OPERATIVNI SISTEMI 1

1. Šta je operativni sistem?

- Operativni sistem je najvažniji deo sistemskih programa čiji je zadatak **upravljanje resursima računara** i koji obezbeđuje **osnovu** za pisanje aplikacionih programa.
- Direktno programiranje određenih delova računara je veoma težak posao pa je zbog toga došlo do ideje da se stavi jedan sloj između aplikacionih programa i hardvera.
-Ideja tog sloja je da se obezbedi neki interfejs(ili virtuelna mašina) ostalim programima radi lakšeg pristupa hardveru, taj sloj je OS.

2. Na koji način se može podeliti softver? Gde tu spada OS.

- Sistemski programi – upravljaju računarom (ovo je mesto za os)
- Korisnički(aplikacioni) programi – rešavaju problem korisnika

3. Kako izgledaju programi po nivoima?

<table>
<tr><td>Office</td><td>Baze podataka</td><td>Igre</td><td>Korisnički programi</td></tr>
<tr><td>Kompajleri, iterpreteri</td><td>Editori</td><td>Linkeri</td><td rowspan="2">Sistemski programi</td></tr>
<tr><td colspan="3">Operativni sistem</td></tr>
<tr><td colspan="3">Mašinski jezik</td><td rowspan="3">Hardver</td></tr>
<tr><td colspan="3">Mikro programi</td></tr>
<tr><td colspan="3">Fizički uređaji</td></tr>
</table>

Mašinski jezik: skup instrukcija koje procesor direktno razume (izvršavaju se pomoću svojih mikro programa)
Mikro programi: direktno kontrolišu uređaje predstavljaju elementarne korake od kojih se sastoje instrukcije mašinskog jezika, obezbeđuje interfejs prema sledećem nivou.
Fizički sloj: najniži sloj – fizički delovi računara.

4. Koje su dve definicije OS i koja je njegova funkcija u njima?

1. **OS kao proširena(extended) ili virtuelna (virtual) mašina:**

-Arhitektura računara na nivou mašinskog jezika je primitivna i nije pogodna za programiranje.

-Primer: kontroler za **disketni uređaj**(NED PD765) koji se koristi na personalnim računarima najosnovnije komande su mu READ i WRITE. Svaki put kad želimo da ga pišemo ili čitamo sa diskete moramo voditi računa o mnogim stvarima(da li je motor uključen naći stazu, sektor..), što otežava proces.

- Zato je zadatak OS kao proširene ili virtuelne mašine da te stvari radi umesto nas, da nam pruža neke funkcije višeg nivoa apstrakcije radi pristupa hardveru.

2. **OS kao upravljač resursima (resource manager):**

- Resurs obuhvata sve što je programu potrebno za rad.
- Kao upravljač resursima OS ima zadatak da vodi računa o njima u smislu da zadovolji potrebe programa, da prati koji program koristi koje resurse…

Primer – dva korisnika istovremeno žele nešto da štampaju, OS je dužan da vodi računa o tome da programi tih korisnika dođu do štampača i da se podaci ne mešaju.

5. Zadaci operativnog sistema

- **Glavna funkcija** je sakrivanje detalja nižih nivoa od ostalih programa i pružanje niza jednostavnih instrukcija za pristup hardveru.
- OS kao proširena (virtuelna) mašina – zadatak je da pruži neke funkcije višeg nivoa apstrakcije radi pristupa hardveru.

OS – kao upravljač resursima ima zadatak da vodi računa o resursima računara, zadovoljava potrebe programa, prati koji program koristi koji resurs.

6. Istorija operativnih sistema

- **Generacija I (1954 – 1955) –** spori, veliki i skupi računari
  - pravljeni od **vakuumskih cevi**(čak do 20000 u jednom računaru)
  - koristila ih je vojska
  - programiranje na mašinskom jeziku (programski jezici uključujući i asembler kao i os su nepoznati)
  - radnici na računarima rade sve, od programiranja do održavanja.

- **Generacija II(1955 – 1965) –** manji, pouzdaniji i jeftiniji
  - pravljeni od tranzistora
  - pošto su jeftiniji kupuju ih i univerziteti i velike korporacije
  - Računari su bili odvojeni u sobama
  - Fortran programi pisani na papiru su se prenosili na bušene kartice i ostavljane u Input sobi, operator kupi kartice i ubacuje ih nakon kartica sa fortran kompajlerom, rezultat (takođe na karticama) se nosi u output sobu.
  - Mnogo vremena troši na šetanje između raznih prostorija
  - **OS j**oš uvek **ne postoji**.
  - **Uvodi se poboljšanje**: pokretna oobrada (**batch system**)- skuplja se količina **sličnih poslova** npr isti kompajler, koji se pomoću jeftinijeg računara(npr IBM 1401) prenose na **magnetnu traku**, traka ide na glavni računar (npr IBM 7049) koji je moćniji i skuplji, on učitava posebni program zadužen da **redom** učitava programe sa trake sa poslovima i izvršava ih (**preteča OS-a**). Nakon čega se rezultat snimi na drugu traku.
  - Kada su svi poslovi odrađeni, operator stavlja drugu traku sa programima a **traku sa rezultatima** prenosi do **jeftinijeg računara** radi **prebacivanja** rezultata na **bušene kartice**.
  - Manji računari nisu **vezani za glavni računar**, rade offline

- **Generacija III(1965 - 1980)**
  - računari se prave od **integrisanih kola**(ic)
  - Početkom 60-ih uglavnom se proizvode **jača i slabija** vrsta računara što je skupo jer su međusobno **nekompatibilni** (težak prelazak sa slabijeg na jači).
  - IBM uvodi seriju **kompatibilnih račnara** različitih snaga **System360** čime nestaje podela na jače i slabije, a svi one rade pod **OS/360** koji je bio jako glomazan i **pun grešaka**.
  - Dolazi do razvoja softverskog inženjerstva.
  - **Novine:**
    - 1. Kada program čeka na rezultate IO operacija, procesor je neiskorišćen pa se javlja **gubljenje procesorskog vremena**(problem kod poslovnih programa)
      - Rešenje koje se javlja je **multiprogramiranje** – memorija se deli na **particije** u kojima se učitavaju različiti programi i dok jedan čeka na IO, procesor izvršava drugi
    - 2**. SPOOL(Simultaneous Peripheral Operation On Line)** – prebacivanje sadržaja bušenih kartica na disk(traku) pomoću posebnog uređaja bez CPU-a što znači da CPU izvršava program u memoriji dok se paralelno disk puni novim programima, čim se program izvrši CPU preuzima drugi sa diska. Isto se koristi i za izlazne podatke.

- 3. **Podela vremena(time sharing) –** Kod prve generacije programer je pisao program, ubacivao u računar i kupio rezultate bez čekanja na obradu drugih poslova, dok se kod pokretne obrade na rezultat čeka i po nekoliko sati što je problem ukoliko ima grešaka u kodu – **rešenje je podela vremena**, što podrazumeva da svaki korisnik ima terminal zakačen na glavni računar(oblik multiprogramiranja), gde im se ciklusi dodeljuju na osnovu dva kriterijuma: **čekanje na IO operacije** i **isteka dodeljenog vremena** (pojam quantum- a – količine vremena nakon čijeg isteka kontrola se predaje drugom programu)
  - (multiprograming + quantum = time-sharing)
- 4. **MULTICS**(Mulltiplexed Information and computing service) – neuspela ideja da se napravi moćan računar sa velikim brojem terminala dostupnih građanima(po uzoru na električnu mrežu), po ideji je davni predak mreža i interneta
- 5. **Miniračunari** : prvi miniračunar DEC-ov (Digital equipment corporation)(PDP-1), najmanji i najeftiniji do tada 120000 dolara
- 6. **UNIX :** Ken Thompson, jedan od naučnika Firme Bell Labs, koji je radio na projektu multics, uzeo je jedan PDP – 7 miniračunar i za njega napisao mini verziju multicsa, od tog projekta je posle nastao UNIX(UNI – jedan, X-CS- Computing service)

- **Generacija IV(1980 – 1990)**
  - **Personalni računari -** razvoj njihov je počeo pojavom LSI čipova(Large scale integration).
  - Dovoljno jeftini da ih poseduju pojedinci(Spectrum, Commodore, Atari, IBM)
  - Pojava prvih pravih OS-a: MS – DOS, UNIX
  - Razvoj korisničkih interfejsa i grafičkog okruženja
  - Nove vrste OS-a(i računara):
    - 1. **Mrežni OS(NOS): Umreženi računari** sa sopstvenim(moguće i različitim) OS-ovima – koji komuniciraju zahvaljujući zajedničkom protokoli, računari u mreži su svesni jedni drugih, mreže mogu biti **lokalne i globalne**.
    - 2. **Distribuirani OS(DOS):** Korisnik ga vidi kao jednoprocesorski sistem iako OS kontroliše više CPU-ova i upravlja svim resursima u mreži. Korisnik ne vodi računa o tome gde su fajlovi ili gde se izvršava program – o tome vodi računa sam OS, koji takođe sakriva činjenicu da postoji mreža računara te tako korisniku izgleda kao da koristi jedan uređaj.
    - 3. **Personalni računari : grafički korisnički interfejs**

- **Generacija V**: neuspela ideja – PROLOG kao mašinski jezik, računari sa veštačkom inteligencijom, još uvek aktuelna generacija **IV**.

7. Koja generacija OS-a je bila najznačajnija i zašto?
- Treća generacija zbog pojave IBM **OS/360,** razvoja softverskog inženjerstva, **timesharing, multiprogramiranja i pojava mini računara.**

8. Objasniti time sharing ima u trecoj generaciji.

9. Arhitektura operativnih sistema

- **1. Monolitni sistemi**(„The big mess“)
  - OS je realizovan kao skup procedura, od kojih svaka može pozvati svaku ako je to potrebno
  - Korisnički programi servisa OS-a koriste se na sledeći način:
    - **parametri sistemskog poziva** se smeštaju na određena mesta(registri stek)
    - potome sledi **kernel call** (poziv jezgra OS-a) koji **prebacuje procesor iz korisničkog režima rada u sistemski režim i kontrolu prebacuje na OS.**(razlika je u tome što u korisničkom režimu nisu dozvoljene neke komande procesora a u sistemskom se mogu koristiti sve operacije)

        - OS preuzima kontrolu, na osnovu parametara određuje koju sistemsku proceduru treba pozvati, poziva je pa vraća kontrolu korisničkom programu.
    - OS ima strukturu(3 sloja):
        - 1. **Glavni program** koji obezbeđuje **sistemske pozive**
        - 2. Skup sistemskih procedura koje se pozivaju prilikom sistemskih poziva
        - 3. skup pomoćnih procedura koje se koriste od strane sistemski procedura

- **2. Slojeviti sistemi**
    - Prvi OS sa ovakvom strukturom je bio THE koji se sastojao od 6 slojeva
        - 5. **komandni interpreter**
        - 4. korisnički programi
        - 3* ulazne-izlazne operacije
        - 2*. procesi – komunikacija između procesa i komandnog interpretera
        - 1*. Upravljanje memorijom – zauzima potrebnu memoriju za procese
        - 0*. procesor i multiprogramiranje – obezbeđuje prebacivanje između procesa
    - *jezgro os-a, rade u sistemskom režimu

-Bitna razlika između monolitne i slojevite realizacije je u tome što se os kod monolitne strukture sastoji od **skupa procedura** bez grupisanja ili hijerarhije, a kod slojevite ralizacija os se deli na više slojeva od kojih se svaki oslanja na slojeve ispod i gde svaki sloji **ima tačno određenu funkciju**.

- Kod MULTICS-a umesto slojeva se koriste **koncentrični prstenovi**(procedura iz spoljašnjeg prstena zove srvis iz unutrašnjeg kao sistemski pozivi)

- **3. Virtuelne mašine:**
    - **Primer IBM VM/370**

Aplikacija 1	Aplikacija 2	Aplikacija 3
OS1	OS2	OS3
Monitor virtuelnih mašina		
Hardver		

- **4.Klijent server model:**
    - Procese delimo na dve vrste: **klijent proces(procesi korisnika)** i **server procesi(obrada klijentskih procesa),** oba rade u korisničkom režimu.
    - Kernel, jezgro je zaduženo za komunikaciju te dve vrste
    - Server procesi su delovi OS-a i grupisani su u zavisnosti od f-je koju obavljaju, npr proces serveri, fajl serveri, serveri memorije. Svi ti **serveri** rade u **korisničkom režimu** pa nemaju direktan pristup hardveru.
    - **Prednosti:**
        - OS je razbijen na manje delove(servere) koji se održavaju nezavisno.
        - u slučaju greške u nekom serveru ne pada nužno ceo sistem.
        - pogodan je za pravljenje distribuiranog sistema (različiti serveri mogu biti na različitim računarima)
    - **Problem:** Neke funkcije se teško ili nikako ne mogu realizovati u korisničkom režimu rada
        - Dva rešenje:
            - 1. **Postoje kritični server procesi** koji se izvršavaju u sistemskom režimu rada i imaju direktan pristup hardveru, ali sa ostalim procesima komuniciraju kao što se radi u ovom modelu
            - 2. Neke kritične mehanizme ugrađujemo u sam kernel, a pravimo server u korisničkom režimu koji sadrži samo interfejs tih operacija.

10. Navesti tri arhitekture OS

monolitna, slojevita, VM, klijent server

11. Koji je jedan od najbitnijih OS-ova koji je uticao na softversko inženjerstvo – IBM os/360

12. Prva virtuelna mašina – IBM VM/370

13. Šta su sistemski pozivi?

- To su niz proširenih instrukcija preko koga se obezbeđuje lakši pristup hardveru.

14. Šta je program a šta proces?

Program: Niz instrukcija koje realizuju neki algoritam.

Proces: je program u statusu izvršavanja, zajedno sa svim resursima potrebnim za rad programa

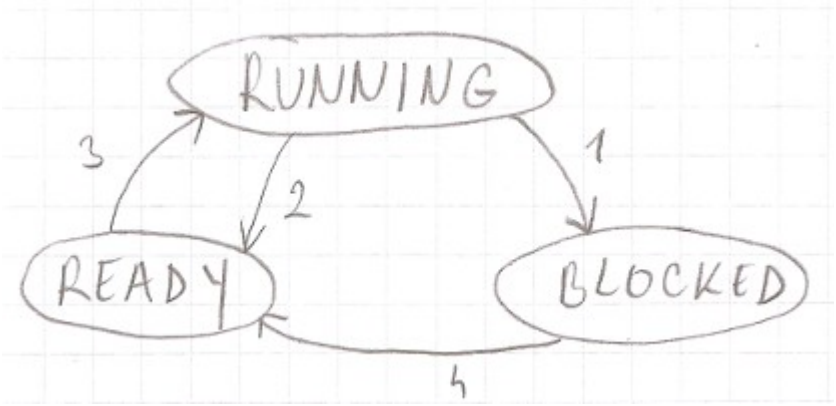
Dakle program je fajl na disku i kada se taj fajl učita u memoriju i počinje da se izvršava dobijamo proces.

15. Navesti moguća stanja procesa, strelicama predstaviti stanja i detaljnije objasniti jedan prelazak

Procesi se nalaze u jednom od sledećih stanja

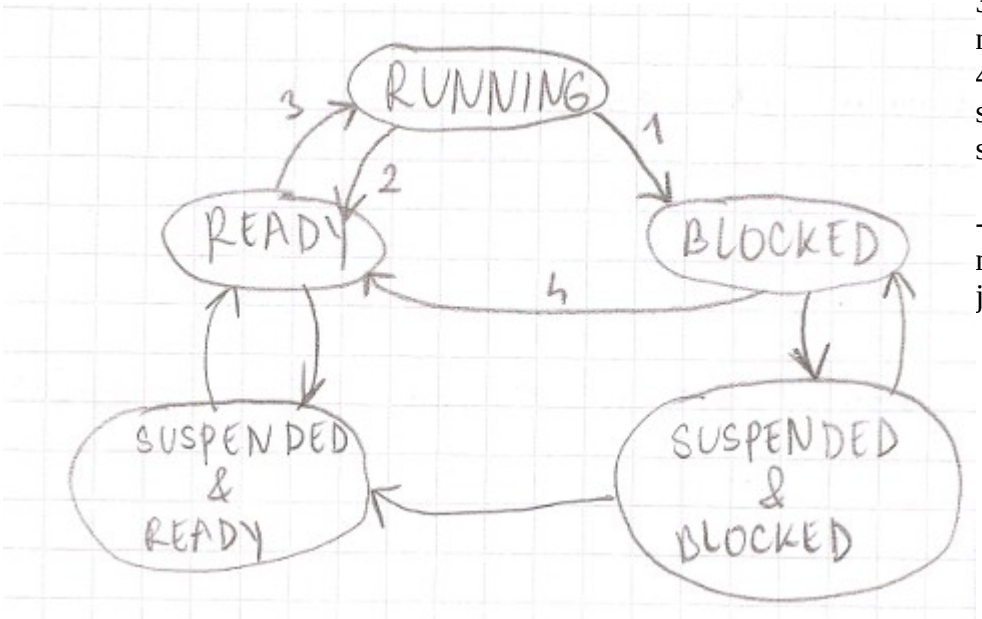
- **RUNNING**(proces se izvršava) – procesor upravo izvršava kod procesa
- **READY(**proces spreman ali se ne izvršava**) –** proces dobio sve potrebne resurse i spreman je za izvršavanje, čeka procesora.
- **BLOCKED (**proces je blokiran, čeka nešto**) –** za dalji rad procesa potrebni su neki resursi koji nisu dostupni npr. čeka da štampač zavrsši štampanje ili rezultat drugog procesa.

Imamo 4 prelaska između stanja



1. procesu potrebni neki resursi koji još nisu dostupni, proces šalje zahtev za resurs i sam menja stanje u blocked dok čeka resurs

2. Proces prelazi u stanje ready ako mu istekne **dodeljeno procesorsko vreme** (time sharing)- tada proces prelazi u listu procesa koji čekaju na procesor.



3. kada procesor oslobodi izabere se neki roces iz liste čekanja i izvršava se
4. proces dodje do potrebnih resursa i spreman je ali procesor trenutno nije slobodan pa prelazi u listu čekanja

- Kod neih operativnih sistema procesi mogu biti **suspendovani** pa se dodaju još dva stanja

- Proces koji je suspendovan prestaje da se takmiči za resurse, oslobađaju se resursi koje je zauzeo al ostaje još uvek proces.

- Iz stanja SUSPENDED & BLOCKED u stanje BLOCKED i iz SUSPENDED& READY u READY procesi mogu preći samo ako su **eksplicitno pozvani(zahtevom korisnika)**

- Iz READY u SUS.READY proces prelazi i nekoliko razloga:
  - prevelik broj ready procesi – procesi se suspenduju da se ne bi preopteretio sistem
  - eksplicitno suspendovanje od strane korisnika
  - izbegavanje zaglavljivanja(**dead lock**) – do zaglavljivanja se dolazi kada dva ili više procesa blokiraju jedan drugi u izvršavanju(npr procesu P1 treba resurs A koji je kod proces P2, a procesu P2 treba resurs B koji drži P1 ovi procesi su se zaglavili jer ni jedan od njih se ne može nastaviti, ovom slučaju jedan od proces se suspenduje pa drugi može da odradi svoj zadatak, pa kada se resursi oslobode prvi će moći da završi svoj rad)

16. Objasniti PCB(Process Control Block)
- Ako imamo 2 proces P1 iP2. P1 se izvršava dok ne dođe do blokiranja jer čeka da se desi neki događaj, tada krećemo da izvršavamo P2 koji nakon nekog vremena takođe postaje blokiran, u međuvremenu se desio događaj na koji je čekao P1 i treba da nastavimo sa njegovim izvršavanjem.
- Da bismo znali gde treba nastaviti potrebno je da se pamte neke informacije o procesu i tome služi PCB
- Svakom procesu se dodeljuje jedinstveni PCB koji sadrži sledeće informacije:
  - 1. Jedinstvedni iindetifikator pocesa **PID – process ID**
  - 2. Stanje procesa
  - 3. Prioritet procesa. Iz liste čekanja biramo proces sa najvećim prioritetom
  - 4. adresa memorije gde se nalazi proces
  - 5.adresa zauzetog resursa
  - 6. sadržaj registara procesa…
- PCB – ovi svih procesa u memoriji se smeštaju u neki niz ili listu

17. **Koje su neke operacije koje možemo vršiti nad procesima pomoću OS-a?**
- Kreiranje novog procesa(kreiranjem PCB)
- uništavanje procesa
- menjanje stanja procesa
- menjanje prioriteta procesa
- izbor procesa za izvršavanje (**context switch -** dodela procesora nekom procesu)
- sinhronizacija procesa
- komunikacija između procesa…

- Sami procesi mogu kreirati nove procese, roditelj i dete, dve vrste:
1. proces- roditelj kreira novi proces dete i čeka da ono završi sa radom
2. proces- roditelj kreira proces dete i oni rade paralelno

18. Kako delimo OS u odnosu na proces? A kako u odnosu na broj korisnika?
- Operativni sistemi mogu podržavati:
  - **monotasking (**jednoprocesni, monoprogramiranje**) –** u memoriji se izvršava jedan proces (npr. DOS)
  - **multitasking** (višeprocesni, multiprogramiranje) – u memoriji se izvršava više procesa istovremeno (npr. Windows, Linux)

- U odnosu na broj korisnika:
  - **monouser**(jednokorisnički)(DOS, Windoes 98)

- **multiuser**(višekorisnički) (UNIX-višekorisnički višeprocesni)

19. Koja je razlika između teških i lakih procesa?

-Za izvršavanje procesa potrebno je 4 dela memorije:

HEAP
STACK
Globalne promenljive
Kod procesa

HEAP – deo memorije rezervisan za dinamičke promenljive(one koje se stvaraju u toku izvršavanja programa, **ALLOCATE** -zauzmi memoriju i **DEALLOCATE –** oslobodi memoriju. Promenljivama pristupamo pomoću pokazivača. Garbage , garbage collector – briše promenljive na koje ne pokazuje ni jedan pokazivač)

STACK – deo memorije rezervisan za čuvanje lokalnih promenljivih, povratnih adresa, parametara, procedura

- Podela na lake i teške procese se vrši na osnovu toga kako koriste ove delove memorije:

-Svaki **TEŠKI** proces ima sopstveni memorijski prostor za kod, globalne promenljive, stack i heap koju **ne deli** ni sakim, pristup tim delovima ima samo dati proces.

- LAKI procesi(niti, threads) – **mogu deliti** memorijski prostor za globalne promenljive, kod i heap. **STEK se ne može deliti** jer ne možemo unapred znati koji proces koristi stek

20. Koji su problemi lakih i teških procesa?

-Problem **teških** procesa je komunikacija jer ne dele ni jedan deo memorije, tj odvojeni su, zato OS mora da obezbedi neke **mehanizme za međuprocesnu komunikaciju**(interpocess communication)

- **Laki** procesi nemaju problema sa komunikacijom jer dele određene delove memorije, ali kod njih se javlja problem **sinhronizacije.**

21. Sekvencijalno i konkurentno programiranje:

-**Sekvencijalni** program predstavlja tačno jedan proces koji se izvršava od početka do kraja u određenom redosledu na jednom procesoru.

-**Konkurentni** program se sastoji iz više **nezavisnih** procesa ili zadataka-tasks koji se mogu istovremeno izvršavati. Stil konkurentnog programiranja možemo koristiti kada se zadatak može razbiti na više međusobno relatino nezavisnih delova, procesa, koji se mogu istovremeno izvršavati. **Konkurentni programi se mogu izvršavati i na računaru sa jednim procesorom i sa više.**

22. Šta su sistemski pozivi?

-**Aplikacioni(korisnički)** programi komuniciraju sa OS-om pomoću sistemskih poziva koji se realizuju pomoću **sistema prekida**:

- korisnički program postavlja parametre sistemskog poziva na određene memorijske lokacije ili registre procesa
- Onda OS preuzima kontrolu, uzima parametre, izvrši tražene radnje i rezultat stavi u određene memorijske lokacije ili u registre i vraća kontrolu programu.

Razlikujemo dva režima rada:
1. **Korisnički(user mode)** . U kome rade isključivo korisnički programi
2. **Sistemski(kernel-systemysupervisor mode**) koji je predviđen za OS.

23. Koji je zadatak komandnog interpretera?
-Komandni interpreter je primarni interfejs između korisnika i os-a
- Vidljiv je za sve korisnike računara,
- Nakon startovanja ispisuje **prompt** i čeka komande korisnika.

24. Prekidi:
Postoje dve strategije za **upravljanje uređajima:**
1. Polling: U određenim vremenskimm intervalima glavni procesor prekida svoj rad i proveri da li neki **kontroler** ima poruku za njega, ako ima on obradi poruku i nastavi sa radom.
Nedostaci polling strategije
1. Uređaj mora čekati na glavni procesor da bi predao svoju poruku – loše jer šta ako je bitna
2. procesor prekida rad u određenim vremenskim intervalima i kada nema nikakvih poruka

2. Prekidi(interrupt): Glavni procesor radi svoj posao kao i uređaji.
Ako uređaj završi svoj posao i dođe do neke greške on obavesti glavni procesor **zahtevom za prekid** i kada procesor dobije taj zahtev on prekida svoj i rad i pamti gde je stao.
Obradi zahtev i nastavi odakle je bio prekinut.

Interrupt table čuva adrese procedura za obradu prekida

Vrste prekida:
- **Hardverski prekidi –** Generišu se od strane hardvera, mogu biti
  - prekidi koji se mogu maskirati(**maskable interrupt**) – ove prekide proces može **ignorisati** ako je dobio takvu naredbu
  - prekidi koje se **ne mogu** maskirati(**non maskable interrupt**) – prekidi čija obrada ne može biti odložena – ozbiljne greške hardvera
- **Softverski prekidi –** generisani od strane programa
- **Sistemski pozivi –** ovo jesu softverski prekidi
- **Izuzeci –** generišu se od strane procesora – npr. ako delimo sa nulom

25. Navedi makar 5 operacija jezgra OS-a
1. Upravljanje prekidima
2. kreiranje i uništavanje procesa
3. sinhronizacija procesa
4. komunikacija procesa
5. manipulaciija sa PCB-om
6.podrška za ulaz izlaz
7- Upravljanje memorijom
8. upravljanje fajl sistemom
9. izbor procesa sa liste spremnih procesa(context switch)

26. Primer softverske sinhronizacije:
- Imamo terminale A i B sa po jednim procesom, koji je zadužen za brojanje pritiskanja ENTER-a, taj broj se čuva u zajedničkoj promenljivoj brojač

Pseudo kod:
1. Uzmi vrednost brojača
2. poveća vrednost za 1
3. vrati novu vrednost u brojač

- Problem nastaje ako A uradi 1. i 2. ali stane na 3., a B u isto vreme uradi sva tri koraka. A je uzela vredndost 5 i pretvorila u 6 ali nije vratila u brojač. B je uzela 5 vratila 6. a se pokrene i vrati 6. 2 puta pritisnut enter jednom povećan brojač.

27. Šta je kritična oblast?
-Deo programskog koda za koji važi: Ako je neki proces unutar svoje kritične oblasti, ni jedan drugi proces ne sme biti unutar svoje kritične oblasti.
-Posmatra se kao neprekidiv niz operacija – kao jedna primitivna operacija
- Određivanje kritične oblasti je zadatak programera – uglavnom je uzimanje vrednosti globalne promenljive i rad sa njom.
-Može se realizovati softverski, hardverski i pomoću OS-a (sistemski pozivi).

28. Softverska realizacija kritične oblasti.
Ovo su pretpostavke za realizaciju
0. Unutar kritične oblasti se može nalaziti istovremeno najviše 1 proces
1. K.O se realizuje softverski bez pomoći OS-a
2. Prilikom razvoja programa **ne** uzimamo u obzir pretpostavke ni o **brzini** ni o **broju procesora.**
3. Ni jedan proces izvan svoje kritične oblasti ne sme sprečiti druge procese da uđu u K.O.
4. Ni jedan proces ne sme neograničeno dugo čekati da uđe u svoju K.O.

Algoritam mora garantovati svakom procesu ulaz u K.O.

Postepen dolazak do Dekerovog algoritma:
Softverska sinhronizacija **Verzija1 – promenljivu ProcessNumber**

```
MODULE Version1 ;                          // teški proces

VAR processNumber : CARDINAL ;             // pomoćna promenljiva

   PROCEDURE Process1 ;                    // prvi laki proces
   BEGIN
      LOOP
         WHILE processNumber = 2 DO END ;  // čekamo dok je red na P2
         KriticnaOblast1 ;
         processNumber := 2 ;
         OstaleStvari1
      END
   END Process1 ;

   PROCEDURE Process2 ;                    // drugi laki proces
   BEGIN
      LOOP
        WHILE processNumber = 1 DO END ;   // čekamo dok je red na P1
         KriticnaOblast2 ;
         processNumber := 1 ;
         OstaleStvari2
      END
   END Process2 ;

BEGIN
   processNumber := 1 ;
   PARBEGIN ;
      Process1 ;
      Process2 ;
   PAREND
END Version1.
```

Govori da P1 i P2 treba da se pokrenu paralelno

Analiza koda: Ovaj algoritam radi dobro, istovremeno može biti najviše jedan proces u svojoj K.O.
Nedostatak:

- Procesi ulaze u svoje kritične oblasti alternativno p1p2p1p2…
- Pošto na početku imamo processNumber:=1, sigurno će P1 biti prvi koji će ući u k.o a kad izađe staviće ga na 2 i tako dozvoliti P2 da uđe u k.o.
- P2 ulazi u K.O. i prilikom izlaska stavi processNumber :=1 i neka se onda zaglavi u delu ostaleStvari2 dovoljno dugo vremena, da P1 udje i izađe iz K.O.
- Znači sada je processNumber=2 ali P2 je još uvek zaglavljen u OstaleStvari2 nije u K.O. što znači da bi P1 mogao ući u k.o ali ne može jer je processNumber = 2, znači P1 mora da čeka da P2 završi i ponovo uđe i izađe iz svoje k.o. kako bi promenio na 1 procesNumber

Zaključak: Problem ove verzije je u alternativnim ulascima u k.o. Radilo bi samo ako su brzine procesa približno jednake.

Verzija 2:

```
MODULE Version2 ;

VAR P1_INSIDE, P2_INSIDE : BOOLEAN ;

PROCEDURE Process1 ;
BEGIN
   LOOP
      WHILE P2_INSIDE DO END ;      // čekamo dok je P2 unutra
      P1_INSIDE := TRUE ;           // ulazimo
      KriticnaOblast1 ;
      P1_INSIDE := FALSE ;          // izlazimo
      OstaleStvari1
   END
END Process1;

PROCEDURE Process2 ;
BEGIN
   LOOP
      WHILE P1_INSIDE DO END ;      // čekamo dok je P1 unutra
      P2_INSIDE := TRUE ;           // ulazimo
      KriticnaOblast2 ;
      P2_INSIDE := FALSE ;          // izlazimo
      OstaleStvari2
   END
END Process2;

BEGIN
   P1_INSIDE := FALSE ;
   P2_INSIDE := FALSE ;
   PARBEGIN ;
      Process1 ;
      Process2 ;
   PAREND
END Version2.
```

Analiza: Uvođenjem ove dve promenljive rešili smo problem alternativnog izvršavanja ali smo dobili algoritam koji **ne radi.**

-Recimo da aje P1 stigao prvi do WHILE petlje i on odmaih ide i dalje. Pošto je P2_Inside = False, neka u ovom trenutku dođe i P2 do while petlje isto nastavi sa radom ali pošto je još uvek p1.inside false, dobili smo da će u isto vreme ući u kritičnu oblast.

Zaključak: Znači do problema dolazimo, ako se proces prekine između WHILE petlje i reda Px_INSIDE:=TRUE.

Verzija 3

```
MODULE Version3 ;

VAR p1WantsToEnter, p2WantsToEnter : BOOLEAN ;

   PROCEDURE Process1 ;
   BEGIN
      LOOP
         p1WantsToEnter := TRUE ;            // javimo da P1 želi ući
         WHILE p2WantsToEnter DO END ;       // dok P2 želi ući, čekamo
         KriticnaOblast1 ;
         p1WantsToEnter := FALSE ;           // izlazimo
         OstaleStvari1
      END
   END Process1;

   PROCEDURE Process2 ;
   BEGIN
      LOOP
         p2WantsToEnter := TRUE ;            // javimo da P2 želi ući
         WHILE p1WantsToEnter DO END ;       // dok P1 želi ući, čekamo
         KriticnaOblast2 ;
         p2WantsToEnter := FALSE ;           // izlazimo
         OstaleStvari2
      END
   END Process2;

BEGIN
   p1WantsToEnter := FALSE ;
   p2WantsToEnter := FALSE ;
   PARBEGIN ;
      Process1 ;
      Process2 ;
   PAREND
END Version3.
```

Analiza: Algoritam **ne radi**
Recimo da se p1 i p2 izvršavaju paralelno, p1 će staviti p1WantsToEnter:=True što će uraditi iP2 za svoju promenljivu.
Zatim će se oba procesa zaglaviti unutar while petlje i čekati jedno na drugo(**dead lock**)

Verzija 4:

```
MODULE Version4 ;

VAR p1WantsToEnter, p2WantsToEnter : BOOLEAN ;

   PROCEDURE Process1 ;
   BEGIN
      LOOP
         p1WantsToEnter := TRUE ;        // javimo da P1 želi ući
         WHILE p2WantsToEnter DO         // sve dok P2 želi ući
            p1WantsToEnter := FALSE ;    // odustajemo od toga da P1 uđe
            DELAY ( maliInterval1 ) ;    // čekamo malo,da to P2 primeti
            p1WantsToEnter := TRUE       // javimo da P1 želi ući
         END;
         KriticnaOblast1 ;
         p1WantsToEnter := FALSE ;       // izlazimo
         OstaleStvari1
      END
   END Process1;

   PROCEDURE Process2 ;
   BEGIN
      LOOP
         p2WantsToEnter := TRUE ;        // javimo da P2 želi ući
         WHILE p1WantsToEnter DO         // sve dok P1 želi ući
            p2WantsToEnter := FALSE ;    // odustajemo od toga da P2 uđe
            DELAY ( maliInterval2 ) ;    // čekamo malo,da to P1 primeti
            p2WantsToEnter := TRUE       // javimo da P2 želi ući
         END;
         KriticnaOblast2 ;
         p2WantsToEnter := FALSE ;       // izlazimo
         OstaleStvari2
      END
   END Process2;

BEGIN
   p1WantsToEnter := FALSE ;
   p2WantsToEnter := FALSE ;
   PARBEGIN ;
      Process1 ;
      Process2 ;
   PAREND
END Version4.
```

Analza: ova verzija zamalo da radi, ali postoji mogućnost greške.
-Pp. da se procesi izvršavaju paralelno: stavlja p1WantsToEnter:=TRUE, Process2 stavlja p2WantsToEnter:=TRUE, zatim oba procesa uđu u WHILE petlju, oba procesa za neko vreme odustaju od ulaska u k.o. Process1 čeka u trajanju maliInterval1 a Process2 u trajanju maliInterval2.
-Neka generatori slučajnih brojeva generišu iste vrednosti, tj neka je maliInterval1=maliInterval2, tada će ponovo biti **p1WantsToEnter:**=TRUE **i p2WantsToEnter**:=TRUE i sve počinjemo iz početka.

-Znači, mada je mala verovatnoća da oba generatora slučajnih brojeva izgenerišu isti niz vrednosti, postoji mogućnost da ovaj algoritam ne radi i da dođe do zaglavljivanja (**dead lock**) – a to može dovesti do vrlo ozbiljnih grešaka npr. ako se algoritam koristi za upravljanje nuklearnom elektranom, saobraćajem itd.

-Do zaglavljivanja dolazimo i ako se procesi izvršavaju baš alternativno: izvršimo jednu naredbu prvog procesa, pa jednu naredbu drugog procesa, pa jednu naredbu prvog, itd.

Rezime:

- **Verzija 1:** jedna globalna promenljiva koja kontroliše
  - **mana**: naizmeničan ulaz u K.O.

- **Verzija 2 :** svaka nit ima svoju globalnu promenljivu koja sadrži informaciju o tome da li je ta nit u svojoj K.O.
  - **mana:** moguć je istovremeni ulazak u K.O.

- **Verzija 3:** svaka nit ima svoju globalnu promenljivu koja sadrži informaciju da li nit želi da uđe u svoju k.o.
  - **mana:** moguće je beskonačno odlaganje tj. deadlock

- **Verzija 4:** uvodi se element slučajnosti
  - **mana** je jer je zasnovano na sličajnosti

REŠENJE: Dekkerov algoritam(kombinacija verzije 1 i 4, enum promenljiva)

Odgovara onom processNumber iz vr1

```
MODULE Dekker ;

VAR   p1WantsToEnter, p2WantsToEnter : BOOLEAN ;
      favouredProcess : ( first,second ) ;

   PROCEDURE Process1 ;
   BEGIN
      LOOP
         p1WantsToEnter := TRUE ;              // P1 javlja da želi ući
         WHILE p2WantsToEnter DO               // sve dok P2 želi ući
            IF favouredProcess = second THEN   // ako P2 ima prednost
               p1WantsToEnter := FALSE ;       // P1 odustaje
               WHILE favouredProcess = second DO END;  // čeka prednost
               p1WantsToEnter := TRUE          // pa javi da želi ući
            END
         END;
         KriticnaOblast1 ;
         favouredProcess := second ;           // prednost dajemo P2-u
         p1WantsToEnter := FALSE ;             // izlaz
         OstaleStvari1
      END
   END Process1;

   PROCEDURE Process2 ;
   BEGIN
      LOOP
         p2WantsToEnter := TRUE ;              // P2 javlja da želi ući
         WHILE p1WantsToEnter DO               // sve dok P1 želi ući
            IF favouredProcess = first THEN    // ako P1 ima prednost
               p2WantsToEnter := FALSE ;       // P2 odustaje
               WHILE favouredProcess = first DO END ;  // čeka prednost
               p2WantsToEnter := TRUE          // pa javi da želi ući
            END
         END;
         KriticnaOblast2 ;
         favouredProcess := first ;            // prednost dajemo P1-u
         p2WantsToEnter := FALSE ;             // izlaz
         OstaleStvari2
      END
   END Process2;

BEGIN
   p1WantsToEnter := FALSE ;
   p2WantsToEnter := FALSE ;
   favouredProcess := first ;
   PARBEGIN ;
      Process1 ;
```

Analiza u poređenju sa verzijama:

- **Sa verzijom 1: nema alternacije –** umesto jedne promenljive kao u vr1 koriste se dve logičke p1wantsToEneter i p2WantsToEnter
- Sa **Verzijom 2: Oba procesa ne mogu istovremeno ući u k.o.** - u slučaju da oba procesa istovremeno zahtevaju ulaz, prioritet određuje favouredProcess, odnosno favorizovani proces će ući u dok će drugi čekati u petlji
- Sa **Verzijom 3: nema zaglavljivanja(**dead lock**) –** proces koji nije favorizovan u datom trenutku odustaće od ulaska u K.O, čime omogućava ulazak drugom.
- Sa **Verzijom 4: nema zaglavljivanja usled elementa slučajnosti –** primena favouredProcess promenljive je deterministička.

Petrsonov algoritam (kraće i lepše od Dekerovog)

```
MODULE Petersen ;

VAR   p1WantsToEnter, p2WantsToEnter : BOOLEAN ;
      favouredProcess : ( first,second ) ;

   PROCEDURE Process1 ;
   BEGIN
      LOOP
         p1WantsToEnter := TRUE ;        // javimo da P1 želi ući
         favouredProcess := second ;     // dajemo prednost drugom proc.
         WHILE p2WantsToEnter AND        // sve dok P2 želi ući i
               ( favouredProcess = second ) DO END ; // P2 ima prednost
                                                     // čekamo
            KriticnaOblast1 ;
            p1WantsToEnter := FALSE ;                // izlaz
            OstaleStvari1
      END
   END Process1;

   PROCEDURE Process2 ;
   BEGIN
      LOOP
         p2WantsToEnter := TRUE ;        // javimo da P2 želi ući
         favouredProcess := first ;      // dajemo prednost prvom proc.
         WHILE p1WantsToEnter AND        // sve dok P1 želi ući i
               ( favouredProcess = first ) DO END ;  // P1 ima prednost
                                                     // čekamo
         KriticnaOblast2 ;
```



```
         p2WantsToEnter := FALSE ;                // izlaz
         OstaleStvari2
      END
   END Process2 ;

BEGIN
   p1WantsToEnter := FALSE ;
   p2WantsToEnter := FALSE ;
   favouredProcess := first ;
   PARBEGIN ;
      Process1 ;
      Process2 ;
   PAREND
END Petersen ;
```

Analiza u poređenju sa verzijama:

- **Verzijom V1: nema alternacije –** neka se Process2 zaglavi u deo OstaleStvari2 dovoljno dugo, da Process1 krene iz početka. Tada je p2WantToEnter=FALSE, p1WantToEnter=TRUE, favouredProcess=second, vrednost logičkog izraza WHILE petlje prvog procesa biće FALSE AND TRUE = FALSE, pa Process1 može ponovo ući u k.o. a da ne čeka na Process2.

- **Verzijom 2:** Ne mogu oba procesa istovremeno ući u kritičnu oblast (greška druge verzije): zbog pomoćne promenljive favouredProcess uvek će ući samo jedan i to onaj proces koji ima prednost.
- **Verzijom 3:**Nema zaglavljivanja (greška treć e verzije): proces koji nema prednost, ne odustaje ali će i dalje čekati na prednost. Drugi proces će moći ući u k.o jer ima prednost, pa će vrednost logičkog izraza u WHILE petlji biti: TRUE AND FALSE, a to je FALSE, što znači da proces može ući u k.o.
-
- **Verzijom 4 :** Nema zaglavljivanja (greška četvrte verzije): pošto ne može biti istovremeno favouredProcess=first i favouredProcess=second, tj. ne mogu oba procesa istovremeno imati prednost (jer promenljiva favouredProcess ne može imati istovremeno dve vrednosti, ili je first ili je second) – ne postoji mogućnost da oba procesa neogranič eno dugo ostanu unutar WHILE petlje.

29. Hardverska realizacija kritične oblasti.
-K.O. se može implementirati i na nivou hardvera, ako postoji procesorska instrukcija koja to podržava.
-Npr. Recimo da imamo naredbu **TestAndSet(a,b)** koja čita vrednost b, **kopira** tu vrednost u **a** i postavlja vrednost **b** na **TRUE.**
- Pošto test and set naredba procesora ona se izvršava kao jedna operacija i ne može biti prekinuta niti paralelno izvršena na više procesora

Primer hardverske sinhronizacije sa TestAndSet i inside

```
MODULE TAS ;

VAR inside : BOOLEAN ;                    // globalna pomoćna promenljiva

   PROCEDURE Process1 ;
   VAR firstCannotEnter : BOOLEAN ;       // lokalna pomoćna promenljiva
   BEGIN
      LOOP
         firstCannotEnter := TRUE ;       // javimo da P1 ne može ući
         WHILE firstCannotEnter DO        // sve dok P1 ne može ući
            TestAndSet(firstCannotEnter,inside)  // nedeljiva operacija
         END;
         KriticnaOblast1 ;
         inside := FALSE ;                // izlaz
         OstaleStvari1
      END
   END Process1;

   PROCEDURE Process2 ;
   VAR secondCannotEnter : BOOLEAN ;      // lokalna pomoćna promenljiva
   BEGIN
      LOOP
         secondCannotEnter := TRUE ;      // javimo da P2 ne može ući
         WHILE secondCannotEnter DO       // sve dok P2 ne može ući
            TestAndSet(secondCannotEnter,inside) // nedeljiva operacija
         END;
         KriticnaOblast2 ;
         inside := FALSE ;                // izlaz
         OstaleStvari2
      END
   END Process2;

BEGIN
   active := FALSE ;
   PARBEGIN ;
      Process1 ;
      Process2 ;
   PAREND
END TAS.
```

Analiza u poređenju sa verzijama softverske:
kod ovog algoritma koristimo jednu globalnu pomoćnu promenljivu inside koja označava, da li se neki proces nalazi unutar svoje kritične oblasti. Pored te globalne, svaki proces ima i jednu pomoćnu promenljivu, koja označava da li dati proces može ući u k.o.

- **Verzijom 1:** nema **alternacije – :** neka se Process2 dovoljno dugo zaglavi, tako da Process1 stigne ponovo do WHILE petlje. Tada je firstCannotEnter=TRUE i inside=FALSE (Process1 je postavio ovu vrednost, pošto je već jednom ušao i izašao iz svoje k.o.). Tada ć e Process1 jednom ući u WHILE petlju, ali će posle instrukcije TestAndSet biti firstCannotEnter=FALSE i inside=TRUE, pa će ponovo ući.

- **Verzijom 2:Ne mogu oba procesa istovremeno ući i u kritičnu oblast** pp. da oba procesa stignu istovremeno do WHILE petlje. Tada je inside=FALSE, firstCannotEnter=TRUE, secondCannotEnter=TRUE. Oba procesa će ući u petlju, ali pošto je naredba TestAndSet nedeljiva, i garantuje da se ne može paralelno izvršavati na više procesora, sigurno je da će samo jedan proces uspeti da je izvrši. Neka je to Process1. Tada će biti firstCannotEnter=FALSE i inside=TRUE. Što znač i, da e 30 Process1 ući u kritičnu oblast. Za sve ostale procese će važiti inside=TRUE, pa ć e svaka pomoć na lokalna promenljiva uzeti tu vrednost, i ti procesi će i dalje čekati unutar WHILE petlje.
- **Verzijom 3: Nema zaglavljivanja (**greška treće verzije): zbog nedeljivosti naredbe TestAndSet.

- **Verzijom 4: Nema zaglavljivanja** -ne postoji moguć nost da nakon prvog ulaska u WHILE petlju istovremeno bude firstCannotEnter=TRUE i secondCannotEnter=TRUE zbog naredbe TestAndSet i zbog toga što je inside=FALSE.

30. Koji su nedostaci softverske i hardverske sinhronizacije?
Zajednički nedostaci obe sinhronizacije su
- **busy waiting**(zaposleno čekanje) – situacija u kojoj proces ne radi ništa, korisno ali ipak troši procesorsko vreme,
- **odsustvo prioritizaciije procesa -** gde se može desiti da onaj navišeg prirориteta uđe u k.o. nakon što uradi mnoštvo procesa manjeg prioriteta

31. Realizacija pomoću sistemskih poziva

-Izbegavanje nedostataka softverske i hardverske realizacije ostvaruje se pomoću operativnih sistema, tj. kritičnu oblast ćemo realizovati pomoću sistemskih poziva.
-Kod ove realizacije proces čeka da uđe u K.O., postaje blokiran sve dok ga **drugi proces** ne odblokira

Problem proizvođača i potrošača – koristi se za ilustraciju svih varijanti sinhronizacije proces pomoću sistemskih poziva.
-Postoje proizvođač(puni bafer) i potrošač koji troši prozivedeno tj. prazni bafer.
- Proizvođač proizvodi sve dok skladište tj. bafer nije puno, a potrošač možr da troši dok bafer nije prazan - kada se desi da uslov nije ispunjen, proizvođač/potrošač ide na spavanje.

- Sistemski poziv **sleep** blokira onog koji ga je pozvao dok wakeup(p) budi proces p.

Kada proizvođač vidi da nema mesta u baferu, ode da spava (postaje blokiran) pomoć u naredbe Sleep – u tom stanju ostaje sve dok ga potrošač ne probudi. Kada potrošač vidi da je bafer prazan, odlazi na spavanje – u tom stanju ostaje sve dok ga proizvođač ne probudi. Proizvođač će probuditi potrošača kada stavi prvi elemenat u bafer – to znač i da je pre toga bafer bio prazan, pa potrošač spava. Potrošač će probuditi proizvođač a kada konstatuje da je zaspao, a ima mesta u baferu.

```
MODULE Sleep&WakeUp ;

CONST n = 100 ;                    // veličina bafera

VAR   count : CARDINAL ;           // broj elemenata u baferu
      bafer : ARRAY [1..n] OF INTEGER ;

   PROCEDURE Producer ;            // proizvođač
   VAR item : INTEGER ;
   BEGIN
      LOOP
         ProduceItem ( item ) ;    // proizvodi elemenat
         IF count = n THEN         // ako je bafer pun,
            Sleep                  // spavaj
         END;
         EnterItem ( bafer,item ) ;    // inače, ubaci elemenat u bafer
         INC ( count ) ;           // povećaj brojač
         IF count = 1 THEN         // ako smo stavili prvi elemenat,
            WakeUp ( Consumer )    // probudi potrošača, pošto je spavao
         END
      END
   END Producer ;

   PROCEDURE Consumer ;             // potrošač
   VAR item : INTEGER ;
   BEGIN
      LOOP
         IF count = 0 THEN          // ako je bafer prazan,
            Sleep                   // spavaj
         END;
         RemoveItem ( bafer, item ) ; // inače, uzmi elemenat iz bafera
         DEC ( count ) ;            // smanji brojač
         IF count = n-1 THEN        // ako je bafer bio pun,
            WakeUp ( Producer )     // probudi proizvođača
         END;
         ConsumeItem ( item )       // radi nešto sa preuzetom elementom
      END
   END Consumer ;

BEGIN
   count := 0 ;
```



```
   Init ( bafer ) ;
   PARBEGIN ;
      Producer ;
      Consumer ;
   PAREND
END Sleep&WakeUp ;
```

Analiza: algoritam **ne radi.**
Recimo da imamo sledeću situaciju:

- bafer je prazan, tada je count = 0, neka se izvršava proces potrošača.
- Potrošač pogleda vrednost counta, vidi da je 0 i uđe unutar if naredbe.
- Onda se prekine pre nego što uradi sleep i procesorsko vreme se daje proizvođaču(context switch)
- Proizvođač proizvede jedan element vidi da bafer nije pun, stavi element bafer poveća za 1 brojač.
- Tada primeti da je pre toga bafer bio prazan pa shvati da je potrošač zaspao i pokuša da ga probudi – tu nastaje problem jer potrošač nije uspeo da zaspi, pre toga je prekinut.

- **Pošto potrošač ne spava ta informacija se gubi**. Sada potrošač ponovo dobije procesor i ode da spava jer je ubeđen da je bafer prazan.
- Nakon nekog vremena, proizvođač će napuniti bafeer do kraja i otići na spavanje(čeka da ga probudi potrošač)
- Tako oba procesa spavaju do beskonačnosti- **dead lock**

Problem je korišćenje zajedničke promenljive count.
Da bi sinhronizacija bila uspešna, pristup toj promenljivoj mmora biti u K.O.

32. Koji je nedostatak sistemskih poziva sleep and wakeup.
- Upotrebom samo ova dva poziva ne garantuje se sinhronizovani pristup globalnoj promenljivoj, budući da se pristup njoj ne štiti a trebao bi(pomoću k.o.)

33. Kakve operacije izvršavamo nad brojačima događaja?
- Brojači događaja predstavljaju apstraktni tip podataka nad kojim se definišu sl operacije:
  - **init(e) –** inicijalizuje brojač e
  - **read(e) –** vraća trenutnu vrednost brojača događaja e
  - **Advance(e) –** povećava vrednost brojača događaja e za 1
  - **Await(e,v) –** čeka dok vrednost brojača e ne postane >= v

Važno je napomenuti da se vrednost brojača događaja uvek povećava nikad se ne smanjuje, ne može se podesiti na željenu vrednost, može se samo čitati i povećavati.
- Inicijalna vrednost brojača je 0.

Realizacija problema potrošača i proizvođača sa brojačem događaja:

```
MODULE ECounters ;

CONST n = 100 ;

VAR bafer : ARRAY [1..n] OF INTEGER ;
    in,out : EventCounter ;                  // brojači događaja

   PROCEDURE Producer ;                      // proizvođac
   VAR item  : INTEGER ;
       count : INTEGER ;            // ukupan broj proizvedenih elemenata
   BEGIN
      count := 0 ;
      LOOP
         ProduceItem ( item ) ;        // proizvodi elemenat
         INC ( count ) ;               // povećaj broj proizvedenih elem.
         Await ( out, count - n );     // spavaj ako nema mesta u baferu
         EnterItem ( bafer,item ) ;
         Advance ( in )                // povećaj brojač in za jedan
      END
   END Producer;

   PROCEDURE Consumer ;                // potrošač
   VAR item  : INTEGER ;
       count : INTEGER ;               // ukupan broj potrošenih elemenata
   BEGIN
      count := 0 ;
      LOOP
         INC ( count ) ;                  // povećaj broj potrošenih elem.
         Await ( in, count ) ;            // spavaj, ako je bafer prazan
         RemoveItem ( bafer,item ) ;
         Advance ( out ) ;                // povećaj brojač out za jedan
         ConsumeItem ( item )
      END
   END Consumer ;
```

Analiza: Koriste se brojači **in** koji označava broj ubacivanja u bafer, i **out** koji označava broj preuzimanja iz bafera pri čemu uvek važi da je in>= out (inače bi značilo da je preuzeto više nego što je ubačeno) i važi da je (in – out) < n gde je n veličina bafera(inače bi značilo da je ubačeno više elemenata nego što je veličina bafera).

Poizvođač: Nakon proizvodnje poveća broj proizvedenih elemenata i naredbom Await(out,count-n) proveri da li ima mesta u baferu da stavi novi elemenat.

- **Kada imamo mesto za novi elemenat?** Ako je broj slobodnih mesta u baferu već i od 1, tj ako je potrošač potrošio barem count-n elemenata (ako je out=count-1, onda je bafer prazan, ako je out=count-2, onda imamo n-1 mesta u baferu,...,ako je out=count-n, onda imamo jedno slobodno mesto u baferu).
- Rad će sigurno početi proizvođač , jer se inicijalno stavlja in=0,out=0, pa ć e važiti 0=out>=1-n, gde je naravno n>=1.
- Pri izlasku iz kritične oblasti, proizvođač javlja da je uspešno ubacio novi elemenat u bafer naredbom Advance(in).

Potrošač: u prvom koraku se povećava broj potrošenih elemenata, a zatim se čeka da proizvođač proizvede dovoljan broj elemenata sa naredbom Await(in,count), tj. potrošač može uzeti k-ti elemenat iz bafera, tek ako je proizvođač proizveo barem k elemenata

34. Semafori su deo i realizovani su pomoći pšerativnih sistema.

35. Šta je semafor?

- Semafor je **apstraktni tip** podataka koji se koristi kao **brojač buđenja.**
- Ako je vrednost semafora 0, to znači da nije došlo do buđenja.
- Ako je vrednost neki pozitivan broj onda je to broj sačuvanih buđenja.

-Nad semaforima definišemo dve operacije **Up(s) i Down(s) i** one su nedeljive tj ako spo počeli sa izvršavanjem, ne možemo biti prekinuti dok ne budemo gotovi.

36. Kako radi Down(s) i kako se ralizuje?

- Radi tako što proveri da je
  - vrednost semafora s>0 (znači imamo br sačuvanih buđenja) – ako jeste smanjuje vrednost semafora za 1 i vraća kontrolu.
  - Ako je vrednost s=0, to znači da nemamo više sačuvanih buđenja, proces odlazi na spavanje(moežže se posmatrati kao **generalizacija operacije sleep**).

```
Down ( s ) :    IF s > 0 THEN
                    DEC ( s )
                ELSE
                    uspavaj proces koji je pozvao Down ( s )
                END;
```

35. Kako radi Up(s) i kako se realizuje?

Radi sledeće

- **ako neki procesi spavaju(tj čekaju na semaforu)** budi se jedan od tih procesa – os bira nasumišno, u ovom slučaju vrednost S ostaje 0 ali imamo jedan manje uspavani proces.
- Ako je s>0 vrednost se povećava za 1.
- **Generalizacija WakeUp**

```
Up ( s ) :      IF ima uspavanih procesa na semaforu s THEN
                    probudi jedan od procesa koji čekaju na semaforu s
                ELSE
                    INC ( s )
                END;
```

Pored operacija Up(s) and Down(s) možemo definisati i:
- Init(s, n), inicijalizuje semafor sa vrednošću n
- Value (s) vraća vrednost semafora s.

38. Šta su binarni semafori?
- Žposeban tip semafora koji se inicijalizuju sa jedan i koriste se za spečavanje ulaska više od jednog procesa u k.o.
- Vrednost binarnih semafora može biti samo 1 ili 0:
- 0 – neki proces je unutar k.o.
- 1 – ni jedan proces nije unutar k.o.

39. Šta je mutex i na šta treba obratiti pažnju prilikom njegovog korišćenja
Ako svaki proces pre ulaska u svoju k.o. pozove Down i nakon ulaska Up, binarni semafori garantuju **međusobno isključivanje (MUT**ual **Ex**clusion**)**. Ne možemo imati vrednost veću od 1.

- Mora se voditi računa da pozivi Down(m) i Up(m) operacija mutex-a ne postave tako da se omogući da proces zaspi dok je svojoj k.o.

Problem poizvođača i potrošača sa semaforima.

```
MODULE Semaphores ;

CONST n = 100 ;

VAR bafer : ARRAY [1..n] OF INTEGER ;
    mutex,empty,full : Semaphore ;          // semafori

   PROCEDURE Producer ;                     // proizvođač
   VAR item : INTEGER ;
   BEGIN
      LOOP
         ProduceItem ( item ) ;       // proizvodi elemenat
         Down ( empty ) ;             // ako nema slobodnog mesta, čekaj
         Down ( mutex ) ;             // ako je neko unutar k.o., čekaj
         EnterItem ( bafer,item ) ;
         Up ( mutex ) ;              // izlazak iz k. o.
         Up ( full )                 // javi, da smo nešto stavili u bafer
      END
   END Producer ;

   PROCEDURE Consumer ;              // potrošač
   VAR item : INTEGER ;
   BEGIN
      LOOP
```



```
         Down ( full ) ;             // ako nema ništa u baferu, čekamo
         Down ( mutex ) ;            // ako je već neko unutar k. o., čekaj
         RemoveItem ( bafer,item ) ;
         Up ( mutex ) ;              // izlazak iz k.o.
         Up ( empty ) ;              // javi, da smo nešto vadili iz bafera
         ConsumeItem ( item )
      END
   END Consumer ;

BEGIN
   Init ( mutex,1 ) ;
   Init ( full,0 ) ;
   Init ( empty,n ) ;
   PARBEGIN ;
      Producer ;
      Consumer ;
   PAREND
END Semaphores ;
```

Analiza: koristimo 3 semafora: **mutex,full,empty**.-
-**Mutex** se inicijalizuje sa 1, što označava da nema procesa unutar k.o.,
-**full** se inicijalizuje sa 0, označ ava broj zauzetih mesta u baferu,
-**empty** se inicijalizuje sa n, i označava da je broj praznih mesta u baferu n, tj. da je bafer potpuno prazan.

Posmatrajmo rad **proizvođača:** pozivanjem Down(empty) obezbeđujemo, da u slučaju da nema više slobodnih mesta u baferu (empty=0), proces ode na spavanje, a ako ima, Down(mutex) obezbeđuje ulazak u k.o. tek ako nema nikoga unutar, inače opet spavanje.
-Izlaskom iz k.o., naredba Up(mutex) javlja potrošaču, da može ući u k.o. – mutex je tu sigurno 0, pa ako potrošač čeka na mutexu, biće probuđen, inače mutex postaje 1. Na kraju sledi naredba Up(full), tj javimo da smo stavili nešto u bafer, pa ako potrošač čeka na to, biće probuđen.

-Posmatrajmo rad **potrošača**: pozivanjem Down(full) obezbeđujemo, da u slučaju da je bafer prazan (full=0), proces ode na spavanje, a ako nije prazan, Down(mutex) obezbeđuje ulazak u k.o. tek ako proizvođač nije unutar, inače opet spavanje.
-Izlaskom iz k.o., naredba Up(mutex) javlja proizvođač u, da može ući u k.o., ako želi (tj. spava na semaforu mutex), ako proizvođač ne želi ući, biće mutex=1. Na kraju sledi naredba Up(empty), tj. javimo da smo uzeli nešto iz bafera, pa ako proizvođač čeka na to, biće probuđen.

Važno je primetiti da ovde koristim**o dva tipa semafora**:
1. mutex – koristimo da bismo obezbedili da unutar **kritične oblasti** u istom trenutku nalazi najviše do j**edan proces**
2. empty,full – koristimo da bismo obezbedili **sinhronizacij**u između procesa proizvođača i procesa potrošača.

Ova realizacija radi bez problema što se tiče implementacije semafora, ali pogrešno korišćenje semafora može dovesti do ozbiljnih grešaka:
npr. pp. da je **bafer pun,** tada ako kod proizvođača najre stavimo Down(mutex), pa tek onda Down(empty), dolazimo do sledeće situacije: proizvođač je ušao u kritičnu oblast i ne može biti prekinut, a pošto je bafer pun, otišao je i na spavanje – tada, potrošač vidi da bafer nije prazan, ali će i on zaspati jer ne može ući u k.o. zbog proizvođača – i tako dolazimo do zaglavljivanja.

40. Šta su monitori? Koje operacije tu imamo?

- Predstavlja skup **procedura, promenljivih, tipova podataka, kontranti …** koji su grupisani u specijalni modul ili paket.

Sadrže sledeće karakteristike:
- procedure iz monitora može istovremeno pozivati najviše 1 proces, znači ako je proces Apozvao neku proceduru monitora M, tada ni jedan drugi proces ne može pozvati ni jednu procesduru tog monitora, dok ta procedura ne završi sa radom.
- Procedure monira se izvršavaju bez prekida **atomarne su.**

- **Monitori nasuprot semafora predstavljaju karakteristiku programskih jezika**, a ne operativnih sistema. To znači da kompajleri tih jezika znaju da sa tim modulima treba raditi na poseban način: na osnovu prethodnih karakteristika.

-Pomoću monitora lako možemo rešiti probleme vezanih **za kritične oblasti**: kritičnu oblast ubacimo u monitor a ostalo je zadatak kompajlera.

-Monitori predstavljaju dobro rešenje za kritičnu oblast, ali to nije dovoljno: potrebno je naći neko rešenje i za to da se proces, koji je pozvao neku proceduru iz monitora pređe u stanje **BLOKIRAN (blocked**) ako je to potrebno

(npr. ako proizvođač primeti da nema mesta u baferu, treba da ode na spavanje i time omogući da potrošač uđe u svoju kritičnu oblast – pozivanjem neke procedure iz monitora).
-Da bismo ovaj problem rešili, uvodimo pojam **uslovnih promenljivih (condition variables).**

Uslovne promenljive su globalne promenljive (vidi ih svaki proces), nad kojima se definišu sledeć e nedeljive operacije (pošto su nedeljive, moraju biti unutar monitora!!):

- **Init (c)** inicijalizuje uslovnu promenljivu c
- **Wait (c) blokira (uspava) proces** koji je pozvao naredbu Wait na uslovnoj promenljivoj c – znači, ovaj proces postaje blokiran (čeka na signal c), a kontrola se predaje nekom drugom procesu koji sada već može ući u kritičnu oblast.
- **Signal (c) šalje signal c ostalim procesima** – ako postoji neki proces koji čeka na ovaj signal, treba da se probudi i da nastavi sa radom.
  - Ovde dolazimo do problema, jer izgleda da ćemo imati **dva procesa unutar monitora**: jedan koji je pozvao nardbu Signal (c), i drugi koji je č ekao na taj signal i sad je probuđen. Imamo dva pristupa za
  - **Rešenje ovog problema**: Hoare je predložio sledeće: neka proces koji je pozvao Signal postaje **suspendovan,** a proces koji je probuđen nastavi sa radom.
  - **Ideja Hansena** je slede a: proces koji je pozvao Signal, mora i**zaći iz monitora, t**j. Signal mora biti poslednja naredba odgovarajuće procedure monitora, tako će probuđeni proces biti jedini u monitoru i može nastaviti svoj rad. Mi ć emo koristiti Hansenovo rešenje. Ako nemamo ni jedan proces koji čeka na signal c, tekući proces nastavlja sa radom a signal se gubi.
- **Awaited (c)** vraća logičku vrednost TRUE, ako neki proces čeka (spava) na uslovnoj promenljivoj c, odnosno FALSE, ako takav proces ne postoji.

Poizvođači i potrošači preko monitora(LIFO)

```
DEFINITION MODULE Monitor ;
      PROCEDURE EnterItem ( item : INTEGER ) ;
      PROCEDURE RemoveItem ( VAR item : INTEGER ) ;
END Monitor.


IMPLEMENTATION MODULE Monitor[1000]; // modul, koji predstavlja monitor

CONST n = 100 ;

VAR count : CARDINAL ;                  // index elemenata
    bafer : ARRAY [1..n] OF INTEGER ;
    notFull, notEmpty : CondVAR ;       // uslovne promenljive, signali

   PROCEDURE EnterItem ( item : INTEGER ) ;       // ubacivanje u bafer
   BEGIN
      IF  count = n THEN                        // ako je bafer pun,
         Wait ( notFull )                       // spavamo, dok nam potrošač
```



```
      END;                                      // ne javi, da nije pun
      INC ( count ) ;
      bafer [ count ] := item ;
      IF count = 1 THEN                  // bafer je bio prazan,
         Signal ( notEmpty )             // potrošač je verovatno spavao
      END                                // traba ga probuditi
   END EnterItem;

   PROCEDURE RemoveItem (VAR item : INTEGER);  // izbacivanje iz bafera
   BEGIN
      IF count = 0 THEN                    // ako je bafer prazan,
         Wait ( notEmpty )                 // spavamo, dok nam proizvođač
      END;                                 // ne javi, da nije prazan
      item := bafer [ count ] ;
      DEC ( count ) ;
      IF count = n-1 THEN                // bafer je bio pun,
         Signal ( notFull )              // proizvođač je verovatno spavao
      END                                // treba ga probuditi
   END RemoveItem ;

BEGIN
   count := 0 ;
   Init ( notFull ) ;
   Init ( notEmpty )
END Monitor.
```

41. Ekvivalencija sistema za sinhronizaciju procesa.
- Problem ekvivalencije sistema za sinhronizaciju pocesa glasi:
Da li i kako možemo pomoću datog mehanizma sinhronizacije ralizovati neki drugi mehanizam.

Implementacija monitora pomoću semafora
- Osnovni zahtevi:
1. najviše jedan proces može pozvati proceduru monitora istovremeno
2. procedure monitora se izvršavaju bez prekida
3. potrebna podrška za uslovne promenljive
Rešenje k.o. - svakom monitoru se dodeljuje binarni semafor nazvan mutex init=1 i on kontroliše ulazak u monitor
Procedure:
- **EnterMonitor(mutex):** Down(mutex) – obezbeđuje da procedure nisu paralelne (npr ako A pozove neku proceduru monitora X, kompajler će automatski ubaciti poziv procedure EnterMonitor, pa ć e mutex postati 0. Neka sada i proces B pozove neku proceduru monitora X, tada se ponovo poziva EnterMonitor, pa pošto je mutex=0, proces B ide na spavanje. Isto tako ovaj postupak garantuje i neprekidno izvršavanje procedura monitora.)
- **LeaveNormally(mutex): Up(mutex) –** omogućava izlazak i putanje sledećeg procesa

Posmatrajmo sada **problem uslovnih promenljivih!**
Posmatrajmo na koje načine može neki proces **“ izaći” iz monitora**
(kako mogu procedure monitora završiti svoj rad).
Imamo 3 mogućnosti:

- **1. naredbom Wait (c)**, što znači da postaje blokiran i neki drugi proces dobija mogućnost ulaska u kritičnu oblast (u ovom slučaju zapravo **nema pravog izlaska** iz monitora, proces jednostavno postaje blokiran da bi pustio drugi proces da uđe u k.o.)
- **2. naredbom Signal (c)**, što znači da šalje signal c, što može dovesti do buđenja nekog procesa koji čeka na taj signal – koristiti samo ako je sigurno da neko čeka signal, tada ne radimo up(x) jer proces koji je čekao nastavlja u k.o. **mutex mora ostati 0**
- **3. normalno,** tj. neće pozvati Signal - koja će probuditi jednog procesa koji čeka na ulazak u k.o, ako takav proces postoji (i ostavić e mutex odgovarajuć eg monitora na nuli), a ako takav proces ne postoji poveć ava vrednost mutexa (pošto je mutex binarni semafor, bić e mutex=1), što će značiti da nema nikoga unutar k.o.

Rešenje je sledeće: svakoj uslovnoj promenljivoj pridružujemo po jedan semafor koji se inicijalizuje na nulu.

Implementacija semafora pomoću monitora:
Ideja: za svaki semafor – jedan brojač count i jedna uslovna promenljiva signal.
-Init(s,v) alocira memoriju(postavlja brojač sna.count:=v) inicijalizuje uslovnu promenljivu
- Down(S)- ako je count = 0 – wait(signal), proces spava, inače DEC(count)
-Up(S) ako neko čeka – signal(signal)- budi proces, ako niko ne čeka onda inc(count)
- Value – vraća trenutnu vrednost brojača
- Kill – oslobađa zauzetu memoriju

42. Komunikacija teških procesa
-Mehanizmi korišćeni za sinhornizaciju lakih procesa se ovde ne mogu koristiti jer teški procesi **ne dele ništa,** pa os mora da obezbedi neke strukture i mehanizme za međusobnu komunikaciju.
Tu nastaju problemi zbog daljine, različitih os/procesor

-**Komunikacija se obavlja slanjem poruka pomoču OS-a, voditi računa o:**

- Može se desiti da se **poruka izgubi** negde u mreži – odgovor: primalac šalje potvrdu pošiljaocu da je primio poruku, ako se ne javi dugo pošiljalac može opet poslati poruku.
- Može se desiti da se **potvrda izgubi,** pošiljalac pošalje opet i primalac odbije istu poruku dva puta
  - Rešenje pridružiti ID poruci da primalac može da razlikuje novu poruku od duplikata
- Treba jedinstveno indentifikovati kome je poruka namenjena u mreži, na kom računaru kom procesu
- **problemi u performansama** kada gledamo dva teška procesa koja se izvršavaju na istom računaru a komuniciraju porukama. Semafori i monitori mnogo brži.

Realizacija komunikacije zasnovane na porukama:
- Poruke mogu biti direktno upućene na jedinstvenu adresu nekog proces(Svaki proces mora imati jedinstveni indentifikacioni broj PID)
Problem: PID se mogu menjati – proces A komunicira sa procesom B, korisnik ubije B i pokrene druge procese, kasnije ponovo pokrene B i on dobije novi PID.

Drugi način je da se komunikacija odvija preko posebnih struktura podataka: **mailbox** i **pipe**
Razlika ove dve strukture:
mailbox razlikuje poruke – zna gde je granica između dve poruke, zna veličinu svake poruke.(FIFO)
Pipe ne vodi računa o granicama – primalac treba znati kkoliko bajtova treba da preuzme iz pipe

Šta se dešava ako proces želi da pošalje poruku ali nema mesta u mail boxu ili ako želi da uzme poruku a mailbox je prazan?
Možemo obavestiti proces da ne može poslati poruku ili ga uspavati(blokirati) dok ne bude mogao poslati poruku.

Kapacitet mailboxa i veličimu pipe-a određuju procesi.
Ako mailbox može primiti više od 1 poruke pricamo o randevuu.

Za realizaciju Komunikacije između teških procesa potrebno je obezbediti dve operacije:
SEND(mailbox, message) – šalje poruku meessage u sanduče mailbox
RECIEVE(mailbox, message) – preuzima poruku iz mailbox u message

Ovo je među klasičnim problemima ne verujem da ide

```
CONST n = 100 ;                    // veličina mailboxa

PROCEDURE Producer ;               // proizvođač
VAR item : INTEGER ;
    m : Message ;                  // poruka
BEGIN
   LOOP
      ProduceItem ( item ) ;       // proizvodi elemenat
      RECEIVE ( Consumer, m ) ;    // uzmi (čekaj na) poruku od potrošača
      BuildMessage ( m,item ) ;    // napravi poruku
      SEND ( Consumer,m )          // pošalji poruku potrošaču
   END
END Producer ;

PROCEDURE Consumer ;               // potrošač
VAR  item, i    : INTEGER ;
     m     : Message ;             // poruka
BEGIN
   MakeEmpty ( m ) ;
   FOR i := 1 TO n DO          // na početku,
      SEND ( Producer, m )     // šaljemo n praznih poruka proizvođaču
   END;
   LOOP
      RECEIVE ( Producer, m ) ;    // uzmi poruku od proizvođača
      ExtractItem ( m,item ) ;     // izvadi elemenat iz poruke
      MakeEmpty ( m ) ;
      SEND ( Producer,m ) ;  // obavesti proizv. da je poruka izvađena
      ConsumeItem ( item )   // koristi elemenat
   END
END Consumer ;
```

Analiza: potrošači proizvođač su se dogovorili, da će potrošač poslati praznu poruku proizvođač u čim je spreman da primi novu poruku.
Na početku rada, potrošač pošalje n praznih poruka da bi time obavestio potrošač a da je spreman za baferisanje ukupno n poruka (ima mailbox kapaciteta n).
Zatim čeka poruku od proizvođača. Ako proizvođač krene prvi, on će nakon što proizvede novi elemenat čekati na poruku od potrošača (RECEIVE(Consumer,m)).
Kada poruka stigne, upakuje novi elemenat u poruku (BuildMessage(m,item)) i šalje potrošaču (SEND(Consumer,m)).
Kada potrošač dobije poruku od proizvođača, vadi elemenat iz poruke (ExtractItem(m,item)) i šalje praznu poruku nazad (SEND(Producer,m)) da bi obavestio proizvođača da je preuzeo poruku i da je spreman za slede u poruku.