i time bespotrebno troši procesorsko vreme
- Moguće je gladovanje – slučajno se bira proces koji će biti raspoređen na izlazu iz kritične sekcije
- Moguće je uzajamno blokranje – npr. na jednoprocesorskom sistemu sa šemom prioriteta

35. Opšti i binarni semafor

Semafori

Semafori su jednostavan mehanizam OS-a za razmenu signala između procesa. Da bi se poslao signal, izvršava se `SemSignal(s)`, a da bi se primio `SemWait(s)`.

Opšti semafor

1. Semafor se može inicijalizovati na pozitivnu vrednost
2. **semWait()** umanjuje vrednost semafora. Ako postane negativna, proces se blokira. Ako ostane pozitivna, nastavlja se.
3. **semSignalB()** uvećava vrednost semafora. Ako postane ≤ 0, vrši se deblokada procesa koji je blokiran pomoću **semWait()**

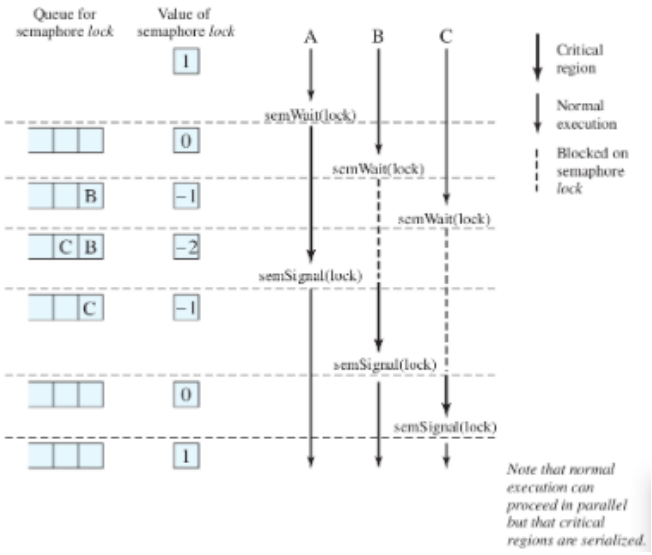
Binarni semafor (*mutex*)

1. Semafor se može inicijalizovati na 0 ili 1
2. **semWaitB()** proverava vrednost semafora. Ako je 0, blokira se proces koji je pozvao **semWaitB()**. Ako je 1, menja se na 0 i nastavlja se dalje.
3. **semSignalB()** proverava da li je neki proces na datom semaforu blokiran. Ako jeste, deblokira se. Ako nema blokiranih procesa, vrednost semafora se postavlja na 1.

```
struct semaphore {
      int count;
      queueType queue;
};
void semWait(semaphore s)
{
      s.count--;
      if (s.count < 0) {
            /* place this process in s.queue */;
            /* block this process */;
      }
}
void semSignal(semaphore s)
{
      s.count++;
      if (s.count <= 0) {
            /* remove a process P from s.queue */;
            /* place process P on ready list */;
      }
}
```

```
struct binary_semaphore {
      enum {zero, one} value;
      queueType queue;
};
void semWaitB(binary_semaphore s)
{
      if (s.value == one)
            s.value = zero;
      else {
                  /* place this process in s.queue */;
                  /* block this process */;
      }
}
void semSignalB(semaphore s)
{
      if (s.queue is empty())
            s.value = one;
      else {
                  /* remove a process P from s.queue */;
                  /* place process P on ready list */;
      }
}
```

```
/* program mutualexclusion */
const int n = /* number of processes  */;
semaphore s = 1;
void P(int i)
{
      while (true) {
            semWait(s);
            /* critical section   */;
            semSignal(s);
            /* remainder   */;
      }
}
void main()
{
      parbegin (P(1), P(2), . . ., P(n));
}
```



Slike iznad pokazuju na koji način može da se izvrši međusobno isključenje dva procesa.

Objašnjenje druge slike: Proces A zatiče semafor sa vrednošću 0 pa može da uđe u kritičan region. Dok se on izvršava i nalazi se u kritičnom regionu, do kritičnog regiona stiže i proces B, ali on ne može da uđe jer je u kritičnom regionu proces A, pa se dodaje na red blokiranih procesa, a vrednost semafora postaje -1. U međuvremenu, isto se dešava i sa procesom C, pa se i on dodaje na red blokiranih procesa, a vrednost semafora je -2. Proces A je završio svoj posao u kritičnom regionu, pa šalje semsignal() kojipovećava vrednost semafora na -1 i time je odblokiran proces B. Kad proces B izađe iz kritičnog regiona, onda u kritični region ulazi proces C. Na kraju je vrednost semafora 0, a red blokiranih procesa je prazan.

36. Problem proizvođač/potrošač sa neograničenim baferom

Prvo, da bi došlo do konzumiranja potrebno je da in>out. Problem do kog može doći je ako je bafer prazan i tada proces potrošač mora da se blokira, a proizvođač čim doda element na bafer, mora da odblokira potrošača ukoliko je bio blokiran i mora se obezbediti da baferu može pristupati samo jedan proces u jednom trenutku.

Ovde se javlja dodatan problem, ako dođe do situacije da vrednost semafora bude -1, zato se uvodi dodatna promenljiva koja će uzeti vrednost n pre kritičnog regiona.

37. Problem proizvođač/potrošač sa ograničenim kružnim baferom

Proizvođač ne sme da dodaje u bafer ako je bafer pun, a potrošač ne sme da uzima iz praznog bafera.

38. Načini implementacije semafora

```
semWait(s)
{
   while (!testset(s.flag))
      /* do nothing */;
   s.count--;
   if (s.count < 0) {
      /* place this process in s.queue*/;
      /* block this process (must also set
s.flag to 0) */;
   }
   s.flag = 0;
}

semSignal(s)
{
   while (!testset(s.flag))
      /* do nothing */;
   s.count++;
   if (s.count <= 0) {
      /* remove a process P from s.queue */;
      /* place process P on ready list */;
   }
   s.flag = 0;
}
```

```
semWait(s)
{
   inhibit interrupts;
   s.count--;
   if (s.count < 0) {
      /* place this process in s.queue */;
      /* block this process and allow inter-
rupts */;
   }
   else
      allow interrupts;
}

semSignal(s)
{
   inhibit interrupts;
   s.count++;
   if (s.count <= 0) {
      /* remove a process P from s.queue */;
      /* place process P on ready list */;
   }
   allow interrupts;
}
```

Levo je implementacija semafora pomoću funkcije testset(), a desno onemogućavanjem prekida

Slabosti testset() instrukcije su busy wait, a slabosti onemogućavanjem prekida je u tome što usporava performanse.

39. Monitori

Šta su to monitori?

Semafori predstavljaju primitivan sistem kontrole uzajamnog isključivanja i sinhronizacije. Zašto? U nekim programskim jezicima postoje strukture koje se zovu monitorima koje rešavaju problem kapsulacijom funkcionalnosti. **Sastoji se iz procedura, init sekvence i lokalnih podataka.**

Osobine monitora

1. Lokalnim promenljivama se može pristupiti isključivo preko procedura monitora.
2. Proces ulazi u monitor pozivanjem jedne od svojih procedura.
3. U monitoru se istovremeno može izvršavati samo jedan proces.

40. Prosleđivanje poruka

Prednost poruka u odnosu na semafore i monitora je u tome što slanje poruka može da radi i između računara. Ima dve primitive: send(oderediste, poruka) i receive (izvor, poruka)

Karakteristike

Sinhronizacija
- Slanje
  - blokirano
  - neblokirano
- Primanje
  - blokirano
  - neblokirano
  - ispitivanje prijema

Format
- Sadržaj
- Dužina
  - fiksna
  - promenljiva

Adresiranje
- Direktno
  - slanje
  - primanje
    - eksplicitno
    - implicitno
- Indirektno
  - statičko
  - dinamičko
  - vlasništvo

Disciplina redova
- FIFO
- po prioritetu

41. Problem čitaoci/pisci, rešenje preko semafora (kod sa druge slike iznad)

```
/* program readersandwriters */
int readcount;
semaphore x = 1,wsem = 1;
void reader()
{
    while (true){
     semWait (x);
     readcount++;
     if(readcount == 1)
         semWait (wsem);
     semSignal (x);
     READUNIT();
     semWait (x);
     readcount;
     if(readcount == 0)
         semSignal (wsem);
     semSignal (x);
    }
}
```

```
void writer()
{
    while (true){
     semWait (wsem);
     WRITEUNIT();
     semSignal (wsem);
    }
}

void main()
{
    readcount = 0;
    parbegin (reader,writer);
}
```

Rešenje: Pisac je regulisan normalno, uvođenjem semafora koji dozvoljava da samo jedan pisac bude u kritičnom regionu.

Semafor x obezbeđuje kritični region za promenljivu readcount. Ako je readcount>1, tada mogu da uđu i ostali čitaoci da čitaju.

Semafor wsem služi da se signalizira piscu da može da uđe da promeni dokument.

U ovom rešenju se javlja problem prioriteta jer ne znamo kome treba da damo prioritet, piscu ili čitaocu? U ovom rešenju, prioritet imaju čitaoci, pa će pisac morati da čeka da svi čitaoci da završe da bi on mogao da menja dokument.

Rešenje kada pisci imaju prednost

Problem prethodnog rešenja je u tome što dok god ima i jedog čitaoca, pisac neće moći da pristupi. Može doći do **gladovanja**.

Novi semafori

1. Semafor **rsem** sprečava sve čitaoce dok god ima i jednog pisca koji pristupa dokumentu
2. Promenljiva **writecount** kontroliše postavljanje **rsem**
3. Semafor **y** kontroliše ažuriranje promenljive **writecount**
4. Semafor **z** je potreban da se ne bi stvorio dug red čitalaca ispred **rsem**. Samo jedan čitalac sme da čeka ispred **rsem**, dok svi ostali čekaju ispred **z**.

```
/* program readersandwriters */
int readcount,writecount;
semaphore x = 1, y = 1, z = 1, wsem = 1, rsem = 1;
void reader()
{
    while (true){
     semWait (z);
          semWait (rsem);
               semWait (x);
                    readcount++;
                    if (readcount == 1)
                         semWait (wsem);
                    semSignal (x);
               semSignal (rsem);
          semSignal (z);
          READUNIT();
          semWait (x);
               readcount--;
               if (readcount == 0) semSignal (wsem);
          semSignal (x);
    }
}
```

```
void writer ()
{
    while (true){
          semWait (y);
               writecount++;
               if (writecount == 1)
                    semWait (rsem);
          semSignal (y);
          semWait (wsem);
          WRITEUNIT();
          semSignal (wsem);
          semWait (y);
               writecount;
               if (writecount == 0) semSignal (rsem);
          semSignal (y);
    }
}
void main()
{
    readcount = writecount = 0;
    parbegin (reader, writer);
}
```

Diskusija o rešenju: Svaki novi pisac koji dođe da menja dokument, moraće da čeka na semaforu wsem. Svaki novi čitalac može da dođe da čita. Ako dođe pisac, on zatvara semafor rsem, pa će prvi novi čitalac da čeka upravo na ovom semaforu dok će ostali novopristigi da čekaju na semaforu z, a pisac će da sačeka na semaforu wsem dok ne završe čitaoci koji su se zatekli kada je naišao pisac.

42. Problem čitaoci/pisci, rešenje preko poruka

```
void reader(int i)
{
   message rmsg;
      while (true) {
         rmsg = i;
         send (readrequest, rmsg);
         receive (mbox[i], rmsg);
         READUNIT ();
         rmsg = i;
         send (finished, rmsg);
      }
}
void writer(int j)
{
   message rmsg;
   while(true) {
      rmsg = j;
      send (writerequest, rmsg);
      receive (mbox[j], rmsg);
      WRITEUNIT ();
      rmsg = j;
      send (finished, rmsg);
   }
}
```

```
void  controller()
{
      while (true)
      {
         if (count > 0) {
            if (!empty (finished)) {
               receive (finished, msg);
               count++;
            }
            else if (!empty (writerequest)) {
               receive  (writerequest,  msg);
               writer_id = msg.id;
               count = count - 100;
            }
            else if (!empty (readrequest)) {
               receive (readrequest, msg);
               count--;
               send (msg.id, "OK");
            }
         }
         if (count == 0) {
            send (writer_id, "OK");
            receive (finished, msg);
            count = 100;
         }
         while (count < 0) {
            receive (finished, msg);
            count++;
         }
      }
}
```

Diskusija o rešenju:

Rešenje se sastoji iz tri vrste komponenti: čitalaca, pisca i jedinstvenog kontrolera (count).

Ako je count>0, onda nema pisaca koji čekaju.

Ako je count=0, onda pisac čeka i treba mu dozvoliti pristup i sačekati poruku finished.

Ako je count<0, onda je pisac napravio zahtev, ali mora da čeka da završe aktivni čitaoci. Iz tog razloga, primaju se samo finished poruke.

Čitalac se obraća kontroleru, šalje mu zahtev i čeka odgovor kontrolera. Kada dobije poruku, on kreće da čita. Pisac isto šalje zahtev kontroleru i

čeka odgovor. Kontroler vodi računa o promenljivoj count koju smanjuje i povećava u zavisnosti od toga od koga je dobio zahtev. Ukoliko je COUNT=100 i zahtev šalje čitalac, COUNT će da se smanji za 1, ali ako zahtev pošalje pisac, ona će da se smanji za 100, odobriće zahtev piscu, čekaće da pisac završi i ponovo će povećati promenljivu COUNT na 100.

VI. Uzajamna blokada i gladovanje

43. Definicija uzajamne blokade, joint progress dijagrami
44. Blokada kod potrošnih resursa i resursa koji se mogu ponovo koristiti
45. Grafovi alokacije resursa, uslovi za nastanak blokade
46. Sprečavanje blokade
47. Izbegavanje blokade - odbijanje pokretanja procesa
48. Izbegavanje blokade - bankarski algoritam
49. Otkrivanje blokade, oporavak od blokade
50. Problem filozofa na večeri

VII. Upravljanje memorijom

51. Zahtevi od sistema za upravljanje memorijom
52. Fiksno particionisanje
53. Dinamičko particionisanje
54. Partnerski sistem dodele memorije
55. Relokacija, prosto straničenje

VIII. Virtuelna memorija – hardverski mehanizmi

56. Prosta segmentacija
57. Koncept virtuelne memorije i princip lokalnosti
58. Tabela stranica, šema u 2 nivoa
59. Obrunuta tabela stranica
60. TLB bafer
61. Uticaj veličine stranice na performanse virt. memorije

IX. Virtuelna memorija – mehanizmi OS-a

62. Politika donošenja (*fetch*), politika smeštanja, koncepti politika zamene
63. Osnovni algoritmi zamene
64. Performanse algoritama zamene i baferovanje stranica
65. Upravljanje rezidentnim skupom
66. Politika čišćenja, nivo multiprogramiranja i suspenzija procesa

Linux komande:

5) ls – pokazuje fajlove i direktorijume, veličinu fajla/foldera, datum modifikacije, vlasnika fajla i njegove permisije
6) ssh – kačenje na neki drugi računar
7) pstree – stablo procesa
8) lsof – lista fajlova koji su otvoreni od strane procesa
9) ps aux – PID, UID, PPID procesa