Zoltan Geller

Operativni sistemi

Beleške sa predavanja profesora Zorana Budimca

2000/2001

Univerzitet u Novom Sadu
Prirodno-matematički fakultet
Departman za matematiku i informatiku

2003

SADRŽAJ

PREDGOVOR

Skripta koju upravo čitate predstavlja beleške sa predavanja profesora Zorana Budimca iz predmeta Operativni Sistemi, održanih 2000/2001 godine, Prirodno-Matematički Fakultet, Novi Sad. Nastala je na osnovu sledećih knjiga, skripti i beležaka :

1. Sveska sa beleškama sledećih studenata : Zoltan Geller, Igor Mladenović, Agneš Sepeši
2. Andrew S. Tanenbaum : Modern Operating Systems
3. Skripta Dejana Špegara
4. Benyó Balázs, Fék Márk, Kiss István, Kóczy Annamária, Kondorosi Károly, Mészáros Tamás, Román Gyula, Szeberényi Imre, Sziray József: Operációs Rendszerek Mérnöki Megközelítésben
5. Knapp Gábor, Dr.Adamis Gusztáv: Operációs Rendszerek
6. Zoran Budimac, Mirjana Ivanović, Đura Paunić: Programski jezik Modula-2
7. Dragoslav Pešović: Sinhronizacija procesa (skripta)

Poglavlje ‘SIGURNOST I ZAŠTITA’ je u celosti i bez promena preuzet iz skripte Dejana Špegara.

UVOD

Softver se može podeliti u dve grupe:

1. **sistemski programi** – upravljaju računarom
2. **korisnički** (**aplikacioni**) **programi** – rešavaju probleme korisnika

Operativni sistem je fundamentalni deo sistemskih programa, čiji je zadatak upravljanje resursima računara i koji obezbeđuje osnovu za pisanje aplikacionih programa.

Kako obezbeđuje OS osnovu za pisanje aplikacionih programa?

Odgovor:

Računar je kompleksni sistem sastavljen od raznih delova, kao što su: procesor, memorija, diskovi, tastatura, miš, štampač, skener itd. Pisanje programa na taj način da se ti delovi računara programiraju direktno je vrlo težak posao. Zato je došlo do ideje da se stavi jedan sloj između aplikacionih programa i hardvera. Uloga tog sloja je da obezbedi neki interfejs (ili **virtuelnu mašinu**) ostalim programima radi lakšeg i bezbednijeg pristupa hardveru. Taj sloj je upravo OS.

Sada imamo sledeću situaciju:

<table>
<tr><td>office</td><td>baze podataka…</td><td>igre…</td><td>Korisnički programi</td></tr>
<tr><td>kompajleri,interpreteri</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>

Na najnižem nivou imamo **fizičke uređaje** – fizički delovi računara. Iznad tog sloja su **mikro programi** – direktno kontrolišu fizičke uređaje i obezbeđuju interfejs prema sledećem nivou. Predstavljaju elementarne korake od kojih se sastoje instrukcije mašinskog jezika. **Mašinski jezik** je skup instrukcija koje procesor direktno razume (izvršava ih pomoću svojih mikro programa).

Glavna funkcija operativnog sistema je sakrivanje detalje ovih nižih nivoa od ostalih programa i pružanje niza jednostavnijih instrukcija za pristup hardveru.

Šta je operativni sistem ?

1. **OS kao proširena** (**extended**) **ili virtuelna** (**virtual**) **mašina –** arhitektura (niz instrukcija, organizacija memorije, IO, itd.) računara na nivou mašinskog jezika je primitivna i nije pogodna za programiranje. Kao primer, možemo uzeti *NEC PD765* kontroler za disketni uređaj (koristi se na personalnim računarima). Taj kontroler ima 16 komandi. Najosnovnije komande su *READ* i *WRITE* i zahtevaju 13 parametara koja su smeštena u 9 bajtova. Prilikom

pristupa disketnom uređaju, treba voditi računa o tome, da li je motor uključen, pa treba naći stazu, pa sektor itd… - i to bi trebalo raditi svaki put kada želimo nešto pisati ili čitati sa diskete. Zadatak OS-a kao proširene ili virtuelne mašine je upravo to da te stvari radi umesto nas i da nam pruža neke funkcije višeg nivoa apstrakcije radi pristupa hardveru.

2. **OS kao upravljač resursima** (**resource manager**) – **RESURS** obuhvata sve što je programu potreban za rad (memorija, procesor, disk, štampač, skener, itd.). Kao upravljač resursima OS ima zadatak, da vodi računa u resursima računara – da zadovolji potrebe programa, da prati koji program koristi koje resurse, itd. Primer: imamo višekorisnički sistem: dva korisnika istovremeno žele nešto štampati – OS je dužan da vodi računa o tome da programi tih korisnika dođu do štampača kao resursa i da se podaci ne mešaju…

Istorija operativnih sistema

Istoriju operativnih sistema ćemo posmatrati preko istorije računara:

1. **Generacija:** (1945-1955) - računari pravljeni od **vakuumskih cevi**. Računari su bili ogromnih dimenzija i jako skupi (koristio ih je vojska), u jednom računaru je bilo čak i do 20.000 cevi, bili su jako spori, programirao se na mašinskom jeziku, programski jezici (ulkučujući i asemblera) i operativni sistemi su bili nepoznati. Ljudi koji su radili na tim računarima radili su sve: od programiranja do održavanja računara.

2. **Generacija:** (1955-1965) – računari se prave od **tranzistora**, postaju manji, pouzdaniji i jeftiniji tako da su ih mogli kupovati (pored vojske) i velike korporacije, univerziteti itd. Računari su bili odvojeni u posebnim sobama. Programeri pišu programe na papiru u programskom jeziku *FORTRAN*, zatim se ti programi prenose na bušene kartice. Bušene kartice se ostavljaju u sobi sa poslovima (input room). Sistem operator pokupi bušene kartice, u računar ubaci kartice sa *FORTRAN* kompajlerom, pa bušene kartice sa programom koji treba izvršiti. Računar odradi posao i razultat se dobija isto na bušenim karticama, koje se prenose u prostoriju sa rezultatima (output room). Ovde se mnogo vremena troši na šetanje između raznih prostorija sa bušenim karticama. Još uvek nema operativnog sistema.
   Uvodi se sledeće poboljšanje: - **paketna obrada** (**batch system**) – u sobi sa poslovima se sakuplja jedna količina sličnih (npr. svi zahtevaju fortran kompajler) poslova (programa), koji se pomoću jeftinijeg računara (npr. *IBM 1401*) prenosi na magnetnu traku. Nakon toga se magnetna traka prenosi u sobu sa glavnom računarom – moćniji i skuplji računar predviđen za izvršavanje programa (npr. *IBM 7094*), na glavni računar se učitava poseban program koji je zadužen da sa trake sa poslovima redom učitava programe i izvršava ih. Taj program je **predak današnjih OS-a**. Nakon izvršavanja programa, rezultat se snima na drugu magnetnu traku. Kada su svi poslovi (programi) odrađeni, operator stavlja drugu traku sa programima, a traku sa rezultatima prenosi do trećeg računara koji je zadužen za prebacivanje rezultata sa

magnetne trake na bušene kartice – to je ponovo neki jeftiniji računar (npr. *IBM 1401*). Manji računari nisu vezani za glavni računar, rade **off-line**.
Tipičan izgled poslova (na karticama):

```
$JOB, ,programer,….      // ime posla i neki podaci
$FORTRAN                 // treba nam FORTRAN kompajler
…fortranski program…     // program u FORTRANU koji treba prevesti
$LOAD                    // učitaj preveden program
$RUN                     // izvrši program sa sledećim podacima :
…ulazni podaci…
$END                     // kraj posla
```

3. **Generacija :** (1965-1980) – računari se prave od **integrisanih kola** (*IC*) – početkom 60-ih većina proizvođača računara proizvodi dve vrste računara : jednu jaču verziju (kao *IBM 7094*) i jednu slabiju (kao *IBM 1401*) – što je skup poduhvat. Novi korisnici računara žele slabije računare koji su jeftiniji, a posle nekog vremena treba im jači model koji je u stanju da izvrši sve stare programe, ali brže. *IBM* taj problem pokušava rešiti uvođenjem **Systema**/**360**. To je **serija kompatibilnih računara različitih snaga**. Svaki od ovih računara je pogodan i za naučnu i za poslovnu primenu, pa je time podela računara na dve vrste nestala. Ovaj koncept su preuzeli i ostali proizvođači računara. Računari iz serije *System/360* su radili pod **operativnim sistemom OS/360** – koji je bio jako glomazan i prepun grešaka.
   Razvija se nova disciplina : **softversko inžnjerstvo**.
   Multiprogramiranje (multiprogramming) : kada program čeka na rezultate IO operacija, procesor je neiskorišćen, pa se javlja gubljenje procesorskog vremena – to nije problem kod naučno-orijentisanih programa, koji retko koriste IO operacije, ali jeste kod poslovnih programa. Kao rešenje, javlja se multiprogramiranje: memorija se deli na particije u kojima se učitavaju različiti programi (jobs-poslovi). Dok neki program čeka na neke IO operacije, procesor može izvršavati drugi program. Na taj način, ako imamo dovoljan broj programa u memoriji, procesor je u stalnoj upotrebi.
   SPOOL (***S**imultaneous **P**eripheral **O**peration **O**n **L**ine*) – prebacivanje sadržaja bušenih kartica na disk (traku) pomoću posebnog uređaja – a bez procesora. Znači procesor izvršava program u memoriji, a paralelno se disk puni novim poslovima (jobs,programi). Kada jedan program završi sa radom, procesor na njegovo mesto može učitati drugi program sa diska. Isto se koristi i za izlazne podatke.
   Podela vremena (time-sharing): kod prve generacije, imali smo jednu grupu ljudi koji su koristili računar, programer je napisao program, ubacio u računar i dobio rezultate, nije čekao na obradu ostalih programa (jobs). Kod paketne obrade programer donese program na bušenim karticama, pa na rezultate ponekad čeka i nekoliko sati. Ako ima grešku u kodu, izgubi pola dana na čekanje. Kao rešenje uvodi se time-sharing (podela vremena): svaki korisnik ima tastaturu i monitor (**terminal**) koji su priključeni na **glavni računar**. To je jedan oblik multiprogramiranja. Procesorsko vreme različitim terminalima dodeljuje OS na osnovu dva kriterijuma:
   - čekanje na IO operacije
   - istek dodeljenog vremena (uvodi se pojam **quantum**-a – to je količina vremena nakon čijeg isteka kontrola se predaje drugom programu; multiprogramming + quantum = time-sharing).

   MULTICS (***MULTI**plexed Information and **C**omputing **S**ervice*) – neuspela ideja (*MIT,Bell Labs,General Electric*) da se napravi moćan računar koji će biti u stanju da radi sa velikim brojem terminala. Kao osnovu, uzeli su model distribucije električne energije: želim da slušam radio, samo ga uključim na struju… Isto su hteli napraviti sa računarima: u jednom gradu

imamo moćan centralni računar, a građani imaju terminale, kojima se pristupa tom računaru – predak računarskih mreža i Interneta.
Miniračunari: prvi miniračunar je *DEC*-ov (*Digital Equipment Corporation*) *PDP-1*, do tada najmanji i najjeftinij računar. Koštao je samo $120.000
UNIX: Ken Thompson, jedan od naučnika firme *Bell Labs*, koji je radio na projektu *MULTICS*, uzeo je jedan *PDP-7* miniračunar i napisao za taj računar mini verziju *MULTICS*-a – od tog projekta je posle nastao *UNIX* (UNI = jedan , X = CS – Computing Service).

4. **Generacija:** (1980-1990) – **Personalni Računari** – razvoj personalnih računara počeo je pojavom **LSI** (***L**arge **S**cale **I**ntegration* – veliki stepen integracije) čipova. Miniračunari su bili dovoljno jeftini, da ih imaju i više departmana iste firme ili univerziteta, a personalni računari su postali dovoljno jeftini da ih imaju i pojedinci. Primer personalnih računara: *Spectrum*, *Commodore*, *Atari*. Zatim *IBM PC*, *Apple Macintosh* , itd. Javljaju se prvi pravi operativni sistemi kao što su *MS-DOS*,*UNIX*,itd.
   Prave se programi za korisnike, koji nisu stručnjaci za računare, pa se razvija i korisnički interfejs programa. Dolazi do razvoja grafičkog okruženja.
   Pored klasičnih operativnih sistema javljaju se i dve nove vrste:
   Mrežni OS – računari su povezani u **mrežu**, svaki računar ima svoj OS koji međusobno komuniciraju pomoću nekog protokola (operativni sistemi mogu biti različiti, potrebno je samo zajednički protokol, tj. zajednički jezik za komunikaciju). Korisnik jednog računara, može se prijaviti na drugi, preuzeti neke fajlove itd. Korisnik zna da nije sam u mreži, svestan je različitih računara sa kojima komunicira preko mreže. Mreže mogu biti **lokalne** i **globalne**.
   Distribuirani OS – korisnici ga vide kao jedno-procesorski sistem, ali u stvari radi sa više procesora. Imamo više računara povezanih u mrežu, ali samo jedan OS, koji upravlja svim resursima u mreži. U pravom distribuiranom sistemu korisnik ne treba da vodi računa o tome, gde su njegovi fajlovi smešteni ili gde se izvršava njegov program, to je posao OS-a. Distribuirani OS se dakle ponaša kao jedinstvena celina. Korisnik nije svestan toga da je u mreži sa drugim računarima, njemu to izgleda kao da je jedan računar.

Osnovni pojmovi, koncepti OS-a

Imamo hardver, operativni sistem i korisničke programe. Videli smo da je jedan od zadataka OS-a da sakrije hardver od aplikacionih programa, odnosno da obezbedi lakši pristup hardveru. To se ostvaruje preko niza proširenih instrukcija, koji se zovu **sistemski pozivi** (**system calls**).

Procesi

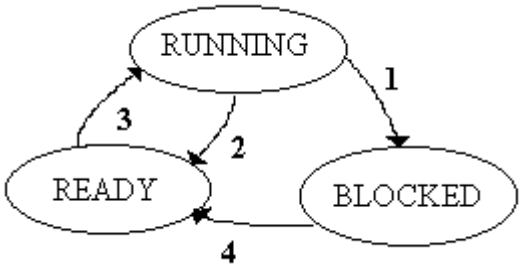
Procesi predstavljaju jedan od najvažnijih koncepata operativnih sistema. **Program** je niz instrukcija koji ostvaruje neki algoritam. **Proces** je program u statusu izvršavanja, zajedno sa svim

resursima koji su potrebni za rad programa. Znači: program je fajl na disku. Kada se taj fajl učita u memoriju i počinje da se izvršava dobijemo proces.

Stanja procesa

Procesi se nalaze u jednom od sledećih stanja:

- proces se **izvršava** (*RUNNING*) - procesor upravo izvršava kod ovog procesa
- proces je **spreman**, ali se ne izvršava (*READY*) - proces je dobio sve potrebne resurse, spreman je za izvršavanje, čeka procesora
- proces je **blokiran**, čeka na nešto (npr. čeka štampača da završi sa štampanjem – *BLOCKED*) - za dalji rad procesa potrebni su neki resursi, koji trenutno nisu na raspolaganju, čeka IO operaciju, rezultat nekog drugog procesa itd.



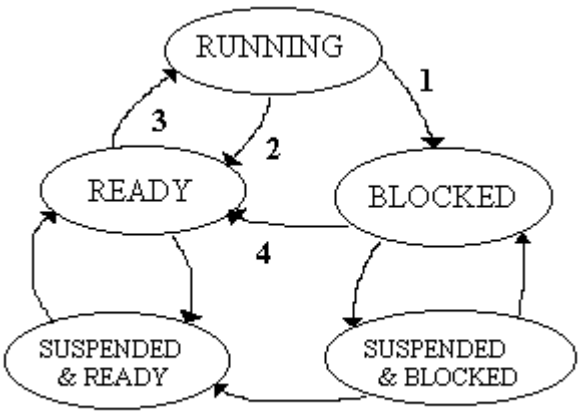
Imamo 4 prelaska između različitih stanja:

1. proces prelazi iz stanja IZVRŠAVANJA u stanje BLOKIRAN kada su mu za dalje izvršavanje potrebni neki resursi, koji trenutno nisu dostupni. Ovu promenu stanja vrši sam proces: predaje zahtev za neki resurs, pa čeka tog resursa. Npr.: pošalje zahtev skeneru da skenira neku sliku, i čeka rezultat skeniranja
2. proces prelazi iz stanja IZVRŠAVANJA u stanje SPREMAN ako mu istekne dodeljeno procesorsko vreme (*time-sharing*) – tada proces prelazi u listu procesa koji čekaju na procesor
3. proces prelazi iz stanja SPREMAN u stanje IZVRŠAVANJA kada se procesor oslobodi i može da izvršava kod posmatranog procesa (izabere se iz liste čekanja po nekom kriterijumu i izvršava se)
4. proces prelazi iz stanja BLOKIRAN u stanje SPREMAN, kada dođe do potrebnih resursa i spreman je za dalji rad, ali procesor trenutno nije slobodan, pa prelazi u listu čekanja (npr. skener je završio skeniranje, i sad proces može nastaviti sa radom (spreman je), ali procesor je trenutno zauzet izvršavanjem nekog drugog procesa, pa mora da čeka u red…)

Kod nekih operativnih sistemima procesi mogu biti i **suspendovani** (**suspended**). Na taj način dobijamo još dva stanja:

- proces je **suspendovan** i **spreman** (ako je došlo do suspendovanja u stanju spreman)
- proces je **suspendovan** i **blokiran** (ako je došlo do suspendovanja u stanju blokiran)

i sledeći dijagram:



Proces koji je **suspendovan**, prestaje da se takmiči za resurse, oslobađaju se resursi koje je zaouzeo, ali ostaje i dalje proces.

Proces koji je u stanju suspendovan i blokiran prelazi u stanje suspendovan i spreman, ako postaje spreman, tj. ako može da nastavi sa radom (npr. proces pošalje zahtev skeneru da skenira sliku, čeka da skener završi sa radom, pa se blokira, u međuvremenu se suspendira, pa postaje suspendovan i blokiran, kada skener završi skeniranje, proces prelazi iz stanja suspendovan i blokiran u stanje suspendovan i spreman.)

Iz stanja suspendovan i blokiran u stanje blokiran i iz stanja suspendovan i spreman u stanje spreman procesi mogu preći samo explicitno, tj. zahtevom korisnika.

Iz stanja spreman u stanje suspendovan i spreman proces prelazi iz nekog od sledećih razloga :

- **prevelik broj spremnih procesa** – procesi se suspendiraju kao zaštita od preopterećivanja sistema
- **explicitno suspendiranje procesa od strane korisnika** (npr. da bi korisnik mogao proveriti neke međurezultate izvršavanja procesa – i nakon toga mogao nastaviti rad bez ponovnog pokretanja celog programa.)
- **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 procesa P2, a procesu P2 treba resurs B koji drži P1 – ovi procesi su se zaglavili, jer nijedan od njih ne može nastaviti sa radom – u ovom slučaju jedan od procesa se suspenduje, pa drugi može da odradi svoj zadatak, pa kada se resursi oslobode i prvi će moći da završi svoj rad).

PCB – Process Control Block

Posmatrajmo sledeću situaciju: imamo multiprogramirano okruženje sa dva procesa. Proces P1 se izvršava dok ne dođe do blokiranja zbog čekanja na neki događaj (npr. skeniranje). Tada krenemo sa izvršavanjem procesa P2 koji se nakon nekog vremena isto postaje blokiran. U međuvremenu se desi događaj na koji je čekao proces P1 i sada možemo nastaviti sa izvršavanjem. Da bismo znali, gde treba nastaviti, potrebno je pamtiti neke informacije o procesu. Upravo tome služi *PCB* tj. *Process Control Block*. Svakom procesu se dodeljuje jedinstveni *PCB* koji sadrži informacije koje su potrebne za nastavak izvršavanja tog procesa. Te informacije uključuju:

- jedinstveni identifikator procesa (**pid** – process ID)
- stanje procesa
- **prioritet** procesa (iz liste čekanja biramo najpre procese sa većim prioritetima)
- adresa memorije gde se nalazi proces
- adrese zauzetih resursa

- sadržaj registara procesora, itd.

PCB-ovi svih procesa u memoriji smeštaju se u neki niz ili povezanu listu.

Operacije sa procesima

Operativni sistem treba da obezbedi sledeće operacije sa procesima:

- kreiranje novog procesa (kreiranjem *PCB*-a)
- uništavanje procesa (brisanjem *PCB*-a iz liste)
- menjanje stanja procesa
- menjanje prioriteta procesa
- izbor procesa za izvršavanje (dodela procesora nekom procesu – **context switch**)
- **sinhronizacija** procesa
- **komunikacija** između procesa… itd.

Odnos procesa

I sami procesi mogu kreirati nove procese. U tom slučaju proces koji kreira novi proces zove se **roditelj** a novi proces **dete**, pa je odnos procesa hijerarhijski (u obliku stabla). Između roditelja i deteta možemo imati dve vrste veze:

1. proces-roditelj kreira novog procesa i čeka dok proces-dete završi sa radom
2. proces-roditelj kreira novog procesa i nastavljaju sa radom oba procesa (rade paralelno)

Podela operativnih sistema u odnosu na procese

Operativni sistemi mogu podržavati:

- **monotasking** (jednoprocesni, monoprogramiranje) : u memoriji istovremeno imamo samo jedan program, tj. "istovremeno" se izvršava samo jedan proces (npr. *DOS*)
- **multitasking** (višeprocesni, multiprogramiranje) : u memoriji istovremeno imamo i više programa, tj. "istovremeno" se izvršavaju i više procesa (npr. *Windows*,*Linux*)

U odnosu na broja korisnika, operativni sistemi mogu biti:

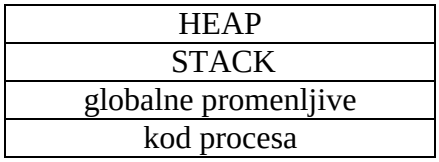
- **monouser** (jednokorisnički): npr. *DOS* (jednokorisnički,jednoprocesni), *Windows 98* (jednokorisnički, višeprocesni)
- **multiuser** (višekorisnički): npr. *UNIX* (višekorisnički, višeprocesni)

Teški i laki procesi

Procesi se dele na **teške** i na **lake** procese. Da bismo videli razliku između ove dve vrste procesa, posmatrajmo najpre kako procesi koriste memoriju. Za izvršavanje nekog procesa, potrebno je 4 dela (vrste) memorije:

- **heap**
- **stack**
- **memorija za globalne promenljive**
- **memorija za kod procesa**

Memorija zauzeta od strane nekog procesa izgleda na sledeći način:



HEAP (**hrpa**) je deo memorije koji je rezervisan za dinamičke promenljive, tj. za promenljive koje se stvaraju u toku izvršavanja programa. Npr. u *Moduli-2* zauzimanje memorije (kreiranje dinamičke promenljive) se vrši naredbom *ALLOCATE* iz modula *Storage*, a oslobađanje zauzete memorije naredbom *DEALLOCATE* isto iz modula *Storage*. Pristup dinamički kreiranim promenljivama se ostvaruje pomoću pokazivača. To su promenljive koje sadrže početnu adresu zauzete memorije. Na jednu istu dinamičku promenljivu mogu pokazivati i više pokazivača. Ako smo zauzeli neku količinu memorije i nemamo više ni jednog pokazivača na tu memoriju, taj deo *HEAP*-a je izgubljen (nemamo više pristupa, ne znamo adresu zauzete memorije, ne možemo je osloboditi) – to se zove **smeće** u memoriji (**garbage**) i može dovesti do ozbiljnih grešaka u radu programa. Ovaj problem se kod nekih OS-a rešava pomoću **sakupljača smeća** (**garbage collector**) koji vode računa o zauzetim delovima memorije: npr. broje pokazivače na zauzetu memoriju i ako taj brojač postaje nula, oslobađaju memoriju. Postoje i programski jezici koji imaju ugrađen sakupljač otpadaka kao što su *Java*, *PC Scheme* itd.

STACK (**stek**) je deo memorije koji je rezervisan za čuvanje lokalnih promenljivih, parametara procedura, povratnih adresa itd.

Pod *DOS*-om maximalna veličina steka-a je 64Kb, a maximalna veličina heap-a je 640Kb. Veličina heap-a i stack-a je unapred određena i ne može se menjati u toku izvršavanja programa. Stek se puni prema gore a heap se puni prema dole. Npr. u *Turbo Pascalu* postoji **direktiva** (kompajler vrši prevod izvornog koda u funkciji direktiva ugrađenih u izvorni kod) *$M* koja određuje veličinu *heap*-a i *stack*-a na sledeći način:
{$M veličinaSteka,heapMin,heapMax}.

Podela procesa na teške i lake procese vrši se na osnovu toga, kako koriste gore navedene delove memorije :

- svaki **teški proces** ima sopstveni memorijski prostor za kod, globalne promenljive, stek i heap koju ne deli ni sa kim, pristup tim delovima ima samo dati proces.
- **laki procesi** (**niti, threads**) mogu deliti memorijski prostor za kod, globalne promenljive i heap. Stek se ne može deliti jer ne možemo unapred znati koji proces koristi stek: proces A stavi nešto na stek i nastavlja rad, dolazi proces B i on isto stavi nešto na stek, A je završio svoj rad i sad su mu potrebni podaci sa steka, a tamo su podaci procesa B… Znači laki procesi imaju sopstveni stek a mogu deliti kod,globalne promenljive i heap.

Programski jezici, koji omogućavaju kreiranje lakih procesa: *Java,Modula-2,Concurrent Pascal,Ada*,itd.
U *M2* laki procesi su zapakovani u teške procese, kompajler deli vreme između više procesa.

Postoje operativni sistemi, koji podržavaju:

1. samo teške procese (*UNIX*)
2. samo lake procese (*Oberon*)
3. podržavaju i teške i lake procese (*Windows*)

Problem komunikacije kod teških procesa je u tome, što su oni međusobno odvojeni, tj. ne dele nijedan deo memorije. Kako nemaju zajedničku memoriju, operativni sistem mora obezbediti neke strukture i mehanizme za **međuprocesnu komunikaciju** (**interprocess communication**).

Laki procesi mogu bez problema komunicirati jer dele određene delove memorije. Kod njih se javlja problem **sinhronizacije**.

Konkurentno programiranje

Kod **SEKVENCIJALNOG** programiranja, podrazumevamo da se prorgam sastoji od niza naredbi koji se izvršava u tačno određenom redosledu od strane jednog procesora. Znači sekvencijalni program predstavlja tačno jedan proces koji se izvršava od početka do kraja na jednom procesoru.

KONKURENTNO programiranje uzima u obzir mogućnost postojanja i više procesora, pa se konkurentni program sastoji od više nezavisnih procesa ili zadataka (*task*), koji se mogu istovremeno izvršavati. Stil konkurentnog programiranja koristimo kada se zadatak može razbiti na više međusobno relativno nezavisnih delova – procesa, koji se mogu istovremeno izvršavati. Konkurentni programi se mogu izvrsavati i na računaru sa jednim procesorom i na računaru sa više procesora.

Konkurentno programiranje se koristi kod programiranja operativnih sistema, simulacija itd. Npr.: želimo napisati program za simulaciju igre jednog fudbalskog tima. Sekvencijalno: pišemo program koji vodi računa o svakom igraču tima. Konkurentno: za svaki igrač pišemo jedinstvenu proceduru – te procedure se izvršavaju istovremeno.

Fajl sistem (file system)

Videli smo da je operativni sistem dužan da obezbedi lakši pristup hardveru programima na višim nivoima. Na isti način, treba da obezbedi i apstraktniji pogled na fajlove, na fajl sistem. Znači, programi na višim nivoima ne treba da vode računa o tome, kako su u stvari fajlovi smešteni na disku, o tome će voditi računa OS. Za rad sa fajlovima, OS treba da obezbedi operacije, kao što su:

- otvaranje fajlova
- zatvaranje fajlova
- promena pozicije fajl-pokazivača (*FilePos*)
- čitanje iz fajlova
- pisanje u fajlove
- kreiranje novih fajlova
- brisanje postojećih fajlova
- reimenovanje,kopiranje, itd.

Mnogi operativni sistemi podržavaju i koncept **direktorijuma**, kao način grupisanja fajlova – pa obezbeđuju i operacije za rad sa njima. Elementi direktorijuma mogu biti fajlovi i drugi direktorijumi, tako dolazimo do fajl sistema u obliku stabla.

Sistemski pozivi (system calls)

Aplikacioni programi komuniciraju sa OS-om pomoću **sistemskih poziva**, tj. preko operacija (funkcija) definisanih od strane OS-a. Sistemski pozivi se realizuju pomoću sistema **prekida**: korisnički program postavlja parametre sistemskog poziva na određene memorijske lokacije ili registre procesora, inicira prekid, OS preuzima kontrolu, uzima parametre, izvrši tražene radnje, rezultat stavi u određene memorijske lokacije ili u registre i vraća kontrolu programu.

Sistemske pozive često podržava i hardver, tj. procesor, na taj način što razlikuje dva režima rada: **korisnički režim** (**user mode**) i **sistemski režim** (**kernel mode**, **system mode**, **supervisor mode**) rada. Korisnički programi mogu raditi isključivo u korisničkom režimu rada procesora, sistemski režim rada je predviđen za OS. Ako korisnički program pokuša izvršiti neku operaciju koja je dozvoljena samo u sistemskom režimu rada, kontrola se predaje OS-u. Prilikom sistemskih poziva procesor prelazi iz korisničkog režima rada u sistemski, OS obradi poziv pa se procesor vraća u korisnički režim rada.

Primer sistemskog poziva u *TopSpeed Moduli-2* pod *DOS*-om :

```
FROM SYSTEM IMPORT Registers ;
FROM Lib IMPORT Dos ;

VAR r : Registers ;
BEGIN
   r.AH : = 02H ;             // broj sistemskog poziva: servisa DOS-a
                              // 02H označava servis za ispis jednog
                              // karaktera na standardni izlaz – ekran
   r.DL : = BYTE('A') ;       // karaketer koji treba ispisati:
                              // parametar sistemskog poziva
   Dos ( r )                  // sistemski poziv
END
```

Komandni interpreter (shell)

Operativni sistem je zadužen za sistemske pozive. Komandni interpreter interpretira komande korisnika i izvršava ih oslanjajući se na sistemske pozive OS-a. To je deo OS-a koji je vidljiv za korisnike računara. Npr. u *DOS*-u to izgleda na sledeći način: komandni interpreter nakon startovanja ispisuje **prompt**, i čeka komande korisnika:

C:\TEMP\dir

Komandni interpreter je primarni interfejs između korisnika i operativnog sistema.U *DOS*-u to je fajl `COMMAND.COM`.

Prekidi (interrupts)

Imamo galvni procesor, koji izvršava programe i imamo IO uređaje (disk,štampač,skener,itd.). Svaki IO uređaj ima svoj kontroler koji sadrži neki slabiji procesor koji je jedino zadužen za upravljanje tim uređajem i za komunikaciju sa glavnim procesorom. Imamo dve strategije za upravljanje uređajima:

1. **polling**: u određenim vremenskim intervalima glavni procesor prekida svoj rad i proveri da li neki kontroler ima neku poruku za njega, ako ima, obradi poruku i nastavlja sa radom
2. **prekidi (interrupt)**: glavni procesor radi svoj posao, i uređaji rade svoj posao. Ako uređaj završi svoj posao ili dođe do neke greške, uređaj o tome obaveštava glavni procesor **zahtevom za prekid** (**interrupt request**). Kada procesor dobije zahtev za prekid, prekida svoj rad, zapamti gde je stao, obradi prekid, pa nastavlja sa radom tamo gde je bio prekinut (ako prekid ne predstavlja neku fatalnu grešku, takvu da se rad ne može nastaviti).

Primer: Mama kuva ručak a dete se igra. Ako je polling, mama svakih 10 sekundi proveri šta dete radi i ako nešto ne valja onda dotrči da mu pomogne. Ako su interapti onda ga ne proverava. Ali, dete se poseče i počne da plače. To je interapt. Mama skine šerpe sa šporeta (prekine proces pravljenja ručka) zapamti gde je stala, locira dete (pogleda u tabelu), previje ga i poljubi (obradi prekid), vrati se u kuhinju, seti se gde je stala i nastavi sa kuvanjem. Ako se dete baš unakazilo, mama suspenduje proces pravljenja ručka i vodi dete na urgentno.

Nedostaci polling strategije:

1. uređaj mora čekati na glavni procesor da bi predao svoju poruku (koja može biti npr. vrlo važna poruka o grešci)
2. procesor prekida svoj rad u određenim vremenskim intervalima i kada nema nikakvih poruka

Obrada prekida se izvršava po sledećem algoritmu:

- procesor radi svoj zadatak
- stiže zahtev za prekid
- sačuva se stanje trenutnog procesa
- onemogućavaju se dalji prekidi
- u **tabeli prekida** (**interrupt table** - čuva adrese procedura za obradu svih mogućih prekida) traži se adresa procedure za obradu prekida
- izvršava se procedura za obradu prekida
- omogućavaju se dalji prekidi
- nastavlja se rad na prekinutom procesu

Vrste prekida:

1. Hardverski prekidi – generišu se od strane hardvera, mogu biti:
    1. prekidi koji se **mogu maskirati** (**maskable interrupt**) – ove prekide procesor može ignorisati ako je dobio takvu naredbu (npr. pritisnuta je tipka na tastaturi)
    2. prekidi koji se **ne mogu maskirati** (**non maskable interrupt**) – prekidi čija obrada ne može biti odložena – to su neke ozbiljne greške hardvera (npr. greška memorije)
2. Softverski prekidi – prekidi generisani od strane programa
3. Sistemski pozivi (jesu softverski prekidi)
4. **Izuzeci** (**exceptions**) – generišu se od strane procesora, npr. ako se pokuša deliti sa nulom

Jezgro OS-a (kernel,nucleus,core)

Jezgro je deo operativnog sistema, koji obavlja najbitnije operacije:

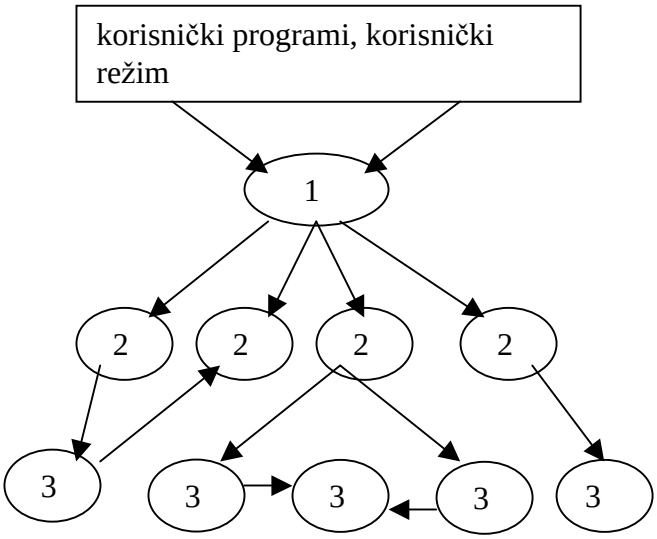
- upravljanje prekidima
- kreiranje i uništavanje procesa
- odabiranje procesa iz liste spremnih procesa (context switch)
- suspenzija i nastavljanje procesa
- sinhronizacija procesa
- komunikacija između procesa
- manipulacije sa *PCB*-om
- podrška za ulaz/izlaz (IO)
- upravljanje memorijom
- upravljanje fajl sistemom, itd.

Dizajn, struktura operativnih sistema

Monolitni sistemi (monolithic systems)

U prošlosti monolitni sistemi su predstavljali najčešću organizaciju OS-a. Ovaj način organizacije OS-a dobio je naziv "**The Big Mess**" – velika zbrka, videćemo zašto. OS je realizovan kao skup procedura, od kojih svaka može pozvati svaku ako je to potrebno. Korisnički programi servise OS-a koriste na sledeći način: parametri sistemskog poziva se smeštaju u određena mesta, kao što su registri procesora ili stek, pa sledi specijalna operacija – **poziv jezgra** OS-a (**kernel call**). Ova operacija prebacuje procesor iz **korisničkog režima** rada u **sistemski režim** rada i kontrolu predaje OS-u. U korisničkom režimu rada nisu dozvoljene neke komande procesora, a u sistemskom režimu mogu se koristiti sve operacije poznate od strane procesora. Posle poziva jezgru, OS preuzima kontrolu, na osnovu parametara poziva određuje koju sistemsku proceduru treba pozvati i poziva tu proceduru i na kraju se kontrola vraća korisničkom programu. Znači OS ima sledeću strukturu (sastoji se od 3 sloja):

1. glavni program koji obrađuje sistemske pozive
2. skup sistemskih procedura koje se pozivaju prilikom sistemskih poziva
3. skup pomoćnih procedura koje se koriste od strane sistemskih procedura



Slojevita realizacija (layered systems)

Kod slojevite realizacije OS se deli na različite slojeve na hijerarhijski način: svaki sloj se gradi na slojeve ispod njega. Prvi OS koji je napravljen na ovaj način je OS sa imemom **THE** (*Technische Hogeschool Eindhoven*) od strane **E.W.Dijkstre**. *THE* se sajtojao od 6 sloja na sledeći način:

5.	komandni interpreter
4.	korisnički programi
3.	ulazne,izlazne operacije
2.	procesi
1.	upravljanje memorijom
0.	procesor i multiprogramming

Nulti sloj se bavi upravljanjem procesora, obezbeđuje prebacivanje između različitih procesa.

Prvi sloj upravlja memorijom: zauzima potrebnu memoriju za procese. Slojevi iznad prvog sloja ne treba da brinu o memorijskim potrebama, to će umesto njih uraditi prvi sloj.

Drugi sloj upravlja komunikacijom između različitih procesa i komandnog interpretatora.

Treći sloj obavlja ulazno\izlazne operacije.

Slojevi 0..3 predstavljaju jezgro OS-a i rade u sistemskom režimu rada.

Na četvrtom nivou imamo korisničke programe – oni ne treba da se brinu ni oko procesa, memorije,komandnog interpretera, IO operacija, sve to obavljaju slojevi ispod.

Znači **bitna razlika između monolitne i slojevite realizacije** je u tome, što se OS kod monolitne strukture sastoji od skupa procedura bez ikakvog grupisanja ili hijerarhije, a kod slojevite realizacije OS se deli na više slojeva od kojih se svaki oslanja na slojeve ispod, i gde svaki sloj ima tačno određenu funkciju (upravlja tačno određenim resursima).

Kod *MULTICS* sistema umesto slojeva, koriste se koncetrični prstenovi. Kada neka procedura iz većeg (spoljašnijeg) prstena želi pozvati neki servis nekog manjeg (unutrašnijeg) prstena to će uraditi pomoću poziva koji je sličan sistemkim pozivima.

Virtuelne mašine (virtual machines)

Struktura virtuelnih mašina je izrađena od strane firme *IBM* na sledeći način: imamo na najnižem nivou hardver, iznad hardvera imamo sistem koji se zove **monitor virtuelnih mašina** (**virtual machine monitor**) i koji obezbeđuje niz **virtuelnih mašina** - to nisu proširene mašine kao operativni sistemi koji pružaju niz proširenih i jednostavnijih operacija za pristup hardveru, već su **tačne kopije hardvera** ispod monitora virtuelnih mašina. Zatim se na te virtuelne mašine mogu biti instalirani operativni sistemi – koji mogu biti i različiti. Sistemske pozive korisničkih programa primaju odgovarajući operativni sistemi, a hardverske operacije koje šalju ti OS-ovi prema svojim virtuelnim mašinama hvata monitor virtuelnih mašina i realizuje ih u skladu sa hardverom ispod sebe. Kao primer možemo navesti *IBM*-ov *VM/370*.

aplikacija 1	aplikacija 2	aplikacija 3
operativni sistem 1	operativni sistem 2	operativni sistem 3
monitor virtuelnih mašina		
HARDVER		

Klijent-server model (client-server model)

U ovom modelu, procese delimo na dve vrste: **klijent procesi** (**client process**) predstavljaju procese korisnika i **server procesi** (**server process**) koji su zaduženi za obradu klijentskih procesa. I klijentni i serverski procesi rade u korisničkom režimu rada. **Kernel** OS-a je zadužen za komunikaciju između klijentskih i serverskih procesa. Server procesi su delovi OS-a i grupisani su prema tome koje funkcije obavljaju: proces serveri (zaduženi za procese), fajl serveri (zaduženi za fajl sistem), serveri memorije itd. Svi serveri rade u korisničkom režimu, pa nemaju direktan pristup hardveru.

Prednosti ovog modela:

1. OS je razbijen na manje delove (servere) koji se mogu održavati nezavisno
2. ako dođe do greške u nekom serveru, ne mora pasti i ceo sistem
3. pogodan je za pravljenje **distribuiranog sistema**: različiti serveri mogu se izvršavati na različitim računarima (koji su naravno povezani), kada neki klijent proces pošalje neku poruku nekom serverskom procesu, ne treba da vodi računa o tome, kako će stići poruka do servera, to je zadatak kernela. Isto tako, kernel će znati kako da vrati odgovor.

Postoji još jedna stvar kod ovog modela, o čemu treba voditi računa: neke funkcije OS-a ne mogu se ili se teško mogu realizovati u korisničkom režimu rada. Imamo dve mogućnosti da rešimo ovaj problem:

1. imamo neke **kritične server procese** koji se izvršavaju u sistemskom režimu rada i imaju direktan pristup harveru, ali sa ostalim procesima komuniciraju kao što se to radi u ovom modelu

2. neke kritične **mehanizme** ugrađujemo u sam kernel, a pravimo server u korisničkom režimu koji sadrži samo interfejs tih operacija

klijentni proces 1	klijentni proces 2	server procesa	fajl server	.....	server memorije
KERNEL					

SINHRONIZACIJA PROCESA

Kritična oblast (critical section)

Posmatrajmo sledeći primer: imamo dva terminala A i B. Na oba terminala po jedan proces. Ti procesi vode računa o broju pritiskanja tipke ENTER. Taj broj se čuva u zajedničkoj promenljivoj *brojač*. Program tih procesa je dat pomoću sledećeg pseudo-koda:

```
ako se pritisne ENTER:
     1. uzmi vrednost broja
     2. povećaj vrednost za 1
     3. vrati novu vrednost u brojač
```

Posmatrajmo sada sledeću situaciju: neka je brojač=5, i neka se na terminalu A pritisne ENTER, proces A uzima vrednost *brojača* i povećava je za jedan – u tom trenutku se zbog nečega dođe do zastoja u izvršavanju procesa A. Sada, dok proces A čeka, neka se pritisne ENTER na drugom terminalu i neka proces B izvrši sva tri koraka – tada će vrednost *brojača* biti 6. Neka sada proces A nastavi sa radom. Znači on je primio za vrednost *brojača* 5, povećao je za jedan i postavio vrednost *brojača* na 6. Dva puta je pritisnut ENTER, a *brojač* ima vrednost 6!!

Ovu grešku možemo izbeći tako što ćemo zabraniti procesu B (A) da uđe u deo gde pristupa *brojaču* ako je drugi proces već ušao u taj deo. Znači, proces B će da sačeka da proces A završi svoj rad sa zajedničkom promenljivom i tek će tada početi sa radom.

Kritična oblast (critical section) – je 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. Drugim rečima, proces ne sme biti prekinut dok se nalazi u svojoj kritičnoj oblasti. Kritična oblast se posmatra kao neprekidiv niz operacija (kao jedna primitivna operacija). Određivanje dela programskog koda koji predstavlja kritičnu oblast je zadatak programera.

Kritična oblast može se realizovati :

- softverski
- hardverski
- pomoću operativnog sistema (sistemski pozivi)

Softverska realizacija kritične oblasti

Pretpostavke i osobine softverskog rešenja :

0. Unutar kritične oblasti istovremeno može se nalaziti najviše do jedan proces.
1. Kritična oblast se realizuje softverski bez pomoći operativnog sistema.
2. Prilikom razvoja programa ne uzimamo u obzir pretpostavke ni o brzini, ni o broju procesora.
3. Ni jedan proces koji je izvan svoje kritične oblasti ne sme sprečiti druge procese da uđu u svoje kritične oblasti. Jedino proces unutar kritične oblasti može sprečiti ostale procese da uđu u k.o.
4. Ni jedan proces ne sme neograničeno dugo čekati da uđe u svoju kritičnu oblast. Algoritam mora garantovati svakom procesu mogućnost ulaska u kritičnu oblast.

Sledeći primeri ilustruju probleme softverske realizacije kritične oblasti. Svi programi su dati pomoću pseudo-koda:

```
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.
```

Sadržaj pomoćne promenljive *processNumber* je index onog procesa koji se nalazi u kritičnoj oblasti. *PARBEGIN* i *PAREND* su pseudo-naredbe. Označavaju da procedure koje se pozivaju između njih treba izvršavati paralelno.

Analiza: ovaj algoritam **radi dobro**: istovremeno može biti najviše do jedan proces u svojoj kritičnoj oblasti. **Nedostatak** ovog algoritma je: procesi ulaze u svoje k.o. alternativno, tj u sledećem redosledu: *Process1*, *Process2*, *Process1*, *Process2*,... **Objašnjenje:** pošto smo na početku stavili `processNumber:=1`, sigurno će *Process1* biti prvi, koji će ući u k.o., prilikom izlaska iz k.o., on će staviti `processNumber:=2` i time onemogućiti sebi da uđe u k.o, odnosno dozvoliće da *Process2* uđe u k.o. Neka sada *Process2* uđe u k.o, pa prilikom izlaska stavi `processNumber:=1` i neka sada ovaj proces zaglavi u delu `OstaleStvari2` dovoljno dugo vremena, da *Process1* uđe i izađe iz k.o. Sada je processNumber=2 (to je postavio *Process1*), *Process2* radi nešto što dugo traje u delu `OstaleStvari2` – što znači da nije unutar kritične oblasti, pa bi *Process1* mogao slobodno ući, ali ne može jer je processNumber=2, što znači da mora sačekati da *Process2* završi sa radom i da ponovo uđe i izađe iz svoje kritične oblasti.

```
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 dve logičke promenljive *P1_INSIDE* i *P2_INSIDE* (imaju vrednost *TRUE*, ako su odgovarajući procesi unutar svojih k.o.) rešili smo problem alternativnog izvršavanja, ali smo dobili algoritam koji **ne radi**!! **Objašnjenje:** krenimo od početka: `P1_INSIDE:=FALSE`, `P2_INSIDE:=FALSE.` Pp. da je *Process1* stigao prvi do *WHILE* petlje, on odmah ide i dalje, pošto je P2_INSIDE=FALSE, neka u ovom trenutku i *Process2* dođe do *WHILE* petlje, i on će nastaviti sa radom, pošto je još uvek P1_INSIDE=FALSE – i dobili smo da će oba procesa istovremeno ući u k.o. Znači do problema dolazimo, ako se proces prekine između *WHILE* petlje i reda `Px_INSIDE:=TRUE`.

```
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.
```

Promenljive *p1WantsToEnter* i *p2WantsToEnter* označavaju da odgovarajući proces želi ući u kritičnu oblast.

Analiza: algoritam **ne radi**!! **Objašnjenje:** pp. da se procesi izvršavaju istovremeno – *Process1* će staviti `p1WantsToEnter:=TRUE` a *Process2* stavlja `p2WantsToEnter:=TRUE`. Zatim će oba procesa zaglaviti unutar *WHILE* petlje: *Process1* će čekati na *Process2* i obrnuto – to se zove **zaglavljivanje** (**dead lock**). Oba procesa čekaju na neki događaj koji se nikada neće desiti.

```
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.
```

Analiza: ova verzija zamalo da radi, ali postoji mogućnost greške. Pp. da se procesi izvršavaju paralelno: *Process1* 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.

Dekkerov algoritam

```
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 ;
```

```
      Process2 ;
   PAREND
END Dekker.
```

Analiza: **u čemu je razlika između Dekkerovog algoritma i Verzije 4?** Kod Dekkerovog algoritma uvodi se još jedna pomoćna promenljiva, *favouredProcess*, koja označava koji proces ima prednost (tj. veći prioritet).

Ne mogu oba procesa istovremeno ući u kritičnu oblast (greška druge verzije): pp. da oba procesa istovremeno daju zahtev za ulazak – pošto postoji prioritet procesa, ući će onaj koji ima veći prioritet, a drugi će odustati od ulaska i čekati u *WHILE* petlji dok dobije prednost. Tada drugi proces ulazi u k.o. a prvi će čekati.

Nema zaglavljivanja (greška treće verzije): proces koji nema prednost odustaće od toga da uđe u k.o. – time će omogućiti drugom procesu da uđe.

Nema zaglavljivanja (greška četvrte verzije): pošto prilikom čekanja umesto generatora slučajnih brojeva proveravamo da li smo dobili prednost, ne može doći do zaglavljivanja.

Nema alternacije (greška prve verzije): zato što umesto jedne pomoćne promenljive (*processNumber*) koristimo dve logičke: *p1WantsToEnter*, *p2WantsToEnter*.

Dekkerov algoritam (1965) je dugo bio **najbolje rešenje** za softversku realizaciju sinhronizacije. 1981 **Petersen** objavljuje novo, kraće i lepše rešenje.

Petersenov algoritam

```
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: Petersenov algoritam koristi iste pomoćne procedure kao i Dekkerov algoritam, ali je kod mnogo jednostavniji. Proverimo da li izbegava greške prethodnih algoritama:

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.

Nema zaglavljivanja (greška treće 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.

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 neograničeno dugo ostanu unutar *WHILE* petlje.

Nema alternacije (greška prve verzije): 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*.

Hardverska realizacija kritične oblasti

Kritična oblast može se implementirati i na nivou harvera – ako postoji procesorska instrukcija koja to podržava. Neka je to npr. naredba **TestAndSet (a,b)**, koja radi sledeće tri stvari:

1. čita vrednost logičke promenljive **b**
2. kopira tu vrednost u logičku promenljivu **a**
3. postavlja vrednost promenljive **b** na **TRUE**

Kako je TestAndSet naredba procesora, ona se izvršava kao jedna operacija, tj. ne može biti prekinuta. **Operacija TestAndSet garantuje, da promenljivama a i b istovremeno može pristupiti samo jedan procesor, što znači da se ova operacija ne može biti paralelno izvršena na više**

procesora. (u praksi **a** predstavlja neki registar procesora, a **b** neku memorijsku lokaciju) Sada, kada imamo podršku hardvera za realizaciju kritične oblasti, možemo koristiti sledeći algoritam:

```
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: 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.

Ne mogu oba procesa istovremeno ući u kritičnu oblast (greška druge verzije): 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 znači, da će

Process1 ući u kritičnu oblast. Za sve ostale procese će važiti inside=TRUE, pa će svaka pomoćna lokalna promenljiva uzeti tu vrednost, i ti procesi će i dalje čekati unutar *WHILE* petlje.

Nema zaglavljivanja (greška treće verzije): zbog nedeljivosti naredbe *TestAndSet*.

Nema zaglavljivanja (greška četvrte verzije): ne postoji mogućnost da nakon prvog ulaska u *WHILE* petlju istovremeno bude firstCannotEnter=TRUE i secondCannotEnter=TRUE zbog naredbe TestAndSet i zbog toga što je inside=FALSE.

Nema alternacije (greška prve verzije): 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 će *Process1* jednom ući u *WHILE* petlju, ali će posle instrukcije *TestAndSet* biti firstCannotEnter=FALSE i inside=TRUE, pa će ponovo ući.

Realizacija pomoću sistemskih poziva

Nedostatak i softverske i hardverske realizacije je to što se dolazi do pojave koja se zove **busy waiting** (**zaposleno čekanje**): proces se neprestano vrti unutar WHILE petlje, ne radi ništa korisno, svaki put samo proverava vrednost neke promenljive da bi saznao da li može ući u svoju kritičnu oblast ili ne. Znači i pored toga što dati proces ne radi ništa korisno, troši procesorsko vreme, a pošto znamo da unutar kritične oblasti istovremeno može biti samo jedan proces, možemo zamisliti i situaciju da imamo 1001 procesa od kojih se jedan nalazi unutar k.o, a ostali stoje u jednom mestu, vrte se u krugu unutar *WHILE* petlje. Tako imamo jedan proces koji radi nešto korisno i 1000 procesa koji čekajući na taj jedan troše procesorsko vreme.

Drugi nedostatak prethodnih realizacija je u tome što ne vode računa o prioritetima procesa: može se desiti da najprioritetniji proces uđe u kritičnu oblast, tek posle mnoštva neprioritetnih procesa.

Da bismo izbegli ove nedostatke softverske i hardverske realizacije, koristićemo pomoć operativnog sistema, tj. kritičnu oblast ćemo realizovati pomoću sistemskih poziva. Kod ove realizacije, kada proces čeka da uđe u kritičnu oblast, postaje **blokiran** (**blocked**), sve dok ga neki drugi proces ne odblokira.

Sve varijante sinhronizacije procesa pomoću sistemskih poziva ilustrovaćemo na **PROBLEMU PROIZVOĐAČA I POTROŠAČA (The Producer-Consumer Problem):** imamo jednog proizvođača i jednog potrošača. Proizvođač nešto proizvodi – to je proces koji puni neki bafer. Potrošač troši ono što je proizvođac proizveo – to je proces koji prazni bafer (uzima elemente iz bafera). Proizvođač može da proizvodi dok skladište (bafer) ne postane pun, a potrošač može da troši dok skladište (bafer) ne postane prazan. U tim situacijama, proizvođač odnosno potrošač odlazi na spavanje.

Sleep & WakeUp

Kod ove realizacije koristimo dva sistemska poziva: *Sleep* (spavaj) i *WakeUp* (probudi se). Sistemski poziv **Sleep** blokira onog procesa koji ga je pozvao, a **WakeUp (x)** probuđuje procesa **x** . U slučaju problema proizvođača i potrošača, ove pozive koristimo na sledeći način: Kada proizvođač vidi da nema mesta u baferu, ode da spava (postaje blokiran) pomoću 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 znači da je pre toga bafer bio prazan, pa potrošač spava. Potrošač će probuditi proizvođača 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**!! **Objašnjenje:** posmatrajmo sledeću situaciju: neka je bafer prazan, tada je count=0, neka se izvršava proces potrošača. Potrošač pogleda vrednost *count*-a, vidi da je nula, i uđe unutar *IF* naredbe. Neka se u tom trenutku, potrošač prekida (pre nego što izvrši naredbu *Sleep*) , i procesorsko vreme se daje proizvođaču. Proizvođač proizvodi jedan elemenat, vidi da bafer nije pun, stavi elemenat u bafer i poveća brojač za jedan. Tada primeti da je pre toga bafer bio prazan, pa shvati da je potrošač zaspao (tu je problem: potrošač je bio prekinut pre nego što je uspeo da zaspi) – i pokuša ga probuditi. **Pošto potrošač još ne spava, ta informacija se gubi.** Sada potrošač ponovo dobije procesor i ode da spava – pošto je ubeđen da je bafer prazan. Nakon izvesnog vremena, proizvođač će napuniti bafer do kraja i odlazi na spavanje – čeka potrošača da ga probudi. I tako, i potrošač i proizvođač spavaju do beskonačnosti..., tj. javlja se zaglavljivanje (deadlock).

U čemu je problem? Problem je u korišćenju zajedničke, globalne promenljive *count*!! Da bi sinhronizacija bila uspešna, pristup toj promenljivoj treba da bude unutar kritične oblasti!! (unutar kritične oblasti istovremeno može biti samo jedan proces) – što u ovom slučaju ne važi.

Semafori (semaphores)

Korišćenje semafora za sinhronizaciju procesa predložio je **Dijkstra** (1965). Semafor je apstraktni tip podataka, koji se koristi kao brojač buđenja. Ako je vrednost semafora nula (0), to znači da nije došlo do buđenja, ako je vrednost semafora pozitivan broj, označava broj sačuvanih buđenja. Nad semaforima definišemo dve nedeljive operacije: **Up (S)** i **Down (S)**.

Operacija **Down (S)** radi na sledeći način: uzima vrednost semafora S i proverava da li je ta vrednost veća od nule (u tom slučaju imamo sačuvanih buđenja) – ako jeste, smanjuje vrednost semafora za jedan i vraća kontrolu. Ako je vrednost semafora S jednak nuli, to znači da nemamo više sačuvanih buđenja, proces odlazi na spavanje. Operacija *Down*, može se posmatrati kao generalizacija operacije **Sleep**.

Operacija **Up (S)** radi na sledeći način: ako neki procesi spavaju – čekaju na semaforu S – probuđuje jedan od tih procesa. Izbor procesa je zadatak operativnog sistema (može biti slučajan). U ovom slučaju se vrednost semafora S ne menja, tj. ostaje nula, ali ćemo imati za jedan manji broj spavajućih procesa. Ako je vrednost semafora S veća od nule, tada se samo povećava ta vrednost za jedan. Operacija *Up* može se posmatrati kao generalizacija operacije **WakeUp**.

Važno je zapamtiti da su operacije *Down* i *Up* nedeljive, tj. ako smo počeli sa izvršavanjem, ne možemo biti prekinuti dok ne budemo završili.

Rad ovih naredbi možemo opisati pomoću sledećih pseudo-kodova:

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

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

Pored gore navedenih operacija, možemo definisati i sledeće dve:

- **Init (S,n)** inicijalizuje semafor S sa vrednošću n,
- **Value (S)** vraća vrednost semafora S.

BINARNI SEMAFORI predstavljaju poseban tip semafora koji se inicijalizuju sa 1 (jedan) i koji se koriste za sprečavanje ulaska više od jednog procesa u kritičnu oblast. Ako svaki proces pre ulaska u svoju k.o. pozove *Down* i nakon izlaska pozove *Up*, binarni semafori garantuju **međusobno isključivanje (mutual exlusion)** – zbog toga se ti semafori često označavaju sa imenom **mutex** (od *MUTual EXclusion*). Vrednost binarnih semafora može biti samo 1 ili 0 i imaju sledeće značenje:

0 – neki proces je unutar kritične oblasti

1 – ni jedan proces nije unutar kritične oblasti

Kako do povećavanja *mutex* semafora dolazi samo pri izlasku iz kritične oblasti naredbom Up, ne može dobiti vrednost veću od 1, pošto da bi neki proces izašao iz k.o., mora najpre da uđe, a pri ulasku poziva naredbu *Down*, koja će ili smanjiti za 1 vrednost *mutex*-a ili će (ako je već 0) proces biti poslat na spavanje – a znamo da se *mutex* inicijalizuje sa 1.

Problem proizvođača i potrošača se pomoću semafora rešava na sledeći način:

```
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, označava 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 *mutex*u, 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 proizvođaču, 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 koristimo **dva tipa semafora:**

1. *mutex* – koristimo da bismo obezbedili da unutar kritične oblasti u istom trenutku nalazi najviše do jedan proces
2. *empty,full* – koristimo da bismo obezbedili sinhronizaciju 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.**

Brojači događaja (event counters)

Brojači događaja (event counters, **Reed, Kanodia, 1979**) predstavljaju apstraktni tip podataka, nad kojim se definišu sledeće operacije:

1. **Init (e)** inicijalizuje brojač **e**,
2. **Read (e)** vraća trenutnu vrednost brojača događaja **e**,
3. **Advance (e)** povećava vrednost brojača **e** za jedan
4. **Await (e,v)** čeka dok vrednost brojača **e** ne postane veće ili jednako od **v**

Važno je napomenuti, da se vrednost brojača događaja uvek povećava, nikada se ne smanjuje, ne može se podesiti na željenu vrednost (vrednost b.d. može se samo čitati i povećavati). Inicijalna vrednost brojača je nula.

Realizacija problema proizvođača i potrošača pomoću brojača događaja izgleda na sledeći način:

```
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 ;
```

```
BEGIN
   Init ( in ) ;
   Init ( out ) ;
   PARBEGIN ;
      Producer ;
      Consumer
   PAREND
END ECounters.
```

Analiza: algoritam koristi dva brojača događaja: **in** označava broj ubacivanja u bafer (ukupan broj *ubačenih* elemenata od startovanja programa) , a **out** broj preuzimanja iz bafera (ukupan broj *preuzetih* elemenata od startovanja programa). Pri tome mora važiti da je **in** uvek veći ili jednak od **out**-a (u suprotnom, dobijamo da je preuzeto više elemenata nego što je ubačeno, što je nemoguće), ali razlika između **in**-a i **out**-a ne sme biti veća od **n**, tj. od veličine bafera (u suprotnom, dolazimo do situacije, da smo ubacili više elemenata, nego što bafer može primiti).

Posmatrajmo rad **proizvođača**: nakon što proizvođač proizvede novi elemenat, 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 veći 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 će 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)`.

Posmatrajmo rad **potrošača**: 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.

Monitori (monitors) i uslovne promenljive (condition variables)

Prilikom analize rada algoritma za rešavanja problema proizvođača i potrošača pomoću semafora uočili smo da neispravno korišćenje semafora (loš redosled) može dovesti do zaglavljivanja. Semafori rade savršeno, ali ako ih pogrešno koristimo mogu se javiti ozbiljne greške u radu programa. Znači, bilo bi poželjno naći neki mehanizam koji bi olakšao pisanje korektnih programa. Upravo je to bila pokretačka ideja koja je dovela do razvoja koncepta monitora.

Monitor predstavlja skup procedura, promenljivih, tipova podataka, konstanti itd. koji su grupisani u specijalni modul ili paket. Korišćenje monitora za sinhronizaciju procesa predložili su **Hoare** (1974) i **Brinch Hansen** (1975).

Monitori predstavljaju specijalne module, sa sledećim karakteristikama :

1. procedure iz monitora može istovremeno pozvati najviše do jedan proces, znači ako je proces A pozvao neku proceduru monitora X, tada ni jedan drugi proces ne može pozvati ni jednu proceduru iz tog monitora, dok ta procedura ne završi sa radom,
2. procedure monitora se izvršavaju bez prekida

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.

Programski jezici koji podržavaju monitore: *Modula-2*, *Concurrent Pascal*, *Concurrent Euclid*, itd.

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 sledeće nedeljive operacije (pošto su nedeljive, moraju biti unutar monitora!!):

1. **Init (c)** inicijalizuje uslovnu promenljivu **c**
2. **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.
3. **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 čekao 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 sledeća: proces koji je pozvao *Signal*, mora izaći iz monitora, tj. *Signal* mora biti poslednja naredba odgovarajuće procedure monitora, tako će probuđeni proces biti jedini u monitoru i može nastaviti svoj rad. Mi ćemo koristiti Hansenovo rešenje. **Ako nemamo ni jedan proces koji čeka na signal c, tekući proces nastavlja sa radom a signal se gubi.**
4. **Awaited (c)** vraća logičku vrednost *TRUE*, ako neki proces čeka (spava) na uslovnoj promenljivoj **c**, odnosno *FALSE*, ako takav proces ne postoji

Za demonstraciju korišćenja monitora za rešavanje problema proizvođača i potrošača koristimo programski jezik *Modula-2* (program je zadat u pseudo-kodu) :

```
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.


MODULE Primer ;

FROM Monitor IMPORT EnterItem, RemoveItem ;      // koristimo monitor

   PROCEDURE Producer ;                          // proizvođač
   VAR item : INTEGER ;
   BEGIN
      LOOP
         ProduceItem ( item ) ;
         EnterItem ( item )
      END
   END Producer ;

   PROCEDURE Consumer ;                        // potrošač
   BEGIN
      LOOP
         RemoveItem ( item ) ;
         ConsumeItem ( item )
      END
   END Consumer ;

BEGIN
   PARBEGIN ;
      Producer ;
      Consumer
   PAREND
END Primer.
```

Primedba: radi jednostavnosti, odnos potrošača i proizvođača je realizovan na osnovu *LIFO* (*Last In First Out*) sistema, što znači: onaj elemenat koji je poslednji put ubačen u bafer, biće prvi izbačen (bafer se puni prema gore a prazni se prema dole).

Ekvivalencija sistema za sinhronizaciju procesa

Problem ekvivalencije sistema za sinhronizaciju procesa glasi: da li i kako možemo pomoću datog mehanizma sinhronizacije realizovati neki drugi mehanizam. Pokazaćemo da se monitori mogu biti implementirani na osnovu semafora i obrnuto.

Implementacija monitora korišćenjem semafora

Pretpostavimo da operativni sistem podržava rad sa semaforima. Šta nam je potrebno za realizaciju monitora? Treba obezbediti:

- da procedure iz monitora istovremeno može pozvati najviše do jedan proces
- da se procedure monitora izvršavaju bez prekida
- rad sa uslovnim promenljivama

Problem kritične oblasti (prva dva uslova) rešavamo na sledeći način: **Svakom monitoru dodeljujemo jedan binarni semafor koji se inicijalizuje sa vrednošću 1** (podsetimo se: binarni semafori mogu imati samo vrednost 1 (označava da nema ni jednog procesa unutar kritične oblasti) ili 0 (označava da je neki proces unutar k.o.) , inicijalno uzimaju vrednost 1). Neka se taj semafor zove *mutex*. Njegov zadatak je da kontroliše ulazak u monitor (pozivanje procedura monitora). Kada kompajler vidi da sledi poziv neke procedure iz nekog monitora, on će pre tog poziva ubaciti poziv sledeće procedure :

```
PROCEDURE EnterMonitor ( mutex ) ;
BEGIN
     Down ( mutex )
END EnterMonitor ;
```

Ova procedura garantuje da se procedure datog monitora ne izvršavaju paralelno, naime, ako proces A pozove neku proceduru monitora X, kompajler će automatski ubaciti poziv procedure *EnterMonitor*, pa će *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.

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
3. normalno, tj. neće pozvati *Signal*

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

U trećem slučaju, jednostavno treba pozvati proceduru *LeaveNormally***,** koja će probuditi jednog procesa koji čeka na ulazak u k.o, ako takav proces postoji (i ostaviće *mutex* odgovarajućeg monitora na nuli), a ako takav proces ne postoji povećava vrednost *mutex*a (pošto je mutex binarni semafor, biće mutex=1), što će značiti da nema nikoga unutar k.o.

```
PROCEDURE LeaveNormally ( mutex ) ;
BEGIN
     Up ( mutex )
END LeaveNormally;
```

U slučaju poziva naredbe *Wait (c)*, treba pozvati sledeću proceduru:

```
PROCEDURE Wait ( mutex,c ) ;
BEGIN
     Up ( mutex ) ;
     Down ( c )
END Wait;
```

Naredba `Up ( mutex )` radi sledeće: ako postoji neki proces koji spava na *mutexu*, biće probuđen, ako ne, *mutex* postaje 1, što označava da se ni jedan proces ne nalazi "unutar" odgovarajućeg monitora. Naredbom `Down ( c )` dobijamo da se proces blokira na semaforu c.

U slučaju da se procedura završava naredbom *Signal (c)*, potrebno je pozvati sledeću proceduru:

```
PROCEDURE LeaveWithSignal ( c ) ;
BEGIN
     Up ( c )
END LeaveWithSignal ;
```

Ova procedura probuđuje procesa koji čeka na signal c. **Pažnja**!! **Ovde se pretpostavlja sledeće: iz procedure monitora izlazimo sa operacijom Signal (c), samo ako je to potrebno, tj. samo ako smo sigurni da postoji neki proces koji čeka na taj signal.** U tom slučaju, ne treba nam *Up (mutex)*, tj, ne treba javiti da nema nikoga unutar kritične oblasti, jer je to netačno: kako neko čeka na signal c, to znači da je pozvao proceduru *Wait (mutex,c)* unutar kritične oblasti i taj proces će nastaviti svoj rad unutar k.o., pa mora ostati mutex=0.

Implementacija semafora pomoću monitora

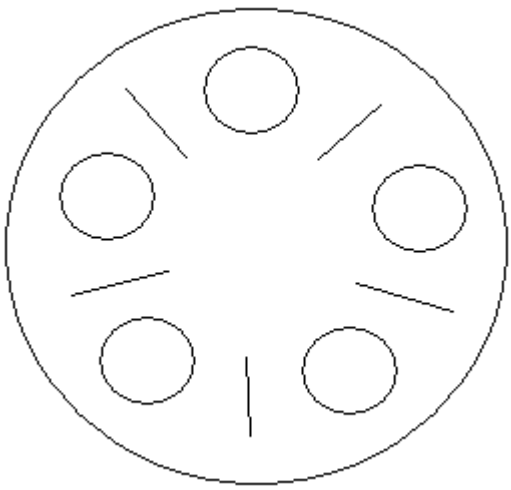
Sada ćemo pretpostaviti da operativni sistem podržava monitore i uslovne promenljive za sinhronizaciju lakih procesa, pa ćemo na osnovu njih implementirati semafore. Ideja je sledeća: za svaki semafor treba nam po jedan brojač buđenja i jedna uslovna promenljiva. Rešenje je dat u obliku pseudo-koda koji se oslanja na karakteristike programskog jezika *Modula-2*:

```
DEFINITION MODULE Semaphores ;          // interfejs modula semafori
TYPE Semaphore ;                        // nedostupni tip podataka
PROCEDURE Init ( VAR s : Semaphore ; v : CARDINAL ) ;
PROCEDURE Down ( s : Semaphore ) ;
PROCEDURE Up ( s : Semaphore ) ;
PROCEDURE Value ( s : Semaphore ) : CARDINAL ;
PROCEDURE Kill ( s : Semaphore ) ;
END Semaphore ;

IMPLEMENTATION MODULE Semaphores [1000] ;

TYPE Semaphore  = POINTER TO SemDesc ;  // Semafor je pokazivač na slog
     SemDesc    = RECORD
                     signal : CondVAR ;       // uslovna promenljiva
                     count : CARDINAL         // brojač
                  END;

   PROCEDURE Init (VAR s : Semaphore ; v : CARDINAL) ;
   // inicijalizacija semafora
   BEGIN
      NEW ( s ) ;                   // alokacija memorije
      s^.count := v ;               // inicijalizacija brojača
      Init ( s^.signal )            // ini. uslovne promenljive
   END Init ;

   PROCEDURE Down ( s : Semaphore ) ;
   BEGIN
      IF s^.count = 0 THEN          // ako je brojač na nuli,
         Wait ( s^.signal )         // čekamo na signal (spavamo)
      ELSE                          // inače, imamo signal, pa
         DEC ( s^.count )           // ne treba spavati, možemo dalje,
      END                           // samo treba javiti da nismo zasp.
   END Down ;

   PROCEDURE Up ( s : Semaphore ) ;
   BEGIN
      IF Awaited ( s^.signal ) THEN         // ako neko čeka na signal,
         Signal ( s^.signal )               // biće probuđen,
      ELSE                                  // ako niko ne čeka,
         INC ( s^.count )               // zapamtimo da smo imali signal
      END
   END Up ;

   PROCEDURE Value ( s : Semaphore ) : CARDINAL ;
   BEGIN
      RETURN s^.count
   END Value ;

   PROCEDURE Kill ( VAR s : Semaphore ) : CARDINAL ;
   BEGIN
      FREE ( s )                    // ako semafor više nije potreban,
   END Kill;                        // treba osloboditi zauzetu mem.

END Semaphores;
```

Klasični problemi sinhronizacije

Problem jedućih filozofa (The Dining Philosophers Problem)

Problem jedućih filozofa potiče od **Dijkstre**, a glasi ovako: pet filozofa sede oko okruglog stola. Ispred svakog filozofa nalazi se po jedan tanjir sa špagetom. Između svaka dva tanjira postavljena je po jedna viljuška. Filozofi su svoj život posvetili razmišljanju. Ako postanu gladni, jedu špagete, pa ponovo razmišljaju. Problem je u tome, što da bi neki filozof mogao jesti, potrebne su mu dve viljuške. Zato kada filozof postane gladan, pogleda, da li može uzeti obe viljuške, ako može – jede (za to vreme njegovi susedi ne mogu jesti), ako ne može, ode na spavanje dok viljuške ne postanu slobodne. Zadatak je sledeći: napisati program koji će rešiti problem ovih filozofa, tj. koji će uspešno sinhronizovati rad 5 procesa koji predstavljaju filozofe.



Očigledno rešenje bi izgledalo na sledeći način:

```
PROCEDURE Philosopher ;
BEGIN
   LOOP
      Think ;                          // razmišljaj
      TakeFork ( leftside ) ;          // uzmi viljušku sa leve strane,
                                       // ako ne možes čekaj
      TakeFork ( rightside ) ;         // uzmi viljušku sa desne strane,
                                       // ako ne možes čekaj
      Eat ;                            // jedi
      PutFork ( leftside ) ;           // vrati levu viljušku
      PutFork ( rightside ) ;          // vrati desnu viljušku
   END
END Philisopher;
```

Ovakvo rešenje neće raditi: pp. da se svi procesi izvršavaju paralelno, tada će svaki filozof uzeti levu viljušku i onda će svaki od njih čekati na svog desnog suseda da mu da desnu viljušku – dolazimo do zaglavljivanja. Pokušajmo sada, sledeće: uzmimo levu viljušku, pa ako desna nije slobodna, vratimo levu čekamo neko vreme, pa ponovo uzmemo levu itd. Nažalost ni ova ideja nam neće pomoći: ponovo pp. da svi procesi rade paralelno, i da će na neki način svi čekati istu količinu vremena, tada opet niko neće jesti – ovaj algoritam će možda raditi, ali može doći do zastoja ako svi generatori slučajnih brojeva generišu iste brojeve, što u nekim slućajevima može dovesti do katastrofalnih rezultata (npr. upravljanje nuklearnim elektranama, saobraćajem ili raketama itd.).

Mogli bismo koristiti jedan binarni semafor, pa staviti *Down(mutex)* ispred `TakeFork(leftside)` i *Up(mutex)* iza `PutFork(rightside)` – takav algoritam bi sigurno radio: najpre pogledamo da li postoji neki filozof koji već jede, ako jeste, čekaćemo da završi ručak, ako niko nejede, red je na nas. Nedostatak je u tome, što će istovremeno jesti samo jedan filozof, tj. koristićemo samo dve viljuške od pet, a mogli bismo četri, tj. mogli bi i dva filozofa jesti istovremeno.

Pravo rešenje korišćenjem semafora izgleda na sledeći način:

```
MODULE Philosophers ;

CONST n = 5 ;                                  // broj filozofa

TYPE  State = ( thinking, hungry, eating ) ;
// stanja filozofa : razmišlja,gladan,jede

VAR mutex : Semaphore ;                        // binarni semafor
    states : ARRAY [1..n] OF State ;           // stanja filozofa
    s : ARRAY [1..n] OF Semaphore ;            // pomoćni semafori

   PROCEDURE Philosopher ( i : CARDINAL ) ;
   // i označava redni broj filozofa
   BEGIN
      LOOP
         Think ;                               // razmišljaj
         TakeForks ( i ) ;                     // uzmi viljuške
         Eat ;                                 // jedi
         PutForks ( i )                        // vrati viljuške
      END
   END Philosopher ;

   PROCEDURE TakeForks ( i : CARDINAL ) ;
   // i označava redni broj filozofa
   BEGIN
      Down ( mutex ) ;                  // ulazak u kritičnu oblast
      states [ i ] := hungry ;          // filozof je gladan
      Test ( i ) ;                      // ako bar jedan od suseda jede,
      Up ( mutex ) ;                    // izlazimo iz kritične oblasti,
      Down ( s[i] )       // ali ćemo čekati na viljušku (možda na obe)
   END TakeForks ;

   PROCEDURE PutForks ( i : CARDINAL ) ;
   // i označava redni broj filozofa
   BEGIN
```

```
      Down ( mutex ) ;                  // ulazak u k.o.
      states [ i ] := thinking ;        // filozof prestaje da jede,
                                        // počinje razmišljati
      Test ( left ( i ) ) ;             // javimo levom filozofu
      Test ( right ( i ) ) ;            // javimo desnom filozofu
      Up ( mutex )                      // izlaz iz k.o.
   END PutForks ;

   PROCEDURE Test ( i : CARDINAL ) ;
   // i označava redni broj filozofa
   BEGIN
      IF ( states [ i ] = hungry ) AND    // ako je naš filozof gladan,
         ( states [ left ( i ) ] <> eating ) AND // a ni levi filozof,
         (states[ right ( i ) ] <> eating) // ni desni filozof ne jede,
         THEN
            states[i] := eating ;    // naš filozof može slobodno jesti
            Up ( s [ i ] )           // pa ga treba probuditi
      END
   END Test ;

BEGIN
   Init ( mutex,0 );
   Init ( s[1],1 ); states[1]:=thinking;
   ...
   Init ( s[n],1 ); states[n]:=thinking;
   PARBEGIN ;
      Philosopher ( 1 ) ;
      ...
      Philosopher ( n ) ;
   PAREND
END Philosophers.
```

Analiza: Posmatrajmo rad programa. Neka neki filozof (A) postane gladan, tada pogleda levog i desnog suseda, ako ni jedan od njih ne jede, to znači da je i leva i desna viljuška slobodna, pa može da jede (to se odvija u proceduri *Test*) i pošalje signal na svoj semafor, da ne bi zaspao pri izlasku iz procedure *TakeForks.* Ako bar jedan od suseda jede, ne može uzeti obe viljuške, pa ne može ni jesti, zato će čekati na svom semaforu – to obezbeđuje *Down* operacija na kraju procedure *TakeForks*. Pretpostavimo sada da je ovaj naš filozof (A) uspeo da uzme obe viljuške i počeo da jede. Neka sada ogladi filozof (B) pored njega. Taj isto javi da je gladan i ode da testira susede. Vidi da filozof (A) pored njega jede, pa ne može uzeti viljušku (filozofi su vrlo miroljubivi, pa neće silom oduzeti viljušku od suseda) zato će da zaspi na svom semaforu. Nakon izvesnog vremena filozof A prestaje da jede, i kaže ostalima: idem ja da razmišljam. Ali pre nego što izađe iz procedure *PutForks*, javi susedima da mu viljuške više nisu potrebne, tako što će pozvati *Test* i za levog i za desnog suseda. Kada pozove *Test* za filozofa B (koji gladan čeka na svom semaforu), uz pretpostavku da ni drugi sused od B ne jede, B će početi da jede (`state[B]:=eating`) i dobija signal na svom semaforu da može izaći iz procedure *TakeForks* i nastaviti svoj rad.

Problem jedućih filozofa je koristan za modeliranje procesa koji se takmiče za ekskluzivan pristup nekim resursima (tj. datim resursima istovremeno može pristupiti samo jedan proces).

Problem čitaoca i pisaca
(The Readers and Writers Problem)

Problem čitaoca i pisaca je koristan za modeliranje pristupa bazi podataka. Imamo neku bazu podataka, nekoliko procesa koji žele upisivati podatke u bazu i nekoliko procesa koji žele čitati podatke iz baze – kako to realizovati a da ne dođemo do problema? Moramo se držati sledećih pravila:

- istovremena čitanja možemo dopustiti jer procesi-čitaoci ne smetaju jedan drugom
- istovremeno čitanje i pisanje nije dozvoljeno, jer možemo doći do toga da proces koji čita pročita pogrešne podatke (proces A počne čitati, nakon nekog vremena dođe proces B i izmeni neke podatke u bazi, proces A pročita pogrešne podatke, dobija loše informacije i pokrene proces samouništavanja svemirske letelice...)
- istovremena pisanja nisu dozvoljena, jer mogu prouzrokovati ozbiljne greške

Mi ćemo ovde posmatrati problem jednog pisca i većeg broja čitalaca. Ideja rešenja je sledeća: kada pisac piše, čitaocima je zabranjem pristup, dok čitaoci čitaju, piscu je zabranjen da piše.

```
MODULE RW ;

VAR mutex : Semaphore ;
   // binarni semafor za kontrolu pristupa globalnoj prom. rc
    db : Semaphore ;
   // binarni semafor za kontrolu pristupa bazi podataka
    rc : CARDINAL ;
   // brojač čitaoca koji istovremeno čitaju iz baze

   PROCEDURE Writer ;       // pisac
   BEGIN
      LOOP
         GetData ;          // sakupi podatke koje želiš upisati u bazu
         Down ( db ) ;      // čekaj ako nemaš pristup bazi
         WriteToDataBase ;  // piši podatke u bazu
         Up ( db )          // javi da si završio sa radom
      END
   END Write ;

   PROCEDURE Reader ;       // čitalac
   BEGIN
      LOOP
         Down ( mutex ) ;   // želimo pristup brojaču
         INC ( rc ) ;       // povećava se broj čitaoca
         IF rc = 1 THEN     // ako smo prvi,
            Down ( db )     // treba tražiti pristup bazi podataka
         END;               // inače ne treba jer je baza već otvorena
         Up ( mutex ) ;     // mogu ostali čitaoci pristupiti rc-u
         ReadFromDataBase ; // čitamo iz baze
         Down ( mutex ) ;   // tražimo ekskluzivni pristup brojaču
         DEC ( rc ) ;       // smanjimo broj čitaoca
         IF rc = 0 THEN     // ako smo mi zadnji,
            Up ( db )       // javimo da može pisač pristupiti bazi
         END;
         Up ( mutex );      // mogu ostali čitaoci pristupiti rc-u
         UseData            // koristimo pročitane podatke
```

```
      END
   END Reader ;

BEGIN
   Init ( mutex,1 ) ;        // niko još nije pristupio rc-u
   Init ( db, 1 ) ;          // niko još nije pristupio bazi
   rc := 0 ;
   PARBEGIN ;
      Writer ;
      Reader ;
      ...
   PAREND
END RW ;
```

Analiza: kada čitalac poželi da čita iz baze podataka, najpre proveri da li može pristupiti globalnoj promenljivoj *rc* koja označava broj onih čitaoca koji istovremeno (u datom trenutku) čitaju iz baze. Ako ne može, čeka na semaforu *mutex*. Ako može, onda gleda: ako je on prvi od čitaoca koji žele čitati iz baze, onda mora zatražiti pristup bazi podataka preko semafora *db*. Ako pisac već piše, onda će čekati na tom semaforu. Ako ne, onda će dopustiti ostalim čitaocima pristup rc-u. Zatim će čitati iz baze. Posle toga ponovo traži pristup rc-u, kada dobije pristup, onda proverava da li je on možda zadnji od čitaoca, ako jeste, onda nikom (iz skupa čitaoca) više nije potrebna baza, pa će javiti da može neko drugi pristupiti.

Znači prvi čitaoc će zatražiti pristup bazi (to istovremeno ne mogu uraditi i više njih zbog semafora *mutex*), kada dobije pristup, baza će biti "otvorena" za sve čitaoce. Poslednji čitaoc ima zadatak da "zatvori" bazu i time omogući piscu (ili možda nekom čitaocu koji ponovo želi čitati) pristup.

Nedostatak ovog rešenja je u tome što favorizuje čitaoce: ako pisac poželi da upiše nešto u bazu, mora sačekati da svi čitaoci odustanu od čitanja, pa se može desiti da jako dugo ne dobije pristup. Ova pojava se zove **izgladnjivanje (starvation).**

Problem uspavanog berberina (The Sleeping Barber Problem)

U frizerskom salonu radi jedan berberin. U prostoriji imamo jednu stolicu za šišanje i n stolica za mušterije koji čekaju da dobiju novu frizuru. Ako nema ni jedne mušterije, berber sedi na svoju stolicu i spava. Kada uđe prva mušterija, budi berberina, sedi na stolicu za šišanje i berber ga šiša. Sledeća mušterija koja dođe, sedi na jednu od tih n stolica za čekanje, ako ima mesta, i čeka svoj red, ako nema mesta odustaje od šišanja.

```
CONST n = 5 ;                  // broj stolica za čekanje

VAR barber : Semaphore ;       // da li je berberin slobodan za šišanje
    customers : Semaphore ;    // da li ima mušterija
    mutex : Semaphore ;        // za kontrolu pristupa promen. waiting
    waiting : CARDINAL ;       // broj mušterija koja čekaju na šišanje

PROCEDURE Barber ;           // berberin
BEGIN
```

```
   LOOP
      Down ( customers ) ;   // ako nema mušterija, berberin spava
      Down ( mutex ) ;       // tražimo pristup globalnoj prom. waiting
      DEC ( waiting ) ;      // smanjimo broj mušterija koji čekaju
      Up ( barber ) ;        // javimo, da šišanje može početi
      Up ( mutex ) ;         // pustimo promenljivu waiting
      CutHair                // šišanje
   END
END Barber ;

PROCEDURE Customer ;        // mušterija
BEGIN
   Down ( mutex ) ;         // tražimo pristup prom. waiting
   IF waiting < n THEN      // ako ima mesta za čekanje,
      INC ( waiting ) ;     // ostajemo, povećamo broj onih koji čekaju
      Up ( customers ) ;    // javimo da ima mušterije
      Up ( mutex ) ;        // pustimo prom. waiting
      Down ( barber ) ;     // čekamo berberina
      GetHairCut            // šišanje
   ELSE                     // ako nema mesta, odustajemo od šišanja i
      Up ( mutex )          // javimo, da nam ne treba prom. waiting
   END
END Customer ;

Inicijalizacija :   Init ( barber,0 ) ;      // berberin spava
                    Init ( customers,0 ) ;   // nema ni jedne mušterije
                    Init ( mutex,1 ); // pristup waiting-u je dozvoljen
                    waiting := 0 ;           // ne čeka niko na šišanje
```

Analiza: Na početku nema mušterije (Init (customers,0 , waiting=0)). Dolazi prva mušterija, vidi da ima mesta (waiting<n), zauzme jedno mesto za čekanje (`INC(waiting)`), pa javi da je stigla (`Up(customers)`), zatim čeka na berberina (`Down(barber)`). Berberin čeka mušteriju (`Down(customers)`). Kada se berberin probudi (`Up(customers)` u proceduri *Customer*), kaže mušteriji da pređe na stolicu za šišanje i smanji brojač za čekanje (`DEC(waiting)`), zatim kaže mušteriji: dobro, ja sam spreman (`Up(barber)`), mušterija to primi na znanje (izlazi iz `Down(barber)`) i počinje šišanje. Ako imamo više mušterija koja čekaju, izbor mušterije koja će preći na stolicu za šišanje je posao operativnog sistema. Ako Petar stigne u frizersku radnju i vidi da nema mesta za čekanje, odustaje od nove frizure (*ELSE* grana) i ode kući da gleda utakmicu.

Komunikacija teških procesa

Kao što smo već rekli, problem komunikacije kod teških procesa je u tome, što su oni međusobno odvojeni, tj. ne dele nijedan deo memorije. Kako nemaju zajedničku memoriju, operativni sistem mora obezbediti neke strukture i mehanizme za **međuprocesnu komunikaciju (interprocess communication)**. Upravo **zbog toga što teški procesi ne dele ništa zajedničko, mehanizmi koje smo koristili za sinhronizaciju lakih procesa ne mogu se primeniti.** Nastaju novi problemi, ako posmatramo mrežu računara, gde je potrebno obezbediti komunikaciju između različitih teških procesa koji se možda izvršavaju na različitim platformama (različiti procesori, različiti operativni sistemi itd.) koji su možda fizički jako udaljeni.

Komunikacija između teških proces se odvija **slanjem poruka (message passing)** i pomoću operativnog sistema. OS treba da vodi računa o sledećim stvarima:

1. Može doći do toga da se poruka negde izgubi u mreži – da bismo izbegli greške koje na ovaj način mogu nastati, mogli bismo usvojiti sledeći dogovor: primilac poruke je dužan da obavesti pošaljioca da je primio poruku. Ako pošaljilac ne dobije potvrdu o prijemu poruke unutar određenog vremenskog intervala, može ponovo poslati svoju poruku.
2. Može doći do toga da poruka stigne na odredište, i primilac pošalje potvrdu ali se sada potvrda negde izgubi i ne stigne do pošaljioca unutar predviđenog vremeskog intervala. Tada će pošaljioc misliti da poruka nije stigla do cilja i poslaće ponovo istu poruku a primilac će dva puta dobiti istu poruku. Zbog toga je dobro da se razne poruke mogu jasno identifikovati, tj. da primilac ima mogućnost da razlikuje novu poruku od duplikata. To se može ostvariti npr. pridruživanjem identifikacionih brojeva.
3. Potrebno je obezbediti jedinstvenu identifikaciju izvora i cilja poruke (treba jedinstveno identifikovati kome je poruka namenjena u mreži: na kom računaru, kom procesu).
4. Problemi u preformansama kada gledamo dva teška procesa koja se izvršavaju na istom računaru a komuniciraju porukama. Sinhronizacija i komunikacija lakih procesa pomoću semafora ili monitora je mnogo brža (laki procesi dele zajednički deo memorije). Komunikacija teških procesa se odvija pomoću operativnog sistema: pošaljioc šalje poruku operativnom sistemu, koji šalje dalje prema primiocu.

Posmatrajmo sada, kako bismo mogli realizovati međuprocesnu komunikaciju zasnovanu na porukama: poruke mogu biti direktno upućeni na jedinstvenu adresu nekog procesa (svaki proces mora imati jedinstveni identifikacioni broj - *PID* (*Process ID*), ovde nastaju komplikacije čak i pri komunikaciji između procesa koji se izvršavaju na istoj mašini, pošto se *PID*ovi mogu menjati (proces A komunucira sa procesom B, korisnik ubije proces B i pokrene neke druge procese, nakon nekog vremena ponovo pokrene proces B koji sada dobija drugi pid)). Drugi način je da se komunikacija odvija preko posebnih struktura podataka. Postoje dve strukture koje se koriste: **mailbox** (**poštansko sanduče**) i **pipe** (**cev**). Razlika između ove dve strukture je u tome, što *mailbox* jasno razlikuje poruke (zna veličinu svake poruke, zna gde je granica između dve poruke) a *pipe* ne vodi računa o granicama poruka: primalac treba znati koliko bajtova treba da preuzme iz *pipe*-a. *Mailbox* se ponaša kao red opsluživanja, tj. kao *FIFO* (*First In First Out*) struktura (prva poruka koja je stigla, biće prva izvađena).

Šta se dešava ako proces želi poslati neku poruku a nema više mesta u mailboxu, ili ako želi da uzme poruku a mailbox je prazan? Imamo dve mogućnosti:

1. Proces se obaveštava da ne može poslati poruku (odnosno da nije stigla ni jedna poruka), pa je na procesu šta će raditi

2. Proces odlazi na spavanje (blokira se) dok ne bude mogao poslati poruku (odnosno, dok tražena poruka ne stigne).

Koliko su veliki mailboxovi odnosno pipeovi? Kapacitet (broj poruka koja *mailbox* može primiti, odnosno veličina *pipe*-a npr. u bajtovima) određuju procesi. Ako *mailbox* može primiti najviše do jednu poruku, govorimo o **randevuu** (**rendevouz**).

Za realizaciju komunikacije između teških procesa potrebno je obezbediti dve operacije:

- **SEND (mailbox, message)** – šalje poruku **message** u poštansko sanduče **mailbox,**
- **RECEIVE (mailbox, message)** – preuzima poruku iz poštanskog sandučeta **mailbox** u **message.**

Korišćenje poruka ćemo demonstrirati na problemu proizvođača i potrošača.

```
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: potrošač i proizvođač su se dogovorili, da će potrošač poslati praznu poruku proizvođaču čim je spreman da primi novu poruku. Na početku rada, potrošač pošalje **n** praznih poruka da bi time obavestio potrošača 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 sledeću poruku.

Dispečer (dispatcher, scheduler)

Setimo se da se procesi mogu nalaziti u jednom od sledećih stanja:

1. **IZVRŠAVA SE (RUNNING)** – kada procesor izvršava kod datog procesa
2. **SPREMAN JE (READY)** – proces je spreman za izvršavanje, čeka da dobije procesora
3. **BLOKIRAN JE (BLOCKED)** – proces čeka na rezultat neke akcije (npr. čeka neku IO operaciju).

Proces prelazi iz stanja IZVRŠAVANJA u stanje BLOKIRAN ako su mu za dalji rad potrebni neki resursi koji trenutačno nisu dostupni – uzrok ovog prelaza je sam proces. Iz stanja BLOKIRAN može preći u stanje SPREMAN, kada dobije sve potrebne resurse za rad i sada treba da čeka na procesora. Iz stanja SPREMAN proces prelazi u stanje IZVRŠAVANJA kada se izabere iz liste spremnih procesa i procesor počinje izvršavati njegov kod. Proces prelazi iz stanja IZVRŠAVANJA u stanje SPREMAN, kada istekne dodeljeno procesorsko vreme.

Prelaze između stanja IZVRŠAVA SE i SPREMAN JE kontroliše deo operativnog sistema, koji se zove **DISPEČER** ili **ŠEDULER**. Znači dispečer je taj koji će odrediti koji proces dobija procesorsko vreme i koliko vremena dobija za izvršavanje. Izvršavanje dispečera mora biti dovoljno brzo, da se procesorsko vreme ne bi trošilo na njega, upravo zbog toga mora biti deo jezgra operativnog sistema i uvek biti u operativnoj memoriji. Do izvršavanja dispečera može doći u sledećim situacijama:

1. proces koji se trenutno izvršava završava svoj rad (tada treba izabrati novi proces za izvršavanje)
2. proces se odblokira (probudi se)
3. proces se blokira
4. procesu u stanju izvršavanja istekne dodeljeno procesorsko vreme

Dodela procesorskog vremena nekom procesu, tj. slanje nekog procesa iz stanja izvršavanja u stanje spreman i izbor drugog procesa za izvršavanje se zove **context switching** (**menjanje okruženja**). U prvom, trećem i četvrtom slučaju sigurno dolazi do menjanja okruženja (ako imamo više od jednog procesa), u drugom ne mora.

Možemo razlikovati dve vrste dispečera:

1. **PREKIDLJIVI (PREEMPTIVE)** – izvršavanje procesa može biti prekinuto od strane dispečera bilo kada. Tada proces prelazi u stanje spreman i njegovo izvršavanje se može nastaviti kasnije.
2. **NEPREKIDLJIVI (NON PREEMTIVE)** – izvršavanje procesa ne može biti prekinuto od strane dispečera, jedino sam proces može menjati svoje stanje (npr. završi svoj rad, postaje blokiran jer čeka na neki događaj ili resurs itd.)

Algoritam dispečera mora uzeti u obzir sledeća očekivanja:

1. **Fernost (fairnest)** – svaki proces mora dobiti šansu za izvršavanje,
2. **Efikasnost** – procesor mora biti maximalno iskorišćen,

3. Treba da obezbedi **kratko vreme odziva** (response time) **interaktivnih** procesa (da ne bi došlo do toga da korisnik pritisne S i čeka pola dana dok se S pojavi na ekranu),
4. Treba da obezbedi **što kraće vreme izvršavanja paketnih** (**batch** - oni koji malo komuniciraju sa korisnikom a više nešto rade) programa (**turnaround time**)
5. Treba da ima š**to veću propusnost** (**throughput**), tj. treba da što veći broj paketnih procesa završi svoj rad u nekoj jedinici vremena.

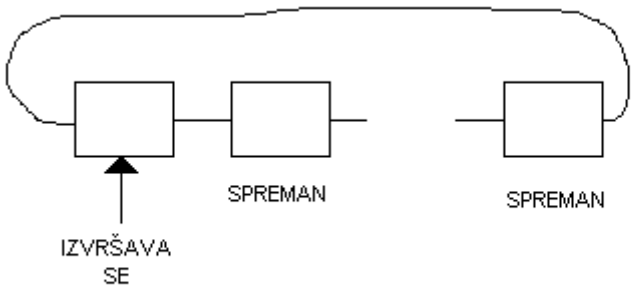
Vrlo je teško naći optimalan algoritam, koji bi ispunio sva prethodna očekivanja. Na primer, ako imamo i interaktivne i paketne procese, treći i četvrti kriterijumi su u koliziji: ako dozvolimo kratko vreme odziva interaktivnih procesa, oni će neprestano prekidati paketne procese, pa će njihovo vreme izvršavanja rasti. Obrnuto, ako pustimo paketne procese da odrade svoj zadatak što brže, interaktivnim procesima ćemo produžiti vreme odziva.

Da bi operativni sistem mogao odlučiti da li je došlo vreme da izvrši menjanje okruženja (tj. da izabere drugi proces za izvršavanje), pomoć stiže već sa hardverskog nivoa. Hardverski sat 50 do 60 puta u sekundi generiše prekid, OS dobija kontrolu. Tada ima mogućnost da odluči, da li je došlo vreme da procesor dobije neki drugi proces.

Sada ćemo razmatrati razne algoritme dispečera.

1. Round Robin algoritam (preemtive, nema prioriteta)

Round Robin je jedan od najstarijih, najjednostavnijih i najčešće korišćenih algoritama. Spremni procesi se stavljaju u kružnu listu, svakom procesu se pridružuje vremenski interval – **kvantum** (**quantum**) vremena. Nakon isteka tog vremena, proces će biti prekinut od strane OS-a a kontrola se predaje sledećem procesu u listi. Ako se proces blokira ili završi rad pre isteka kvantuma, dispečer se aktivira i bira sledeći proces za izvršavanje. Pošto dispečer ide redom po kružnoj listi, ovaj algoritam **garantuje fernost** (svaki proces će dobiti šansu za izvršavanje).



Problemi nastaju pri izboru dužine kvantuma: ako je kvantum previše kratak, dispečer će se izvršavati češće, pa trošimo više procesorskog vremena za prebacivanje sa jednog na drugi proces. Ako je kvantum predugačak, može se desiti da vreme odziva nekih interaktivnih procesa bude prevelik.

Primer1: neka je kvantum 20ms, a neka menjanje procesa traje 5ms. To znači da ćemo za jednu sekundu imati 800ms korisno iskorišćavanje procesora (u vidu izvršavanja procesa) i 200ms potrošeno na dispečer, tj. 20% vremena odlazi na administraciju.

Primer2: neka je kvantum 500ms i neka menjanje procesa traje 5ms. Posmatrajmo sledeću situaciju: imamo višekorisnički sistem sa 10 korisnika od kojih svaki koristi svoj editor texta. Neka svi

tipkaju nešto po tastaturi. Round robin ide okolo, prvi korisnik će rezultat videti odmah, drugi za 505ms, treći za 1010, deseti za 4545ms, tj nakon skoro 5 sekundi čekanja...

2. Najstariji proces (First Come First Served) (non preemtive, nema prioriteta)

Spremni procesi se stavljaju u red opsluživanja, pa se izvršavaju redom i to bez prekida – proces radi dok ne završi svoj rad ili ne postaje blokiran. Izvršavanje dobija uvek najstariji proces. Nedostatak ovog algoritma je u sledećem: uzmimo da imamo jedan proces koji se izvršava jako dugo i iza njega više kraćih procesa. Tada će mnoštvo kratkih poslova dugo čekati zbog tog jednog sporog. Preptostavimo sada, da se taj spori proces zablokira pre nego što završi svoj rad – tada se prebacuje na pocetak reda opsluživanja (čim bude spreman). Neka kratki poslovi završe svoj rad mnogo pre nego što se spori proces odblokira. Tada ćemo imati situaciju da procesor ništa ne radi – čeka na odblokiranje.

3. Prioritetno raspoređivanje (priority scheduling) (preemtive, prioritetni)

Jedan nedostatak Round Robin algoritma je u tome, što su svi procesi ravnopravni, tj. da ne pretpostavlja mogućnost postojanja važnijih i manje važnih procesa. Ideja o pridruživanju prioriteta procesima se uvodi kod prioritetnog raspoređivanja. Prioriteti mogu biti:

1. **statički** – svakom procesu se pridružuje prioritet kada se učita u memoriju
2. **dinamički** – prioritet procesa se menja u toku izvršavanja.

Imamo dva pristupa:

1. procese stavimo u jednu prioritetnu listu i uvek biramo najprioritetniji proces
2. procese grupišemo u prioritetne klase, klasu biramo na osnovu prioriteta a za izbor procesa unutar klase koristimo Round Robin algoritam.

Kao **nedostatak** prioritetnog raspoređivanja javlja se sledeći problem: može se desiti da manje prioritetni procesi nikada ne dođu do procesora – ova pojava se zove **izgladnjivanje (starvation)**. Problem bismo mogli izbeći tako što ćemo prioritet procesa koji se trenutno izvršava prilikom svakog prekida hardverskog sata smanjiti za jedan.

Ako prioritete dinamički određuje OS, dobro je uzeti u obzir sledeća razmatranja:

- procesi koji intenzivno koriste IO uređaje treba da imaju veći prioritet, jer im procesor treba u kraćim intervalima (česće su blokirani).
- procesi koji intenzivno koriste procesor i operativnu memoriju treba da imaju manji prioritet

Objašnjenje: da bismo procesor maksimalno iskoristiti, dobro je pustiti najpre procese iz prve grupe, oni će procesor zauzeti relativno kratko, zatim će se blokirati, pa se mogu slobodno izvršavati procesi iz druge grupe. Ako bi oni imali veći prioritet, procesi iz prve grupe bi dugo čekali da bi na kratko vreme dobili procesor i dugo će zauzimati mesto u operativnoj memoriji. Prioritet procesa mogli bismo računati na sledeći način: neka je veličina kvantuma 100ms. Tada, ako od toga proces koristi procesor u 20ms, dobiće prioritet 100/20=5, ako koristi u 50ms, dobija prioritet 100/50=2, ako koristi 100ms, dobija prioritet 100/100=1. Veći broj označava veći prioritet.

4. Višestruki redovi (multiple queues) (preemtive, prioritetni)

Posmatrajmo sledeću situaciju: imamo sistem sa malo memorije, ali želimo multitasking. Neka menjanje procesa traje 5ms – kada su procesi u memoriji. Pošto sada nemamo dovoljno memorije da svi

spremni procesi budu u memoriji, moraćemo pristupiti disku da bismo dovukli procese, odnosno, da bismo sačuvali trenutno stanje procesa koji se zamenjuje – to će jako povećati vreme administracije.

Ako je kvantum mali, često ćemo pristupati disku, ako je kvantum veliki, vreme odziva se povećava. Rešenje izgleda na sledeći način: procese ćemo grupisati u prioritetne klase (svaka klasa predstavlja jedan red opsluživnja). Procesi iz najprioritetnije klase će dobiti 1 kvantum za izvršavanje, procesi iz druge klase 2 kvantuma, procesi iz treće klase 4 kvantuma, oni iz četvrte klase 8 kvantuma itd. Ako proces iz neke klase iskoristi sve svoje kvantume bez blokiranja, ide u sledeću klasu (dobija više kvantuma). Oni procesi koji nisu interaktivni, tj. intenzivno koriste procesor, dobijaju sve više kvantuma.

klasa	broj kvantuma
1	1
2	2
3	4
4	8
5	16
6	32
7	64

Primer: imamo proces, kome treba 100 kvantuma da završi svoj rad. Nemamo dovojno memorije, pa mora biti prebačen na disk svaki put kad potroši procesorsko vreme. Ako radimo na standardan način (svi procesi dobijaju po jedan kvantum, pa gube procesor), tada da bi ovaj program došao do kraja svog rada, potrebno ga je 100 puta prebaciti sa i na disk. Ako koristimo ovu tehniku, potrebno je samo 7 puta.

Problem: šta ako proces, nakon određenog vremena počinje da bude interaktivan? Jedno rešenje: Svaki put kad se za dati proces pritisne ENTER, ide prema gore po klasama. Rezultat: korisnici shvate da lupajući ENTER dobijaju na prioritetu... i počinje trka – ko može brže pritiskati ENTER...

5. Prvo najkraći posao (Shortest Job First) (non preemtive, nema prioriteta)

Cilj ovog algoritma je da se što više poslova uradi za neko određeno vreme. Radi na sledeći način: od svih spremnih procesa bira onaj, za koji procenjuje da će najbrže završiti svoj rad. Procesi se ne prekidaju (non preempitve). Koristi se pri radu sa paketnim procesima (koji intenzivno koriste procesor a retko komuniciraju sa korisnikom) istog prioriteta. Da bi algoritam funkcionisao, potrebna je procena dužine rada pojedinih procesa.

Primer: imamo 4 procesa : A,B,C,D čije su dužine rada 8,2,3,4 minuta. Pogledajmo dva redosleda izvršavanja:

1. ABCD
2. BCDA (prvo najkraći posao)

Računamo prosečno vreme završetka pojedinih procesa (algoritam je neprekidljiv):

1. A=8, B=10, C=13, D=17, prosečno = (8+10+13+17)/4 = 48/4 = 12
2. B=2, C=5, D=9, A=17, prosečno = (2+5+9+17)/4 = 33/4 = 8.25

U prvom slučaju, za 9 minuta uspemo završiti samo jedan posao (A) , a u drugom slučaju čak tri poslova (B,C,D).

Posmatrajmo sada interaktivne procese. Kod njih se javlja još jedan faktor: korisnik. Da bismo mogli koristiti ovaj algoritam, potrebno je naći neki način za procenjivanje dužine izvršavanja. Na primer: pretpostavimo da procesi funkcionišu na sledeći način: čekaju na komandu, kada se pritisne ENTER (označava kraj komande), izvršavaju komandu. Procenu ćemo vršiti tako što ćemo meriti vreme između dva ENTERA.

Ako je ENTER prvi put pritisnut u trenutku T0, a drugi put u trenutku T1, onda ćemo treći proceniti na osnovu formule **a*T0 + (1-a)*T1**, gde je **a** neka konstanta (obično ½). Ako stavimo a=½ , dobijamo niz:

T0,
T0/2 + T1/2,
T0/4 + T1/4 + T2/2,
T0/8 + T1/8 + T2/4 + T3/4

6. Garantovano raspoređivanje

Prva ideja: imamo n korisnika u sistemu (npr. n terminala na jedan procesor), OS garantuje da će svaki korisnik dobiti 1/n od ukupne snage procesora. Kako to izvesti? Pratimo korisnika i gledamo koliko su svi njegovi procesi vremenski koristili procesor od kada je korisnik počeo svoj rad na terminalu. (npr. korisnik je počeo rad pre dva sata a procesor je od tada radio na njegovim procesima 95 minuta). Sada gledamo, kad je došao prvi korisnik i koliko je od tada utrošeno procesorskog vremena. To vreme delimo sa brojem trenutno ulogovanih korisnika i dobijamo koliko je trebalo da dobiju od procesorskog vremena. Tada ćemo birati procese onog korisnika koji je dobio najmanju pažnju procesora.

Druga ideja: vremenska garancija - vremensko ograničavanje procesa: proces mora biti izvršen za neko određeno vreme.

7. Dvo-nivovski dispečer (two-level scheduling)

Ovaj se algoritam primenjuje kada imamo **malo operativne memorije**, pa se neki od spremnih procesa čuvaju na disku. Radimo na sledeći način: **koristimo dva dispečera.** Najpre učitamo nekoliko spremnih procesa sa diska u memoriju – i sada neko vreme **dispečer nižeg nivoa** radi samo sa tim procesima. Posle određenog vremena dolazi **dispečer višeg nivoa** i kaže: dorbo, ovi procesi su bili dovoljno dugo izvršavani, sada ću ih lepo prebaciti na disk neka se malo odmore, a sa diska ću doneti neke nove procese neka procesor malo izvršava njihove kodove. Dispečer nižeg nivoa to primi na znanje i svu svoju pažnju daje novim procesima.

UPRAVLJANJE MEMORIJOM

Možemo razlikovati 3 vrste memorije :

1. eksterna memorija (najvećeg kapaciteta, najsporiji, najjeftiniji)
2. interna memorija (manjeg kapaciteta, brži, skuplji)
3. keš (cache) memorija (najmanjeg kapaciteta, najbrži, najskuplji)

Nas će najviše interesovati interna memorija (operativna memorija) i njen odnos sa eksternom memorijom. Procesor može izvršavati samo procese koji su u internoj memoriji. Kako je operativna memorija relativno malog kapaciteta, dolazimo do sledećih problema:

1. proces je veći od interne memorije (i kod monoprogramiranja i kod multiprogramiranja)
2. nemamo dovoljno memorije da učitamo sve spremne procese (kod multiprogramiranja)

Deo operativnog sistema koji upravlja korišćenjem memorije zove se **upravljač memorije (memory manager)**. Njegov zadatak je da vodi računa o tome, koji delovi memorije su zauzeti, koji delovi su slobodni, da zauzme potrebnu količinu memorije za nove procese odnosno da oslobodi memoriju zauzetu od strane nekog procesa, i da upravlja prebacivanjem procesa iz interne u eksternu memoriju i obrnuto – pri tome mora voditi računa o tome, da procesi ne štete jedni drugima, a ni operativnom sistemu.

U slučaju kada nemamo dovoljno interne memorije, koriste se sledeće tehnike:

1. **SWAPPING** (prebacivanje procesa) – ako nemamo dovoljno mesta u operativnoj memoriji za smeštanje svih spremnih procesa, neki se izbacuju na disk. Kada je potrebno, celi spremni procesi iz interne memorije se prebacuju na disk, odnosno spremni procesi sa diska se prebacuju u internu memoriju.

2. **PAGING** (straničenje) – delove procesa držimo na disku, učitavaju se po potrebi

Upravljanje memorijom bez swappinga i paginga

Monoprogramiranje (monoprogramming)

Monoprogramiranje znači da istovremeno u internoj memoriji računara može nalaziti samo jedan proces.

Najjednostavniji način je da u internoj memoriji imamo samo jedan proces: pustimo korisnika da učita program koji želi, i taj program tada dobija apsolutnu kontrolu nad računarom (radi šta hoće i gde hoće, pristupa onim delovima memorije kojima želi i koristi one uređaje koje želi i kako želi). Kod ovog pristupa ne treba nam ni operativni sistem, ni posebni drajveri za razne uređaje: program koji se učitava u memoriju je potpuno sam i oslanja se isključivo na sebe. Mora znati sve što mu je potrebno: kako da koristi memoriju, kako da koristi tastaturu, disk, štampač i sve ostalo što mu je potrebno da izvrši svoj zadatak. Kada taj program završi sa radom, može se učitati sledeći program, koji isto tako mora znati da rukuje sa svim uređajima koje koristi. Ova metoda je već davno napuštena, pošto je vrlo nepraktična – programeri moraju ugraditi sve potrebne drajvere u svaki program. Ako nema dovoljno memorije, programer mora naći neko rešenje: optimizuje kod, ili će program razbiti na manje funkcionalne delove koji se mogu samostalno izvršavati ili nešto treće.

Drugi način je da se memorija deli između operativnog sistema i jednog korisničkog programa. Kada se računar uključi, učitava se operativni sistem – bez njega se računar ne može koristiti. Tada na monitoru dobijemo komandni prompt, pa korisnik izdaje naredbe operativnom sistemu. Kaže: izbriši ekran (npr. *CLS* u *DOS*-u) ili izlistaj sve fajlove u tekućem direktorijumu (npr. *DIR* u *DOS*-u), ili učitaj taj i taj program. Ako OS prepozna komandu, pokušava ga izvršiti, ako ne, javlja grešku i čeka sledeću naredbu. Korisnikov program se učitava u preostali deo memorije i tamo se izvršava. Više nije potrebno u svaki program ugraditi i sve potrebne drajvere, jer je OS zadužen da obezbedi sve potrebne funkcije. Imamo više načina za raspodelu memorije:

1. OS se učitava u donji deo *RAM*-a (*Random Access Memory* – memorija koja se može i čitati i isati) , a ostatak memorije je na raspolaganju korisničkim programima
2. OS se nalazi u *ROM*-u (*Read Only Memory* – memorija koja se može samo čitati) na vrhu memorije a korisnički program ispod njega
3. OS se nalazi u donjem delu *RAM*-a, drajveri u *ROM*-u na vrhu memorije a između korisnički program

Osnovni skup funkcija za rad sa uređajima može biti smešten u poseban deo *ROM* memorije koji se zove *BIOS* (*Basic Input Output System*). OS se oslanja na *BIOS*, a korisnički programi mogu pozivati i funkcije OS-a i funkcije *BIOS*-a.

Ni ovde nemamo pravu podršku OS-a za upravljanje memorijom (u slučaju da je proces prevelik, dobićemo poruku o tome, da nemamo dovoljno memorije). Neka razvojna okruženja, pružaju mogućnost da se program razbije na delove, pa se učitava onaj deo koji je potreban za rad (npr. *Turbo Pascal* podržava sličnu tehniku, koja se zove *overlay*: program se deli na razne *unit*-e. Uzmemo skup nekih *unit*-a i pravimo od njih objektni fajl (prevedemo ih u jedan fajl), tokom izvršavanja glavnog programa, ako pozovemo neku funkciju iz nekog od ovih *unit*-a, odgovarajući *unit* se učitava sa diska u memoriju. Pri tome se zauzima onoliko memorije koliko je potrebno za najveći *unit*.)

Multiprogramiranje (multiprogramming)

Multiprogramiranje znači da istovremeno imamo i više spremnih procesa u internoj memoriji.

Neki razlozi za uvođenje multiprogramiranja:

- rad sa više korisnika: imamo računar i nekoliko terminala
- bolje iskorišćavanje procesorskog vremena (dok proces A čeka na podatke sa diska, procesor bi mogao izvršavati proces B koji želi da računa vrednost broja π na što veći broj decimala...)
- jednokorisnički sistemi: dok pišemo u text editoru, hoćemo slušati muziku sa CD-a, a želimo da nam štampač izštampa horoskop koji je u međuvremenu preuzet sa interneta, itd.

Fixne particije

Posmatrajmo drugi način kod monoprogramiranja: memorija se deli na dva dela – jedan deo koristi OS, a drugi deo je namenjen korisničkim programima. Već bismo ovde mogli ostvariti swapping: spremni procesi čekaju na disku, biramo jedan koji se učitava u memoriju i izvršava se neko vreme, zatim se vraća na disk, pa biramo drugi itd. Jedan veliki nedostatak ove realizacije je sporost. Diskovi su mnogo sporiji od memorije, pa ćemo mnogo vremena izgubiti na menjanje procesa.

Proširimo ovu ideju na sledeći način: podelimo memoriju na međusobno nezavisne i odvojene delove – **particije**. U svaku particiju ćemo učitati proces koji je spreman za izvršavanje, taj proces vidi samo svoju particiju i misli, da je sam u memoriji. Particije se određuju nakon startovanja sistema ručno, od strane operatora sistema. **Kako odrediti veličinu particija?** Imamo dva pristupa:

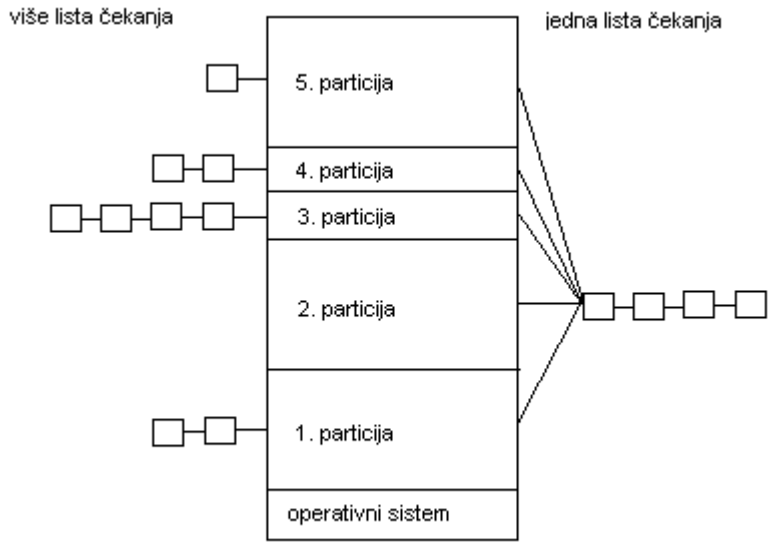
- **neka su sve particije iste veličine** – nedostak: particija je veličine 1MB, a u njoj se izvršava proces veličine 100KB, cela particija je zauzeta, a 90% raspoložive memorije u toj particiji je neiskorišćen
- **neka su particije različith veličina** – imamo manje i veće particije, u svaku particiju stavljamo proces koji najbolje iskorišćava raspoloživu memoriju

Liste čekanja: imamo dve mogućnosti:

1. **svaka particija ima svoju listu čekanja** – novi proces se stavlja u listu najpogodnije particije (gledamo da imamo što manje neiskorišćene memorije). Nedostatak ovog pristupa: može se desiti da na listama čekanja manjih procesa čeka 100 procesa a lista neke veće particije je prazna.
2. **imamo jednu listu čekanja za sve particije** – kada neka particija postaje slobodna, upravljač memorije pregleda listu čekanja i bira sledeći proces. Izbor procesa može se vršiti na dva osnovna načina:
   a.) biramo proces koji je najbliži u redu čekanja i može da stane u datu particiju. Nedostatak: imamo na raspolaganju veliku particiju, ali je mali proces bliže nego veliki, pa će mali proces biti učitan u veliku particiju – gubimo mnogo memorije na mali proces.
   b.) biramo najveći od svih onih procesa koji mogu stati u datu particiju. Nedostatak: mali procesi su diskriminisani, uvek biramo veće procese.

   Rešenja za izbegavanje navedenih nedostataka:

1. definišemo dovoljan broj dovoljno malih particija (u koje mogu stati samo mali procesi a ne i veliki)
2. uvodimo sledeće pravilo: ni jedan proces ne može biti preskočen više od **k** puta. Svaki proces dobija po jedan brojač, svaki put kad se dati proces preskoči, povećava se brojač. Ako pri traženju procesa upravljač memorije vidi da je proces preskočen k puta, učitava ga u particiju, a brojač se resetuje – bez obzira na to da postoji neki proces koji je veći od njega a isto stane u posmatranu particiju. Veličinu broja k određuje sistem operator.



U principu, mogli bismo swapping realizovati i sa fixnim particijama: svaki put kada se neki proces blokira, prebacimo ga na disk, a u tako oslobođenu particiju učitamo drugi spreman proces. Praktično gledano, nećemo dobiti dobre rezultate u iskorišćavanju memorije: možemo imati male procese, koje učitavamo u veće particije, pa gubimo mnogo memorije. To se zove **unutrašnja fragmentacija** (**internal fragmentation**).

Relokacija i zaštita

Relokacija

Particionisanjem memorije dolazimo do situacije, da jedan isti proces može biti učitan i izvršavan u različitim particijama. To dovodi do **problema relokacije**, naime prilikom prevođenja programa **kompajler** (**prevodilac**) generiše objektne fajlove koji sadrže **logičke adrese**. Zatim dolazi **linker** (**povezilac**) i povezuje sve korišćene (i prevedene) module u jedan izvršni fajl. Da bi to uradio, potrebno je da zna početnu **fizičku adresu** gde će program biti učitan i izvršavan. Pošto je to za vreme povezivanja nepoznato, i linker će morati da koristi logičke adrese. Znači sada imamo izvršni fajl sa logičkim adresama. Pošto ne znamo fizičke adrese za vreme prevođenja programa (ne znamo unapred u kojoj će particij program biti učitan), koristimo logičke adrese pa neko treba da izmeni te adrese za vreme učitavanja našeg programa u memoriju. Imamo dve mogućnosti:

1. Adrese će postaviti **punilac (loader)**. Punilac je deo OS-a koji je zadužen da učita program sa diska u memoriju. Kako on već zna, u koju particiju treba učitati, može rešiti i problem relokacije, tako što će izmeniti sve potrebne adrese u programu, tako da se može izvršiti unutar te particije. **Kako će punilac znati koje adrese treba izmeniti?** Tako što će linker

obezbediti potrebne informacije: ubaciće spisak onih mesta u programu gde se nalaze adrese. Nedostatak je slaba zaštita: punilac može proveriti adrese koje su mu poznate zahvaljujući linkera – ali ništa više. Program može izgenerisati adresu koja ne spada unutar svoje particije. Drugi nedostatak je u tome što se program ne može pomerati po memoriji nakon učitavanja a da se sve adrese ne računaju iz početka.

2. **Hardverska podrška:** procesor treba da ima dva posebna registra: **base register** (**bazni registar**) i **limit register** (**granični registar**). Bazni registar čuva početnu adresu particije gde je proces učitan, a granični registar krajnju adresu te particije (ili veličinu particije). Punilac u ovom slučaju ne menja adrese procesa, jednostavno samo učita proces u particiju i postavi vrednost baznog registra. Znači sada u memoriji imamo proces sa logičkim adresama. Svaki put prilikom adresiranja, uzima se logička adresa programa i sabira se sa sadržajem baznog registra i tako se dobija fizička adresa. Hardverska podrška rešava i problem zaštite: kada se izračunava fizička adresa memorije kojoj program želi pristupiti, jednostavno se upoređuje sa sadržajem graničnog registra.
   Korišćenje baznog registra za relokaciju pruža još jednu mogućnost: program se može pomerati po memoriji za vreme izvršavanja - jer su sve adrese logičke, a fizičke se računaju na osnovu baznog registra. Program prebacimo na neku drugu početnu adresu, postavimo sadržaj baznog registra i izvršavanje se može nastaviti.

Napomena: razlika izmedju *EXE* i *COM* fajlova u *DOS*-u je upravo u relokatibilnosti. *EXE* fajlovi su relokatibilni, mogu se učitati počevši od bilo koje memorijske adrese i izvršavati. *COM* fajlovi nisu relokatibilni i ne mogu biti veći od 64KB.

Zaštita

Problem zaštite možemo formulisati ovako: kako možemo sprečiti procesa koji je učitan u particiju A da pristupi sadržaju particije B? Imamo dva rešenja:

1. Memorija se deli na blokove fixne veličine (npr. 1KB, 4KB ili nešto drugo), svakom bloku se dodeljuje 4-bitni zaštitni kod. Proces može pristupati samo onim blokovima, čiji kod odgovara kodu koji je dobio proces. Kodovima upravlja operativni sistem. (Ovo rešenje je napušteno.)
2. Korišćenjem baznog i graničnog registra

Upravljanje memorijom sa swappingom

U slučaju da nemamo dovoljno interne memorije za čuvanje svih spremnih procesa, moramo neke procese izbaciti u eksternu memoriju, a kada je to potrebno, moramo ih vratiti u internu memoriju i nastaviti njihovo izvršavanje. To se zove **swapping**.

Multiprogramiranje sa promenljivim particijama

Da bismo izbegli problem unutrašnje fragmentacije (internal fragmentation) uvodimo promenljive particije: veličina pojedinih particija se određuje na osnovu procesa koji se učita u memoriju. Nemamo unapred određen broj particija sa unapred određenim veličinama. Kada se proces učita, zauzima onoliko memorije koliko mu je potrebno (ako imamo dovoljno memorije). **Razlika između fixnih i promenljivih particija:** kod fixnih particija, broj, veličina, pa i lokacija svih particija je određena unapred pri startovanju sistema od strane sistem operatora. Kod promenljivih particija ne znamo unapred ni broj, ni veličinu ni lokaciju particija – ti parametri se određuju dinamički i zavise od procesa koji se izvršavaju.

Korišćenje promenljivih particija donosi neke druge komplikacije:

Problem **spoljašnje fragmentacije** (**external fragmentation**): posmatrajmo sledeću situaciju: memorija je ispunjena sa procesima (particijama različitih dimenzija). Neka procesi A i B, redom veličine 1MB i 2MB, završe svoj rad, oni se izbacuju iz memorije, na njihovo mesto redom učitamo procese C i D veličine 500KB i 1.5MB. Sada dolazi proces X kome treba 1MB memorije za rad, tačno je da imamo još 1MB slobodnog mesta, ali ne na jednom mestu. Ovaj problem, mogli bismo rešiti tako što ćemo pomerati sve procese tako da se sve slobodne particije nalaze na jednom mestu kontinualno (to se zove **kompakcija - compaction**). Veliki nedostatak je sporost, pa je možda pametnije da proces X čeka dok se ne oslobodi dovoljna količina memorije.

Drugi problem je u sledećem: možemo imati procese koji “rastu”, tj. nakon učitavanja zahtevaju dodatnu memoriju za dinamičke promenljive ili za stek. Imamo više rešenja:

1. Procesi su dužni da prilikom učitavanja u memoriju jave maksimalnu veličinu heapa i stacka.
2. Proces se ubija (prekida) uz poruku o grešci
3. Proces se prebacuje na disk i čeka da se potrebna količina memorije oslobodi

Treći problem je to što treba voditi računa o tome, gde se nalaze zauzete particije i koje su njihove veličine, odnosno gde se nalaze prazne particije i koje su njihove veličine.

Strukture podataka za upravljanje memorijom

Za vođenje evidencije o zauzetosti memorije (o zauzetim particijama) i o rupama između particija (slobodna memorija) možemo koristiti sledeće strukture:

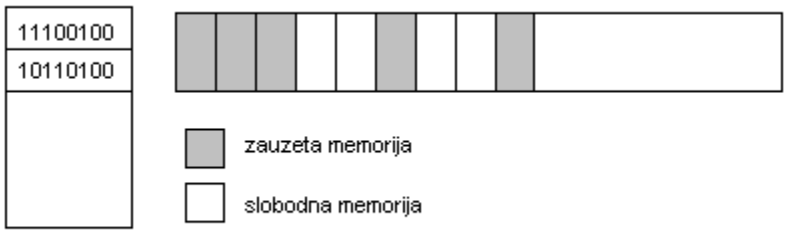
1. Bitne mape (bit maps)
2. Povezane liste (linked lists)
3. Sistem drugova (buddy sistem)

Bitne mape (bit maps)

Memoriju delimo na delove iste veličine. Svakom delu dodeljujemo po jedan bit na sledeći način: 1 označava da je taj deo zauzeto, 0 da je slobodno (ili obrnuto). Tako dolazimo do niza nula i jedinica (bitova) koji se zove bitna mapa memorije.

Pitanje: koje veličine treba da budu ovi delovi memorije ?

Ako je veličina delova 32bita (4 bajta), onda za bitnu mapu koristimo 1/33 od ukupne memorije. Ako su delovi manji, bitna mapa je veća (ako je veličina delova 16bita, onda za bitnu mapu koristimo 1/17 od ukupne memorije: od 512MB ode na administraciju oko 30MB) i traženje slobodne memorije (niz nula) je spora. Ako su delovi veći, kvarimo iskorišćenost memorije: neka su delovi veličine 4KB, ako proces zauzme 1KB od nekog dela, 3KB ostaje neiskorišćen, ali ne može biti alociran od strane drugog procesa jer je formalno zauzeto.



Povezane liste (linked lists)

Memorija je ili zauzeta od strane nekog procesa (P) ili je slobodna (rupa, H – hole). Povezane liste gradimo od slogova sledeće strukture:

P ili H	početna adresa	veličina	sledeći

Prvo polje sloga označava tip memorije: P označava da se radi o procesu, H da se radi o slobodnoj memoriji. Drugo polje sadrži početnu adresu dela memorije koju opisuje dati slog. Treće polje veličinu opisane memorije, a četvrto polje sadrži pokazivač na sledeći slog.

Upravljanje memorijom se odvija na sledeći način:

Zauzimanje memorije: imamo proces, tražimo dovoljno veliku rupu u memoriji. Pretražujemo povezanu listu, tražimo slog tipa H (rupa) dovoljne veličine. Ako nađemo, umesto H upišemo P, i eventualno ubacimo novi čvor tipa H (ako proces ne zauzima celu rupu, ostaje nam neka manja rupa).

Oslobađanje memorije: proces A završi sa radom, oslobodimo zauzetu memoriju. Tada jednostavno stavimo H umesto P ako se i ispred i iza procesa A nalazi neki drugi proces (a ne slobodna memorija), inače vršimo ažuriranje liste:

- ako je ispred procesa A slobodna memorija X a iza proces B, menjamo X i izbacimo A,tj. :

```
X.veličina := X.veličina + A.veličina ;
X.sledeći := A.sledeći ;
Free ( A )
```

- ako je ispred procesa A proces B a iza slobodna memorija X, menjamo A i izbacimo X:

```
A.veličina := A.veličina + X.veličina ;
```

```
A.sledeći := X.sledeći ;
Free ( X )
```

- ako je ispred procesa A slobodna memorija X a iza slobodna memorija Y, menjamo X i izbacimo A i Y:

```
X.veličina := X.veličina + A.veličina + Y.veličina ;
X.sledeći := Y.sledeći ;
Free ( A ) ;
Free ( Y ) ;
```

Sistem drugova (Buddy system)

Upravljač memorije koristi po jednu listu za blokove slobodne memorije veličine 1,2,4,8,16,32,64, itd. bajtova sve do ukupne veličine memorije, po stepenima dvojke. Uzmimo na primer, da imamo 1MB memorije, tada ćemo imati 21 listu (2^0=1 ... 2^20=1MB). Na početku cela memorija je prazna, pa u listi za rupe od 1MB imamo jedan slog, a ostale liste su prazne.

1024KB

Sada dolazi proces A veličine 70KB. Najmanja particija u koju može stati jeste ona od 128KB (mora biti stepen od 2). Na žalost, lista koja sadrži particije te veličine je prazna. Zato ćemo particiju od 1MB razbiti na dva dela od 512KB (ovi delovi se zovu **drugovi – buddy**), pa ćemo prvi razbiti na dva dela od 256KB, i još jednom, prvi delimo na dva dela od 128KB. U prvu particiju stavimo proces:

A **128**	128	256	512

Sada lista od 128Kb sadrži dva čvora (jedan je zauzet – P, a drugi je slobodan – H), a liste od 256KB i 512KB sadrže po jedan slobodan čvor. Ovde već vidimo **nedostatak** ovog sistema: za proces od 70KB zauzimamo čitavu particiju od 128KB, tj. gubimo 58KB memorije (unutrašnja fragmentacija).

Neka sada dođe proces B veličine 35KB. Potrebno je zauzeti particiju veličine 64KB. Lista za takve particije je prazna. Zato delimo prvu slobodnu particiju od 128KB na dva dela od 64KB, u jednu ubacimo proces B a druga ostaje prazna. Sada imamo po jedan čvor u listama za 128KB,256KB i 512KB i dva čvora u listi za 64KB.

A **128**	**B** **64**	64	256	512

Novi proces C od 80KB ubacujemo tako što delimo 256KB na dva dela od 128KB, u prvi ubacimo C a drugi ostaje prazan:

A **128**	**B** **64**	64	**C** **128**	128	512

Posmatrajmo sada stanje memorije: zauzeli smo 128+64+128=320KB za tri procesa ukupne veličine 70+35+80=195KB.

Drugari su susedne rupe iste velicine: A i B (da su rupe) nisu drugovi, ali to jesu npr. B (kao rupa) i rupa pored nje.

Ako neki proces završi sa radom, proverimo veličinu rupe koja nastaje. Ako u susedstvu ima druga, formiramo duplo veću rupu od te dve rupe. Važno je da veličina novoformirane rupe bude neki stepen od 2 – ako to nije slučaj, odustajemo od spajanja.

Algoritmi za izbor prazne particije

Uzećemo povezane liste za strukturu upravljanja memorijom. Pretpostavka je da su liste sortirane na osnovu početnih adresa.

1. **Prvi odgovarajući (FIRST FIT)**: upravljač memorijom pretražuje povezanu listu od početka do kraja i novi proces stavlja u prvu rupu koja je dovoljna velika. Ako je rupa iste veličine kao i sam proces, jednostavno menjamo tip sloga iz H (rupa) u P (proces), ako je rupa veća, menjamo tip iz H na P, postavimo veličinu, zatim iza tog sloga ubacimo još jedan koji će reprezentovati rupu koja preostaje. Ovo je najbolji algoritam.

2. **Sledeći odgovarajući (NEXT FIT)**: radi na isti način kao first fit, samo ne kreće svaki put od početka liste, već od mesta gde je poslednji put stao. Pokazuje slične performanse kao first fit.

3. **Najmanja particija (BEST FIT)**: svaki put pretražuje celu listu i traži najmanju rupu u koju može stati dati proces. Sporiji je od prva dva algoritma jer mora svaki put preći celu listu. Pored toga dolazi do veće spoljašnje segmentacije (gubimo više memorije nego kod prethodna dva algoritma): pošto uvek tražimo najmanju moguću rupu – a mala je verovatnoća da će ta rupa biti po bajtu iste veličine kao proces, pa dobijamo mnoštvo malih rupa koja se ne mogu ni na šta iskoristiti (bez grupisanja na jedno mesto – a to je jako spora operacija i izbegava se).

4. **Najveća particija (WORST FIT)**: radi na isti način kao best fit, samo što uvek traži najveću moguću rupu. Pretpostavka je sledeća: ako proces stavimo u najveću moguću rupu, imamo veću šansu da ostatak bude dovoljno veliko da u njega stane neki drugi proces. Simulacije su pokazale da je worst fit najlošiji algoritam.

Do sada smo pretpostavili da koristimo jednu povezanu listu u kojoj se nalaze procesi i rupe sortirani po početnim adresama. Sada ćemo razmatrati neke moguće varijacije ovih algoritama:

1. **Koristimo dve liste** – jednu listu za procese i jednu listu za rupe. Dobra strana je brzina (ne treba preći i preko procesa). Loša strana je povećana kompleksnost ažuriranja: ako proces postaje rupa mora se izbaciti iz liste za procese a ubaciti u listu za rupe. Ako rupa postaje proces, mora se prebaciti iz liste za rupe u listu za procese – pošto ćemo verovatno dobiti i novu rupu, moramo zauzeti memoriju za novi slog i ubaciti u listu za rupe.
2. Ako koristimo dve liste, mogli bismo **listu za rupe sortirati po veličini** – tada best fit i worst fit veoma brzo rade (postaju efikasni kao i first fit).
3. Koristimo posebnu listu za procese i više lista za rupe. Formiramo liste rupa često korišćenih veličina. Npr. 4KB, 8KB, 20KB itd. Neobične (rupe nepredviđenih veličina) stavljamo u posebnu listu. Ovaj algoritam se zove **quick fit** – dobra strana je brzina, loša strana je komplikovanost ažuriranja (naročito kada proces postaje rupa, pa treba videti da li su susedi rupe radi spajanja u veću rupu).

Virtuelna memorija (virtual memory)

Znamo da procesor može izvršavati samo procese koji su u internoj memoriji. Kako je operativna memorija relativno malog kapaciteta, dolazimo do sledećih problema:

1. proces je veći od interne memorije (i kod monoprogramiranja i kod multiprogramiranja)
2. nemamo dovoljno memorije da učitamo sve spremne procese (kod multiprogramiranja)

Overlay

U slučaju, da je naš program veći od operativne memorije, kao rešenje možemo koristiti tehniku koja se zove **overlay**. Program delimo na međusobno relativno nezavisne delove. Ti delovi će formirati pakete koji se učitavaju po potrebi za vreme izvršavanja, sa diska u memoriju. Inicijalno se zauzima onoliko memorije koliko je potrebno za najveći paket (pored mesta za one delove programa koji stalno moraju biti u memoriji – tj. za pakete overlay-a rezervišemo poseban deo memorije). Zatim, u toku rada programa, kada pozovemo neku proceduru iz nekog paketa koji se nalazi na disku. biće prebačen sa diska u internu memoriju umesto trenutnog paketa. Overlaj paketi mogi biti posebni fajlovi na disku (pored EXE programa i nekoliko OVR fajlova).

Primer: označimo glavni program sa A, a pakete sa B,C,D,E. Naka je glavni program veličine 700KB, a paketi redom veličine 120KB, 200KB, 150KB, 75KB. Neka imamo na raspolaganju 1MB slobodne memorije. Vidimo, da je ukupna veličina našeg programa bez korišćenja ove tehnike skoro 1.3MB što je više nego što imamo. Ako koristimo overlay tehniku, treba nam ukupno 700 + 200 (veličina najvećeg paketa) = 900 KB i naš program može biti izvršen.

Virtuelna memorija

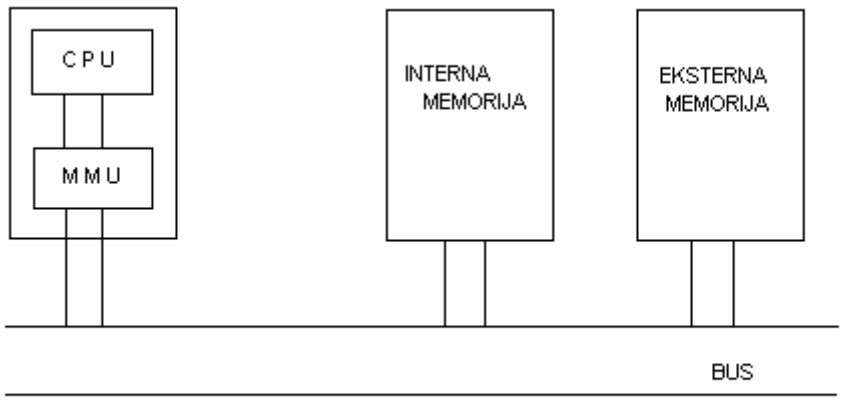
Iako prebacivanje raznih overlay paketa sa diska u memoriju i nazad može biti zadatak OS-a, razbijanje programa na manje, međusobno relativno nezavisnih delova je posao programera. Razbijanje velikih programa u manje delove je mukotrpan i dosadan zadatak. Zato su programeri odlučili da i taj zadatak predaju OS-u. OS treba jednom napisati i onda će on uzeti stvari u ruke, pa ne treba kod pisanja svakog programa razmišljati o razbijanju na delove. Osnovna ideja je da pustimo lepo programera neka napiše program veličine koje želi, i onda će OS voditi računa o tome da u memoriji budu samo oni delovi procesa (i kod i podaci) koji su u datom trenutku potrebni, a ostatak neka čeka na disku. Znači jedan deo procesa je u internoj memoriji a drugi deo je u eksternoj memoriji, a sam proces nema pojma o tome – njemu se čini da ima dovoljno operativne memorije i da se čitav proces nalazi u internoj. Drugačije rečeno: ako nema dovoljno memorije, uzećemo malo od diska – procesi neće znati za to, možda korisnik kada primeti usporavanje rada programa.

Ova ideja može se koristiti i sa multiprogramiranjem: ako imamo 1MB memorije i više procesa veličine 2MB, nekoliko njih će dobiti 256KB za rad, a ostali čekaju na disku (ovde koristimo i

swapping). Ako je procesu A potreban neki deo sa diska, tada - pošto čeka na I/O operaciju – postaje blokiran i kontrola se predaje drugom procesu.

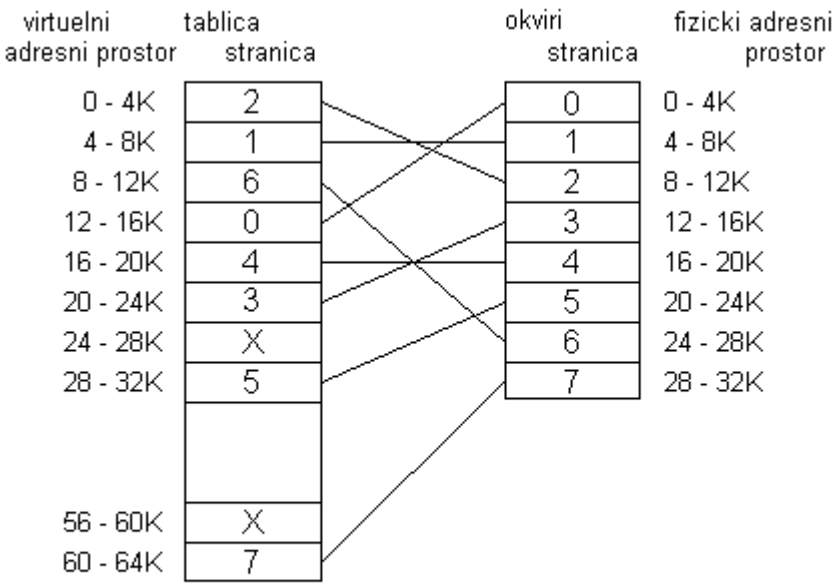
Paging (straničenje)

Paging se koristi kao tehnika za realizaciju virtuelne memorije. Nemamo dovoljno interne memorije da bismo učitali ceo proces, pa koristimo virtuelnu memoriju – memoriju koja je veća od fizičke interne memorije. Proces nema pojma o tome da se izvršava u virtuelnoj memoriji, ubeđen je da je u internoj memoriji – i ponaša se u skladu sa time: pristupa memorijskim lokacijama koje fizički možda ne postoje. Zato je za realizaciju virtuelne memorije potrebna pomoć od strane hardvera. Memorijska adresa generisana od strane programa zove se **virtualna adresa** iz **virtualnog adresnog prostora**. Kod računara bez virtuelne memorije, adresa generisana od strane procesa ide direktno od procesora ka memoriji. Ako koristimo virtuelnu memoriju, virtuelna memorija ide najpre do dela procesora koji se zove **jedinica za upravljanje memorijom (MMU – Memory Managment Unit)**, koja virtuelnu adresu pretvara u fizičku memorijsku adresu:



Virtuelni adresni prostor (virtuelna memorija) se deli na blokove iste veličine. Ti blokovi se nazivaju **virtuelnim stranicama (virtual pages)**. Isto tako internu memoriju delimo na blokove iste veličine. Ti blokovi se zovu **okviri za stranice (page frames)**. Virtuelne stranice i okviri za stranice imaju iste veličine. Računanje fizičke adrese na osnovu virtuelne adrese se radi na osnovu **tabele stranica (page table)** koja predstavlja preslikavanje virtuelnih stranica u fizičke stranice. Ova tabela se dodeljuje svakom procesu i menja se u toku izvršavanja procesa: ako proces želi pristupiti nekoj adresi koja se nalazi na virtuelnoj stranici koja nije učitana u internu memoriju, moramo izbaciti neki okvir za stranice iz operativne memorije (ako je cela operativna memorija već zauzeta) na eksternu memoriju a iz eksterne memorije moramo dovući traženu stranicu na mesto izbačenog okvira za stranice.

Posmatrajmo sada sledeći primer: imamo računar koji radi sa 16-bitnim adresama, što znači da zna adresirati 64KB virtuelne memorije. Pretpostavimo da ovaj računar ima ukupno 32KB operativne memorije. Znači i pored toga da “vidi” 64KB, bez virtuelne memorije ne možemo učitati programe većih od 32KB. Neka su virtuelne stranice (pa i okviri stranica) veličine 4KB. Neka imamo neki proces kome treba 64KB virtuelne memorije i imamo sledeće preslikavanje virtuelnih stranica u okvire za stranice (X označava da data virtualna stranica nije učitana u internu memoriju):



Neka sada proces pokuša pristupiti memoriji na virtuelnoj adresi 0. Centralni procesor (CPU) šalje adresu jedinici za upravljanje memorijom (MMU), koja proverava tablicu stranica i vidi da je data virtuelna stranica (0 – 4K) učitana u drugi okvir (8 – 12K) interne memorije. Zato šalje zahtev dalje internoj memoriji radi pristupa memoriji sa adresom 8192.

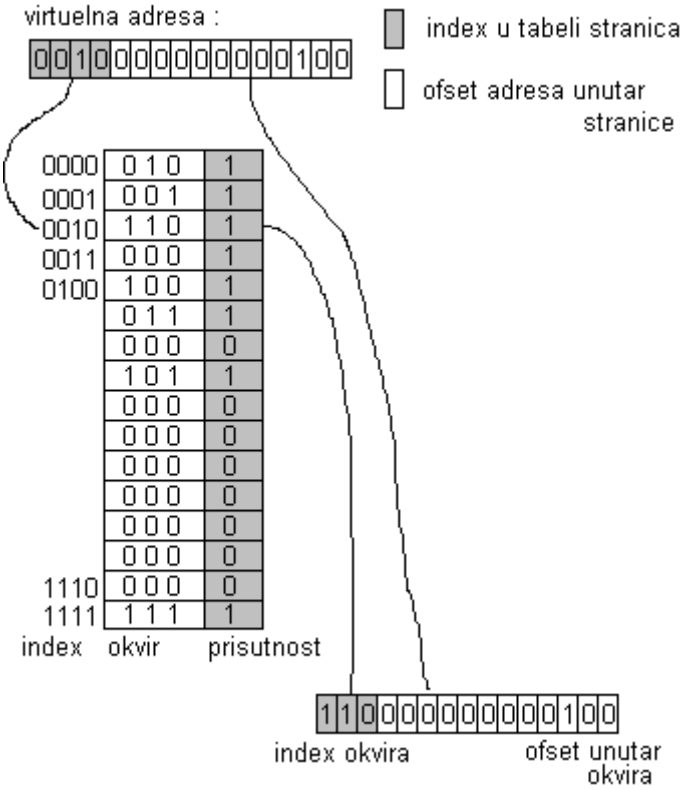
Neka sada proces zatraži pristup memoriji na virtuelnoj adresi 9000. CPU šalje tu adresu MMU-u, MMU pogleda tablicu stranica i vidi da je odgovarajuća virtuelna stranica (8 – 12K) učitana u šesti okvir fizičke memorije (24 – 28K), zato šalje zahtev prema internoj memoriji radi pristupa memoriji sa adresom 25384 (9000 – 8192 = 808, 24576 + 808 = 25384).

Neka u trećem primeru proces pokuša pristupiti virtuelnoj memoriji na adresi 26000. CPU šalje adresu MMU-u, MMU gleda u tablicu stranica, vidi da tražena virtuelna stranica nije učitana u internu memoriju, ne može slati zahtev!! Zbog toga MMU generiše prekid koji se zove **NEPOSTOJEĆA STRANICA (PAGE FAULT)**. Tada kontrolu preuzima operativni sistem. OS najpre izabere jedan okvir iz interne memorije. Sadržaj tog okvira prenosi na disk, a na to mesto učita traženu virtuelnu stranicu, ažurira tabelu stranica vrati brojač instrukcije nazad i pusti neka proces pokuša ponovo.

Kako radi MMU ?

Neka CPU šalje adresu 8196 = **0010**000000000100. Tu adresu delimo na dva dela: 4 najznačanija bita označavaju **index** unutar tabeli stranica, a ostalih 12 bita **ofset** adresu unutar date stranice. Tako možemo imati ukupno 16 virtuelnih stranica (od 0 do 15), a sa 12 bita možemo adresirati ukupno 2^12 = 4096 = 4KB memorije. (Zato smo adresni prostor delili na 4KB stranice.) Tabela stranica sadrži dakle 16 polja. Svako polje će biti predstavljen pomoću 4 bita: 3 najznačajnija bita daju redni broj okvira stranica a najmanje značajni bit informaciju o tome da li je data virtuelna stranica učitana u internu memoriju (1 – jeste, 0 – nije). Taj bit se zove **bit prisutnosti (present/absent bit)**. Fizičku adresu formiramo na sledeći način: na osnovu 4 najznačajnija bita adrese dobijene od CPU-a (0010) , uzmemo index okvira stranice (110) , pogledamo da li je data stranica učitana u internu memoriju (1 – jeste, ako nije, generišemo page fault prekid) i samo iza 110 dodajemo ostatak adrese (000000000100), pa dobijamo **110**000000000100. Prva 3 najznačajnija bita označavaju okvir stranice, a ostalih 12 bitova daju **ofset** adresu unutar tog

okvira. Kako imamo 3 bita za okvire, možemo imati ukupno 8 okvira. Vidimo da je dobijena fizička adresa petnaestnobitna (15), to omogućava adresiranje do 32KB memorije.



Izbor stranica za učitavanje (fetch-strategy)

Imamo dve osnovne strategije:

1. **Učitavanje po potrebi (demand paging)**: učitavamo samo jednu stranicu – onu koju traži MMU. Dobra strana je što u internoj memoriji držimo samo one stranice koje su stvarno potrebne. Loša strana je sporost.

2. **Učitavanje sa predviđanjem (anticipatory paging)**: OS pokuša predvideti, pogoditi koje stranice će biti potrebne za rad procesa i učitava te stranice ako baš nema šta da radi a ni disk nije zauzet. Ako pogodi koje stranice treba učitati, smanjuje se broj page fault-a i izvršavanje procesa dobija na brzinu. Ako ne, od učitavanja nemamo nikakvu korist. Kako cena interne memorije pada a kapacitet raste, ova strategija postaje sve popularnija. (imamo dovoljno memorije da učitamo još neke stranice, pa ako smo pogodili – dobro je, ako nismo, onda ništa).

Izbor okvira za izbacivanje (page replacement algorithms)

Kada dođe do prekida page fault, koji javlja OS-u da MMU nije našao traženu virtuelnu stranicu među okvirima unutar interne memorije, OS treba da odredi u koji će okvir učitati potrebnu stranicu (odnosno, sadržaj kog okvira će izbaciti iz operativne memorije).

Ako je sadržaj izabranog okvira menjan, OS mora ga prebaciti na disk (da bi sačuvao izmene), ako sadržaj nije menjan, onda samo treba učitati novu virtuelnu stranicu.

Sada ćemo razmatrati neke algoritme za izbor okvira koji će biti izbačen iz interne memorije:

1. Optimalni algoritam
2. NRU (okvir koji nismo skoro koristili)
3. FIFO algoritam (okvir koji smo najpre učitali)
4. Druga šansa (varijanta FIFO algoritma)
5. Satni algoritam
6. LRU (okvir koji smo najdavnije koristi)
7. NFU (okvir koji je najređe korišćen – najmanji broj puta)
8. NFU sa starenjem

1. Optimalni algoritam
(Optimal Page Replacement Algorithm)

Kada dođe do page fault-a, optimalni algoritam će truditi da izbaci onu stranicu za koju smatra da će najkasnije biti potrebna. Znači, ako vidimo da će stranica A biti potrebna nakon 100 instrukcija, a stranica B za 1000 instrukcija, onda je bolje da izbacimo stranicu B – time ćemo javljanje sledećeg page faulta odložiti najduže što možemo. MMU ne voli prekidati svoj rad i javljati OS-u da nije našao neku stranicu, zato se trudimo da odložimo prekide što duže.

Ova je najbolja moguća strategija, ali nažalost ne može se implementirati. Ne postoji način da OS dođe do potrebne informacije o tome, kada će koja stranica biti potrebna. (Trebalo bi na neki način predvideti rad procesa.)

2. NRU – Not Recently Used
(okvir koji nismo skoro koristili)

Strategija je sledeća: izbacujemo onu stranicu kojoj nismo skoro pristupili.

Pretpostavimo da je svakom okviru pridruženo dva bita: **R** (referenced) – označava da li smo pristupili (čitali ili pisali) tom okviru, odnosno **M** (modified) – da li smo menjali sadržaj tog okvira. Ako hardver računara ne podržava ove bitove, mogu se implementirati na nivou OS-a.

Pri početku rada procesa svi R-ovi i M-ovi dobijaju vrednost 0. Zatim pri svakom prekidu hardverskog sata OS briše vrednost R-ova (stavlja R=0) – da bi mogao razlikovati stranice koje su skoro korišćene od onih koje nisu. Ako je R=1, to znači da je datom okviru pristupljeno u zadnjih 20ms (ako hardverski sat generiše prekid svakih 20ms).

Kada učitamo neku stranicu sa eksterne u internu memoriju stavimo M=0 (sadržaj okvira nije menjano). Ako dođe do menjanja sadržaja stranice, stavimo M=1 – to onda označava da ćemo morati sadržaj tog okvira prebaciti na disk (da bismo sačuvali promene) ako za izbacivanje izaberemo baš taj okvir.

Na osnovu vrednosti R i M bitova okvire delimo u klase :

klasa	R	M	opis
0	0	0	okviru nismo pristupili u zadnjih 20ms, sadržaj nije menjan
1	0	1	okviru nismo pristupili u zadnjih 20ms, ali je sadržaj menjan
2	1	0	okviru smo pristupili u zadnjih 20ms, ali nije menjan sadržaj
3	1	1	okviru smo pristupili u zadnjih 20ms, i menjan je sadržaj

NRU algoritam bira proizvoljan okvir iz prve neprazne klase sa manjim brojem. Znači ako možemo, biramo okvir iz klase 0, ako ne, onda iz klase 1, ako ne, onda iz klase 2, i tek na kraju iz klase 3.

Prednost: jednostavan je za implementaciju i za razumevanje.

Nedostatak: ne pravi dovoljno finu razliku između stranica iste klase (ne vodimo računa o broju pristupa ni o broju menjanja sadržaja okvira).

3. FIFO algoritam – okvir koji smo najpre učitali
(First In First Out Page Replacement Algorithm)

Izbacujemo onu stranicu koju smo najranije učitali. Ovaj algoritam pretpostavlja da ako je neka stranica davno učitana, verovatno je već odradila svoj deo posla i da će u budućnosti manje biti potrebna od stranica koja su stigla posle nje. Nedostatak ovog pristupa je to što ćemo izbaciti stranicu čak i ako je često koristimo – samo zato što je "ostarela". Zbog ovog nedostatka ovaj algoritam se u ovoj verziji ne koristi.

4. Druga šansa – varijanta FIFO algoritma
(Second Chance Page Replamcement Algorithm)

Koristi istu strategiju kao FIFO algoritam uz dodatak da uzima u obzir i korišćenost stranica – time pokušava izbeći nedostatak FIFO rešenja. Kao pomoć, koristimo vrednost R (referenced) bitova (1 ako smo pristupili stranici, 0 ako nismo). Kada treba izbaciti stranicu, uzmemo zadnju iz reda čekanja, pogledamo R bit. Ako je R=0, to znači da je stranica dugo u memoriji a da joj nismo pristupili – dakle, nije potrebna, može se izbaciti i učitati nova. Ako je R=1, to znači da je stranica dugo u memoriji, ali smo je nedavno koristili. Tada ćemo staviti R=0 a stranicu prebacimo na početak reda opsluživanja – gledamo sledeću stranicu. Znači, ako je stranica korišćena dobija još jednu šansu.

Ako su sve stranice korišćene, algoritam će staviti R=0 za svaku, pa će početi ponovo od najstarije stranice. Sada je već R=0, pa će biti izbačena. Algoritam sigurno nalazi okvir za bacenje.

Ako je R=0 za sve stranice, ovaj se algoritam ponaša isto kao i FIFO verzija.

5. Satni algoritam
(Clock Page Replacement Algorithm)

Nadogradnja algoritma druga šansa. Stranice stavljamo u kružnu listu. Pokazivač pokazuje na stranicu koja je najranije učitana. Kada treba izbaciti stranicu, proverimo R bit stranice na koju pokazuje naš pokazivač. Ako je R=0, stranica se izbacuje a na njeno mesto se učitava nova stranica. Ako je R=1, stavimo R=0 i pomeramo pokazivač na sledeću stranicu – tražimo dalje. Na ovaj način izbegavamo stalno pomeranje opisa stranica sa početka reda opsluživanja na kraj (to se javlja kod algoritma druga šansa) – time poboljšavamo performansu.

Nedostatak satnog i prethodnog algoritma je u tome što ne vode računa o tome koliko puta smo pristupili pojedinim stranicama. Satni algoritam je najbolji u skupu koji čine treći,četvrti i peti algoritam.

6. LRU – okvir koji smo najdavnije koristili
(Least Recently Used Page Replacement Algorithm)

Biramo onu stranicu koju smo najdavnije koristili: stranice sa kojima smo radili u bliskoj prošlosti, verovatno će nam trebati i u bliskoj budućnosti. Ovaj algoritam radi jako dobro, ali se teško ostvaruje. Implementacija zahteva podršku od strane hardvera. Imamo nekoliko realizacija:

1. **Pomoću brojača (hardverska podrška)**: procesor ima brojač instrukcija. Kada se računar startuje, brojač se resetuje na nulu. Nakon svake instrukcije povećamo ga za jedan. Svaki okvir ima svoj interni brojač. Svaki put kada pristupimo okviru, prebacimo sadržaj brojača procesora u interni brojač okvira. Kada dođe do page fault prekida, upravljač memorijom pregleda sve interne brojače i bira stranicu čiji je brojač najmanji (to znači da je korišćena najdavnije). Nedostatak: pri svakom pristupu memorije moramo izvršiti plus jedan pristup radi prebacivanja sadržaja glavnog brojača u lokalni.
2. **Pomoću povezane liste (softversko rešenje)**: slično kao kod FIFO algoritma, stranice stavljamo u red opsluživanja. Kada se javlja page fault prekid, biramo prvu iz reda, novu stavljamo na kraj reda. Svaki put kada pristupimo nekoj stranici, ažuriramo listu: stranicu vadimo iz liste i prebacimo na kraj reda. Ova realizacija je jako spora: pri svakom pristupu memoriji moramo ažurirati listu.
3. **Pomoću dvodimenzionalnog niza (matrice – hardverska podrška)**: imamo matricu formata NxN, gde N označava broj okvira interne memorije. Matrica se inicializuje na nulu (nula-matrica). Prilikom pristupa k-tom okviru, k-tu vrstu matrice ispunimo sa jedinicama a k-tu kolonu sa nulama. Tada vrste označavaju mladost odgovarajuće stranice u binarnom zapisu (što je broj veći, stranica je korišćena u bližoj prošlosti). Biramo stranicu sa najmanjim brojem.

Primer: neka imamo 4 okvira, tada nam treba matrica formata 4x4, neka je redosled pristupa sledeći : 1,1,2,3,1,4. Tada ćemo izbaciti drugu stranicu jer je ona najdavnije korišćena.

	1	2	3	4
1	0	1	1	1
2	0	0	0	0
3	0	0	0	0
4	0	0	0	0

	1	2	3	4
1	0	1	1	1
2	0	0	0	0
3	0	0	0	0
4	0	0	0	0

	1	2	3	4
1	0	0	1	1
2	1	0	1	1
3	0	0	0	0
4	0	0	0	0

	1	2	3	4
1	0	0	0	1
2	1	0	0	1
3	1	1	0	1
4	0	0	0	0

	1	2	3	4
1	0	1	1	1
2	0	0	0	1
3	0	1	0	1
4	0	0	0	0

	1	2	3	4
1	0	1	1	0
2	0	0	0	0
3	0	1	0	0
4	1	1	1	0

7. NFU – okvir koji najređe koristimo
(Not Frequently Used Page Replacement Algorithm)

Biramo stranicu koja je najređe (najmanji broj puta) korišćena. Oslanja se na R bit: svaki okvir ima svoj brojač i R bit. R bit označava da li smo pritupili datoj stranici u predhidnih 20ms. Inicijalna vrednost i brojača i R bita je 0. Pri svakom prekidu hardverskog sata OS pogleda sve R-bitove, doda

vrednost unutrašnjim brojačima i resetuje R. Kada pristupimo stranici R bit se setuje. Prilikom page faulta bacimo stranicu čiji brojač ima najmanju vrednost.

Nedostaci:

1. algoritam sve pamti: stranice koje smo nekada često koristili ostaju u memoriji i pored toga što više nisu potrebne.
2. nove stranice kreću od nule, stare imaju prednost, pa ćemo izbaciti tek učitanu stranicu

8. NFU sa starenjem

(Not Frequently Used with aging)

Pokušava izbeći nedostatak NFU algoritma da ne bi sve pamtio – uračunava i starost stranice: OS prilikom svakog prekida hardverskog sata pomera bitove (**shift**uje) unutrašnjeg brojača prema desno (najmanje značajni bit se gubi), a vrednost R bita se postavi na najznačajniju poziciju (prvi bit sa leve strane). Izbacujemo stranicu čiji brojač ima najmanju vrednost.

Uporedimo rad čisto NFU algoritma i NFU algoritma sa starenjem na jednom primeru:

Sa NFU algoritmom:

	brojač	R	brojač	R	brojač
1	1000	1	1001	1	1010
2	0010	0	0010	1	0011
3	1000	0	1000	0	1000
4	0000	1	0001	1	0010

Vidimo da smo nekada treću stranicu često koristili a kako NFU pamti sve, brojač treće stranice ima veću vrednost nego brojač stranice 4 kojoj smo u bliskoj prošlosti pristupili čak dva puta, a ipak će ta biti izbačena.

Sa NFU algoritmom sa starenjem:

	brojač	R	brojač	R	brojač
1	1000	1	1100	1	1110
2	0010	0	0001	1	1000
3	1000	0	0100	0	0010
4	0000	1	1000	1	1100

Ovde već uzimamo u obzir i starost stranice: tačno je da smo treću stranicu nekada možda češće koristili, ali u bliskoj prošlosti ona nam nije bila potrebna, pa će biti izbačena.

Dalji problemi straničenja

Belady-jeva anomalija

Bilo bi logično da sa povećavanjem broja okvira stranica imamo manji broj page fault-a (ako imamo više interne memorije, možemo veći broj virtuelnih stranica učitati, pa očekujemo manji broj page

fault-a). Belady (**Laszlo Belady, IBM**-ov naučnik, 1969) je našao kontraprimer u slučaju FIFO algoritma, pa se ova pojava zove Belady-jeva anomalija.

Neka imamo proces sa pet virtuelnih stranica i neka imamo na raspolaganju najpre 4 a zatim 3 okvira u internoj memoriji. Dobijamo sledeću situaciju:

Sa 3 okvira:

pristup stranici	0	1	2	3	0	1	4	0	1	2	3	4	0	1
0. okvir	0	0	0	3	3	3	4	4	4	4	4	4	0	0
1. okvir		1	1	1	0	0	0	0	0	2	2	2	2	1
2. okvir			2	2	2	1	1	1	1	1	3	3	3	3
page fault	X	X	X	X	X	X	X			X	X		X	X

Ukupan broj page faulta: **11**

Sa 4 okvira:

pristup stranici	0	1	2	3	0	1	4	0	1	2	3	4	0	1
0. okvir	0	0	0	0	0	0	4	4	4	4	3	3	3	3
1. okvir		1	1	1	1	1	1	0	0	0	0	4	4	4
2. okvir			2	2	2	2	2	2	1	1	1	1	0	0
3. okvir				3	3	3	3	3	3	2	2	2	2	1
page fault	X	X	X	X			X	X	X	X	X	X	X	X

Ukupan broj page faulta: **12**

Radni skup (Working Set)

Radni skup procesa čine one virtuelne stranice koje proces trenutno koristi. Ako je celi radni skup u memoriji, proces će raditi bez mnogo page faulta dok ne pređe u neku drugu fazu rada. Ako je radna memorija premala da bismo učitali radni skup procesa, dolazimo do većeg broja page faulta i izvršavanje se usporava.

Ako imamo sistem sa time-sharingom, često je potrebno ceo proces (sve njegove stranice) prebaciti na disk, da bi neki drugi spreman proces dobio CPU za izvršavanje. Pitanje je: šta uraditi pri vraćanju procesa u memoriju. Najjednostavnije rešenje je, ne raditi ništa posebno, ostaviti neka proces generiše svoje page faultove sve dok se ne učitava radni skup. To se zove **učitavanje po zahtevu (demand paging)** i nije baš neko efikasno rešenje – izvršavanje procesa se usporava i gubimo dodatno procesorsko vreme zbog stalnih prekida. Zbog toga mnogi sistemi vode računa o radnim skupovima i ne puštaju rad procesa dok se ne uvere da je učitan u memoriju. To se zove **model radnog skupa (working set model – Denning**, 1970**)**. Učitavanje stranica pre dozvole za rad zove se **prepaging**.

Za implementaciju modela radnog skupa, potrebno je da OS ima mogućnost da vodi računa o tome koje stranice spadaju u taj skup. Jedno rešenje je korišćenje NFU algoritma sa starenjem: OS posmatra **n** (određuje se eksperimentalno) najznačajnijih bitova unutrašnjih brojača i kaže: ako tu imamo setovane bitove (1), stranica pripada radnom skupu, ako nemamo (to znači da neko vreme nismo pristupili toj stranici), ne pripada.

Lokalnost i globalnost

Postavlja se pitanje: kako raspodeliti okvire među procesima? Imamo dva pristupa:

1. **Loakalni pristup**: svaki proces dobija određen broj okvira i taj broj se ne može menjati. Ako dođe do page faulta, onda stranicu za bacanje biramo u skupu dodeljenih okvira. Nedostatak: broj virtuelnih stranica procesa je veća od dodeljenog broja okvira u internoj memoriji, dobar deo memorije je slobodna, ali ne možemo učitati nove stranice bez izbacivanja već učitanih zbog fixnog broja pridruženih okvira. Drugi nedostatak: proces prelazi u neko stanje u kome od pridruženih 100 okvira koristi samo 25 – gubimo mnogo memorije, jer niko drugi ne može pristupiti tim nekorišćenim okvirima.

2. **Globalni pristup**: broj pridruženih okvira nije ograničeno. Memoriju delimo na okvire, svaki proces uzima koliko može. Ako proces A generiše page fault, tada za razliku od lokalnog pristupa, stranicu koja će biti izbačena nećemo tražiti samo u skupu okvira prirdruženih za taj proces, već u skupu svih okvira. Znači može se desiti, da OS odluči da izbaci neki okvir procesa B a na to mesto ubaci traženu stranicu procesa A. Globalni pristup pokazuje bolje rezultate.

Segmentacija (segmentation)

Do sada smo koristili linearni adresni prostor: počinje od nule i ide do nekog maximuma, ceo proces se posmatra kao linearna struktura. Koristili smo tehniku *overlaya* da bismo program razbili na delove i delove učitavali po potrebi da bismo mogli izvršavati programe koji su veći od raspoložive memorije. Zatim smo brigu o razbijanju programa na delove prepustili operativnom sistemu uvođenjem *paginga*. Sve do sada smo smatrali da je adresni prostor procesa linearan. Neki nedostaci ovog pristupa: ako bilo koju proceduru izmenimo, moramo prevoditi ceo program jer je adresni prostor linearan, pa ako izmenimo veličinu te procedure, menjaju se sve adrese iza nje, ili može doći do rasta steka za vreme izvršavanja programa tako da se stek napuni, pa se podaci pišu na deo gde se nalazi kod programa – kako obezbediti efikasnu zaštiti od toga itd.

Ideja segmentacije je da se svakom procesu dodeli više linearnih adresnih prostora koji počinju od nule i idu do nekog maksimuma. Ti prostori ili delovi procesa se nazivaju **segmentima**. Svaki segment predstavlja linearnu memoriju. Segmenti predstavljaju funkcionalne delove programa kao što su: kod programa, stek, heap, ali segment može da sadrži i samo jednu proceduru ili niz ili nešto drugo. Veličina svakog segmenta može se menjati za vreme izvršavanja od nule do nekog maksimuma. Različiti segmenti mogu biti različitih veličina. Pošto svaki segment ima svoj adresni prostor (linearan), različiti segmenti nemaju uticaja jedan na drugi.

Sa uvođenjem segmentacije adresiranje zahteva dve adrese: jednu **segmentnu adresu** koja identifikuje segment i jednu **ofset adresu** koja daje adresu unutar tog segmenta. Znači adrese su oblika (segment, ofset).

Neke pogodnosti segmentacije:

Ako koristimo segmentaciju, i procedure stavimo u posebne segmente, ne moramo kao kod linearnog pristupa svaki put kad izmenimo neku proceduru prevoditi ceo program. Pošto je svaki segment nezavisan i ima svoj linearni adresni prostor, dovoljno je prevesti samo onu proceduru koju smo izmenili.

Pošto segmenti uglavnom sadrže samo jedan tip struktura: npr. samo stek, samo kod procedure, samo neki niz, razni segmenti mogu imati razne tipove zaštite. Npr. segment procedure može se samo izvršavati, u stek smemo i pisati i čitati, elemente niza možemo samo čitati (konstantni niz) itd.

Kako svaki segment ima sopstveni adresni prostor, otvara se mogućnost korišćenja **deljenih, zajedničkih biblioteka (shared library)**. Biblioteke koje koriste skoro svi programi (npr. za grafičko okruženje) nećemo ugraditi u svaki program posebno, već ćemo napraviti poseban segment koji će svi programi deliti. Tako nema potrebe da te biblioteke učitamo u adresni prostor svakog procesa, već je dovoljna jedna kopija u memoriji – svi će to koristiti.

Da bi OS mogao preuzeti brigu o segmentima, potrebno je da ima sledeće informacije: početna adresa segmenta, veličina (dužina) segmenta, tip segmenta, bezbednosni podaci. Ove informacije se čuvaju u tabeli segmenata.

Upoređivanje straničenja i segmentacije:

	STRANIČENJE	SEGMENTACIJA
da li je programer svestan korišćenja date tehnike ?	NE	DA
broj linearnih adresnih prostora	JEDAN	VIŠE
da li adresni prostori mogu biti veći od fizičke memorije ?	DA	DA
da li možemo procedure (kod) i podatke fizički odvojiti i posebno zaštititi ?	NE	DA
da li postoji mogućnost deljenja procedura među različitim procesima	NE	DA
možemo li lako radtiti sa strukturama promenljivih veličina	NE	DA
zašto je tehnika izmišljena ?	da bismo dobili dovoljno veliki linearni adresni prostor čak i ako nemamo dovoljno interne memorije	da bismo omogućili razbijanje programa na logičke celine (procedure, podaci) koje imaju svoje adresne prostore, da bismo omogućili deljenje i zaštitu

Postoje sistemi koji kombinovano koriste straničenje i segmentaciju. Kod takvih sistema koristimo tro-dimenzionalno adresiranje: (adresa segmenta, adresa stranice, ofset adresa) – segmente delimo na stranice.

FAJL SISTEM

Svakom programu je potrebna mogućnost da sačuva neke informacije – neke podatke procesi mogu čuvati unutar svog adresnog prostora (promenljive, konstante). Za neke procese to je dovoljno, za neke je potrebno neko drugo rešenje. Nedostaci čuvanja podataka u internoj memoriji:

- mali kapacitet
- kada proces završi sa radom, podaci se gube
- računar se blokira ili nestane struja – podaci se gube
- često je potrebno da istim podacima istovremeno pristupi više procesa

Zbog ovih nedostataka, postoji potreba da se neki podaci čuvaju odvojeno od adresnog prostora u eksternoj memoriji u obliku fajlova. Kriterijumi za dugoročno čuvanje podataka:

- veći kapacitet
- podaci moraju biti sačuvani i nakon što proces završi sa radom, i nakon što se računar isključi
- mogućnost istovremenog pristupa više procesa

Deo operativnog sistema koji je zadužen za rad sa fajlovima zove se **fajl sistem (file system)**. Vodi računa o strukturi, načinu imenovanja, načinu korišćenja, o zaštiti i o implementaciji čitavog fajl sistema. Često se pod pojmom fajl sistem podrazumeva struktura direktorijuma i fajlova.

Fajl sistem sa korisničke tačke gledišta

Fajlovi (files)

Fajlovi predstavljaju apstrakciju: pružaju mogućnost da se podaci čuvaju na disku i da se pročitaju sa diska a da pri tome korisnik ne treba da zna detalje implementacije: kako su i gde su podaci sačuvani ili kako zapravo diskovi rade. Upravo OS ima zadatak da obavi tu apstrakciju, odnosno da sakrije detalje realizacije i da pruži interfejs na višem nivou.

Imenovanje (file naming)

Pravila imenovanja fajlova razlikuje se od sistema do sistema. Uglavnom svi sistemi koriste niz znakova za imenovanje fajlova. Neki sistemi dozvoljavaju i korišćenje specijalnih znakova i brojeva. Neki prave razliku između malih i velikih slova (*UNIX*) a neki ne (*DOS*).

Pod **DOS**-om imena fajlova se sastoje od dva dela: ime fajla (8 karaktera) i tipa (ekstenzije, do 3 karaktera) fajla koji su razdvojeni tačkom: readme.txt. Možemo koristiti mala i velika slova, brojeve i neke specijalne znakove. Ne pravi se razlika između malih i velikih slova.

UNIX dozvoljava imena dužine do 255 karaktera. Ime može biti sastavljeno od proizvoljnog broja delova odvojenih tačkom. Pravi se razlika između velikih i malih slova. Možemo koristiti i brojeve i neke specijalne znakove.

Struktura fajlova (file structure)

Najpoznatije strukture su:

1. **Fajl je niz bajtova.** OS ne vodi računa o tome šta se nalazi u fajlovima, sve što vidi su bajtovi. Korisnički programi su oni koji treba da dodeljuju značenje fajlovima (tj. da znaju kako treba gledati na taj niz bajtova). Ovu realizaciju koriste *DOS*, *UNIX*, *Windows*.
2. **Fajl je niz slogova iste veličine.** Ova realizacija je napuštena.
3. **Fajl u obliku B-stabla**. Koristi se za brzo pretraživanje fajla po zadatom ključu.

Tipovi fajlova (file types)

Mnogi operativni sistemi razlikuju više tipova fajlova:

- **regularni fajlovi**: obični, korisnikovi fajlovi (obični nizovi bajtova): mogu biti *ASCII* ili binarni fajlovi (ovu razliku pravimo mi a ne OS). *ASCII* fajlovi sadrže tekst, a binarni fajlovi bilo šta drugo.
- **direktorijumi**: sistemski fajlovi koji sadrže informacije o strukturi fajl sistema
- **uređajni fajlovi**: pod *Unix*om uređajima pridružujemo fajlove (ako želimo štampati, pišemo u odgovarajući fajl).
- **specijalni fajlovi**: zavise od OS-a. Npr. pod *DOS*-om imamo specijalan fajl koji sadrži ime diska

Način pristupa fajlovima (file access)

- **sekvencijalni pristup**: bajtovima (podacima) u fajlu možemo pristupiti samo redom, tj. da bismo pročitali n-ti bajt, moramo izčitati prethodnih n-1 bajtova. Koristi se na primer kod trake.
- **direktan pristup**: možemo skočiti na bilo koji bajt unutar fajla i izčitati (ili upisati) taj bajt – pomeramo "glavu za čitanje" na odgovarajuće mesto i uradimo ono što želimo.

Atributi fajlova (file attributes)

Svaki fajl ima svoje ime i sadržaj. Pored ovih informacija svi operativni sistemi čuvaju još neke dodatne podatke o fajlovima, kao što su vreme nastanka fajla, veličina fajla itd. Te dodatne podatke zovemo **atributima** fajla. Neki mogući atributi:

- kreator
- vlasnik
- read-only (samo za čitanje)
- system (sistemski fajl – deo OS-a)
- hidden (sakriven fajl)

- vreme poslednje modifikacije
- vreme poslednjeg pristupa
- vreme kreiranja
- prava pristupa
- veličina fajla
- maximalna dozvoljena veličina, itd.

Operacije sa fajlovima (file operations)

Svaki OS ima svoj skup sistemskih poziva za rad sa fajlovima. Najčešće operacije su:

CREATE (ime)	- kreiranje novog, praznog fajla
DELETE (ime)	- brisanje fajla
OPEN (ime)	- otvaranje (pripremanje) fajla za rad, operacija vraća jedinstveni deskriptor fajla (file descriptor – fd)
CLOSE (fd)	- zatvaranje fajla koji je prethodno otvoren
READ (fd)	- čita jedan bajt ili više bajtova iz prethodno otvorenog fajla od trenutne pozicije fajl-pokazivaca (file pointer)
WRITE (fd,šta)	- piše jedan ili više bajtova u prethodno otvoren fajl od trenutne pozicije pokazivača. Ako je pokazivač na kraju fajla, podaci se dodaju na kraj.
APPEND (fd,šta)	- isto kao WRITE, sa razlikom da se novi podaci dodaju na kraj fajla
SEEK (fd,pos)	- pomera pokazivač fajla na zadatu poziciju
GETATTRIBUTES (ime)	- vraća atribute fajla
SETATTRIBUTES (ime,atr)	- postavlja (menja) atribute fajla
RENAME (ime,novoime)	- menja ime fajla

Direktorijumi (directories)

Direktorijumi su specijalni fajlovi koji sadrže spisak fajlova unutar tog direktorijuma. Svakom fajlu odgovara po jedan slog koji sadrži ime fajla i još neke dodatne informacije (atributi, gde se fajl nalazi na disku itd.). Unutar istog direktorijuma ne možemo imati dva (ili više) fajla sa istom imenom. Na osnovu dozvoljenog broja direktorijuma možemo razlikovati sledeće fajl sisteme:

1. Podržavaju samo **jedan direktorijum**: imamo samo jedan direktorijum, sve programe i sve fajlove stavljamo u taj direktorijum – nepraktično rešenje iz više razloga: teško ćemo odvojiti fajlove različitih programa, ako imamo mnogo korisnika javljaju se teškoće oko imenovanja fajlova.
2. Podržavaju **direktorijume do dubine 2**: imamo glavni direktorijum (**korenski direktorijum - root directory**) i nekoliko poddirektorujuma. Npr. svaki korisnik bi mogao imati sopstveni direktorijum. Time smo izbegli problem konflikta imena fajlova različitih korisnika, ali nismo rešili problem onih korisnika koji žele grupisati svoje fajlove u logičke celine.
3. Podržavaju **hijerarhijsku strukturu direktorijuma u obliku stabla**: svaki direktorijum pored fajlova može sadržati i više poddirektorijuma.

Ako koristimo hijerarhisjku strukturu, u različitim direktorijumima možemo imati istoimene fajlove. Zato je potrebno naći neku tehniku za jedinstvenu identifikaciju fajlova. Ako želimo pristupiti nekoj datoteci, moraćemo zadati i **putanju** (**path**) do te datoteke. Imamo dve mogućnosti:

1. **Apsolutna putanja (absolute path)**: krećemo od korenskog direktorijuma i redom navodimo imena svih direktorijuma na putu do našeg fajla.
2. **Relativna putanja (relative path)**: krećemo od **tekućeg direktorijuma** (**current directory**) i navodimo put do direktorijuma u kojem se nalazi traženi fajl. Mnogi sistemi dodaju svakom folderu (direktorijumu) dva specijalna fajla: **.** (tačka – reprezentuje tekući folder) i **..** (dve tačke – predstavlja nadređeni (roditelj) direktorijum).

Primer:
apsolutna putanja: **Programi\Posao\kalk.exe**
relativna putanja: **..\Posao\kalk.exe** (ako je tekući direktorijum npr. Programi\Zabava)

Operacije sa direktorijumima

Slično kao i kod fajlova, svaki OS ima svoj skup sistemskih poziva za rad sa folderima. Neke moguće operacije:

CREATE	- kreiranje novog (praznog) direktorijuma
DELETE	- brisanje praznog dirktorijuma
OPENDIR	- otvaranje fajla-direktorijuma za čitanje
CLOSEDIR	- zatvaranje fajla-direktorijuma
READDIR	- čitanje sadržaja fajla-direktorijuma
RENAME	- reimenovanje direktorijuma
LINK	- pod *UNIXOM*: ako želimo da jedan te isti fajl bude u više direktorijuma, nema potrebe da ceo fajl prekopiramo u svaki direktorijum, umesto toga u jedan ćemo staviti sam fajl a u ostale samo pokazivač na taj fajl. Taj pokazivač se zove link (veza). Kada pristupamo linku, pristupićemo fajlu na koji link pokazuje.
UNLINK	- briše link na fajl

Realizacija fajl sistema

Imamo dva pristupa čuvanju fajlova na disku :

- fajl od n bajtova se čuva kao kontinualan niz bajtova – nedostatak: ako se veličina fajla povećava, najverovatnije ćemo morati ceo fajl premestiti na neko drugo mesto. Ova strategija je napuštena.
- fajlovi se razbijaju u blokove fiksne veličine, koji ne moraju biti kontinualno smešteni na disk

Posmatrajmo realizaciju sa blokovima: fajlovi se čuvaju u blokovima fiksne veličine. Npr. jedan blok je veličine 4KB, veličina fajla je 9KB. Tada ćemo zauzeti 3 bloka za taj fajl, odnosno 12KB. U poslednjem bloku čuvamo samo 1KB korisne informacije, a ostalih 3KB je izgubljeno – jer je ceo blok zauzet od strane našeg fajla. Prednost velikih blokova je brzina a nedostatak je loša iskorišćenost prostora diska. Prednost malih blokova je bolje iskorišćavanje prostora a nedostatak je sporost – pošto koristimo veliki broj malih blokova, da bismo učitali fajl moramo više puta pozicionirati glavu diska.

Standardne veličine blokova su od 512B do 8KB.

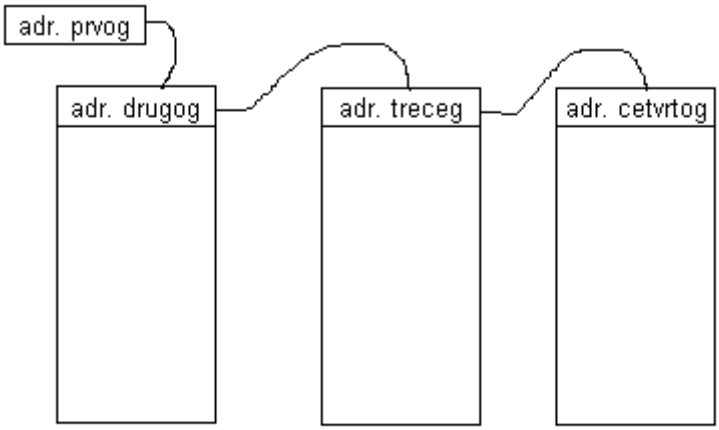
Ako je adresiranje blokova 16-bitna, možemo razlikovati 2^16 = 65536 različitih blokova. To znači da ćemo videti od 32MB (sa blokovima od 512B) do 512MB.

Neiskorišćeni blokovi

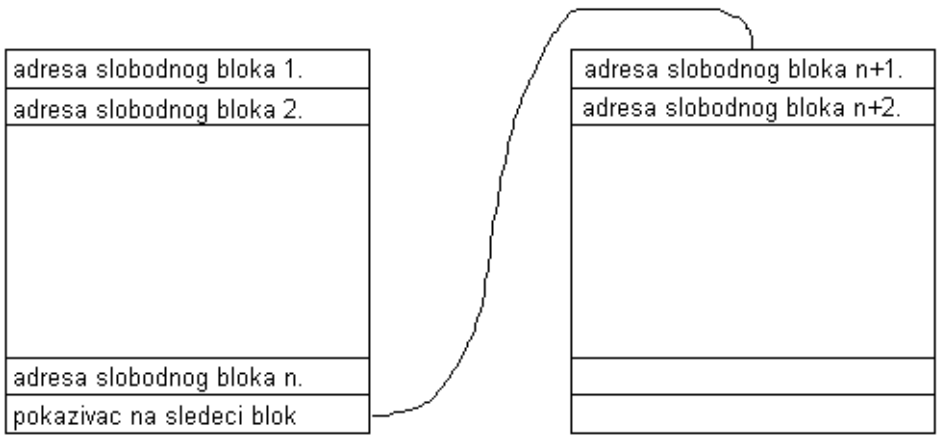
Neka su blokovi veličine 1KB i neka je adresiranje 16-bitna. Imamo dva standardna načina za evidenciju slobodnih blokova:

1. **Pomoću POVEZANE LISTE**: dva pristupa:

    - na početku svakog bloka čuvamo adresu sledećeg praznog bloka. Treba samo zapamtiti adresu prvog bloka, jer ćemo tamo naći adresu drugog itd. Prednost: ne gubimo dodatan prostor za evidenciju slobodnih blokova (blokovi su ionako prazni, pa ništa ne gubimo ako stavimo pokazivač na sledeći blok). Nedostatak: sporost – obilazak liste zahteva neprestano pozicioniranje diska.



- povezanu listu čuvamo u posebnim (praznim) blokovima. Umesto toga da u svakom praznom bloku čuvamo adresu sledećeg, listu praznih blokova smestimo u posebne blokove.Svaki blok sadrži niz adresa praznih blokova, a na kraju svakog bloka imamo adresu sledećeg bloka sa adresama praznih blokova. Prednost: veća brzina obilaska. Neka su blokovi veličine 1KB, neka imamo 16-bitno adresiranje i disk od 20MB. Tada imamo ukupno 20*1024=20480 blokova. Unutar jednog bloka možemo čuvati 511 adresa praznih blokova (+ adresa sledećeg bloka = 512 adresa, 16-bitne (dvobajtne) adrese, pa 512*2=1024), pa nam treba 40 blokova za administraciju. Broj blokova korišćenih za evidenciju se menja sa menjanjem popunjenosti diska.



2. **Pomoću BITNE MAPE:** svakom bloku diska dodeljujemo po jedan bit. Ako je blok zauzet, stavimo 0, ako je prazan stavimo 1 (ili obrnuto). Ako je kapacitet diska n blokova, treba nam n bitova za bitnu mapu diska. Bitnu mapu čuvamo u tačno određenim blokovima diska. Ako je k-ti blok zauzet, onda je vrednost k-tog bita bitne mape 0, a ako je prazan, onda je 1. Ako je veličina blokova 1KB a kapacitet diska 20MB, veličina bitne mape je 20*1024=20480 bitova, odnosno 2560bajtova=2.5KB, tj zauzima 3 bloka diska.

Ako imamo dovoljno memorije da učitamo celu bit-mapu, pretraživanje je brže od pretraživanja povezane liste. Ako nemamo dovoljno memorije (npr. možemo učitati samo jedan blok) – tada je bolje rešenje povezana lista: jedan blok sadrži adresu 511 slobodnog bloka.

Implementacija fajlova (zauzeti blokovi)

1. Kontinualni blokovi

Najednostavniji način je da za svaki fajl zauzmemo onoliko kontinualnih blokova koliko treba. Prednost je laka implementacija: da bismo locirali neki fajl, treba samo zapamtiti prvi blok i veličinu fajla. Druga prednost je brzina pristupa podacima (i sekvencijalno i direktno). Nedostatak je što u većini slučajeva ne znamo unapred maksimalnu veličinu naših fajlova. Ako fajl izraste iz predviđenog broja blokova, moramo pretražiti disk da bismo našli potreban broj kontinualno raspoređenih slobodnih blokova. Ako nađemo, treba prebaciti ceo fajl na novo mesto, ako ne nađemo, možemo pokušati sa grupisanjem slobodnih blokova na jedno mesto (premeštanjem zauzetih blokova). Vrlo neefekasna realizacija. Javlja se problem eksterne fragmentacije.

2. Povezana lista

Blokove zauzetih od strane nekog fajla stavimo u povezanu listu. Treba samo zapamtiti adresu prvog bloka. U svakom bloku čuvamo adresu sledećeg bloka, kao kod evidencije slobodnih blokova pomoću povezane liste na osnovu prvog načina. Prednosti: ako fajl raste, samo zauzmemo bilo koji neiskorišćen blok i ažuriramo povezanu listu – ne javlja se problem eksterne fragmentacije. Nedostaci:

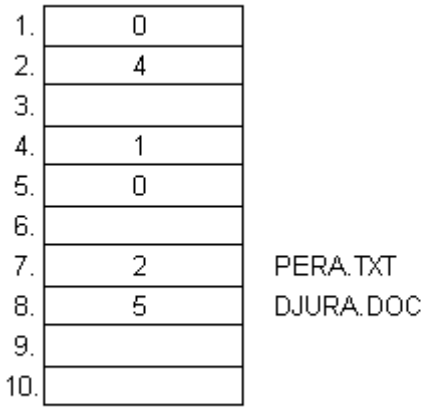
- ne podržava direktan pristup fajlu – ako nam treba neka informacija koja se nalazi u zadnjem bloku, da bismo došli do zadnjeg bloka moramo proći kroz celu listu (moramo posetiti svaki blok da bismo dobili adresu sledećeg bloka).
- veličina korisne informacije u blokovima više nije stepen od 2 (zato što nam treba mesto za adresu sledećeg bloka)

3. FAT (File Allocation Table)

FAT je tabela koja ima onoliko elemenata koliko imamo blokova na disku. Ako se u k-tom bloku nalazi neki fajl, tada k-ti elemenat *FAT*-a sadrži adresu sledećeg bloka koji taj fajl zauzima ili nulu koja označava da je to poslednji blok fajla. Korišćenjem *FAT*-a izbegavamo nedostatke povezane liste: ceo blok možemo iskoristiti za čuvanje korisnih informacija (nije potrebno oduzeti deo za adresu sledećeg bloka), i bolje je podržan direktan pristup sadržaju fajla – direktan pristup ni ovde nije u potpunosti podržan jer ako trebamo pristup 10-tom bloku fajla, moramo pomoću *FAT*-a naći adresu tog bloka, ali je pretraživanje mnogo brže: informacije su na jednom mestu, ne treba obilaziti svaki blok da bismo došli do adrese sledećeg bloka.

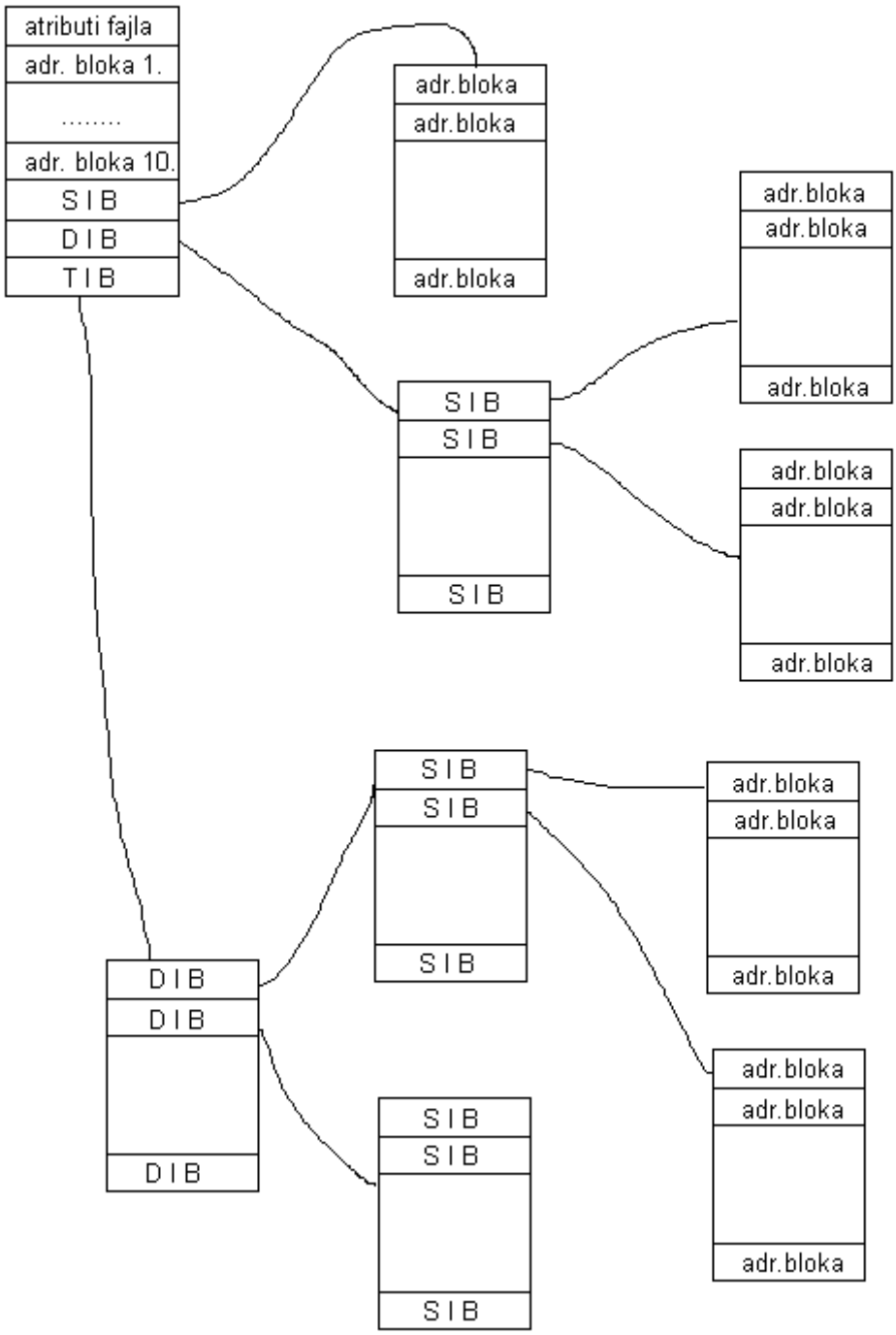
FAT se koristi npr. kod operativnih sistema *DOS,Win9x.*

Primer: neka `PERA.TXT` zauzima blokove 7,2,4,1 a `DJURA.DOC` blokove 8,5. Tada dobijamo sledeću tabelu alokacije:



4. i-čvorovi (i-nodes)

Svakom fajlu dodeljujemo jednu malu tabelu koja se zove **i-čvor (i-node)** ili **index-čvor (index-node)** i koja sadrži atribute fajla i adrese zauzetih blokova:



Znači: i-čvor sadrži atribute fajla i adrese prvih nekoliko blokova zauzetih od strane datoteke. Ako su ti blokovi dovoljni, onda je i-čvor spreman. Ako nisu, koristi se i polje **SIB=Single Indirect Block** koji pokazuje na blok koji sadrži adrese dodatnih blokova. Neka su blokovi veličine 1KB, neka se koristi 32-bitno adresiranje, onda svaki blok može da sadrži 256 adresa. Tako pomoću *SIB*-a dolazimo do dodatnih 256 blokova. Ako ni to nije dovoljno, koristimo i polje **DIB=Double Indirect Block** koji pokazuje na blok koji sadrži dodatne adrese *SIB*-ova. Ako ni to nije dovoljno, onda koristimo i polje **TIB=Tripple Indirect Block**, koji pokazuje na blok sa dodatnim adresama *DIB*-ova. Tako dolazimo do sledeće tablice:

ŠTA KORISTIMO	MAKSIMALAN BROJ BLOKOVA
samo i-čvor	10
i-čvor + SIB	10+256
i-čvor + SIB + DIB	10+256+256*256
i-čvor + SIB + DIB + TIB	10+256+256^2+256^3

Implementacija direktorijuma

1. CP/M

CP/M je višekorisnički OS, ima samo jedan direktorijum. Svi fajlovi svih korisnika su tamo. Svakom fajlu se dodeljuje jedan ili veći broj slogova sa sledećim poljima:

broj bajtova	**1**	**8**	**3**	**1**	**2**	**1**				
opis	kod korisnika	ime fajla	ekstenzija (tip) fajla	redni broj opisa	rezervisano	broj zauzetih blokova	adresa prvog bloka	adresa drugog bloka	.....	adresa bloka 16.

Svaki korisnik ima svoj jedinstveni idenkifikacioni broj koji se čuva u prvom bajtu – znači možemo imati najviše do 256 korisnika. Zatim sledi ime i tip fajla. Posle toga redni broj opisa – ako fajlu nije dovoljno 16 blokova, dobija još jedan opis. Prvi opis ima redni broj 0, drugi 1 itd. Razlika između ovih slogova je jedino u rednom broju i u listi zauzetih blokova.

Karakteristika *CP/M* sistema je da ne zna tačnu veličinu fajlova (bajt-po-bajt), već samo blokovsku veličinu.

2. MS-DOS

Podržava hijerarhijsku realizaciju direktorijuma. Fajlovima se dodeljuju slogovi sledeće strukture:

broj bajtova	**8**	**3**	**1**	**10**	**2**	**2**	**2**	**4**
opis	ime fajla	ekstenzija (tip) fajla	atributi	**rezervisano**	vreme kreiranja	datum kreiranja	adresa prvog bloka	veličina fajla u bajtovima

Jednokorisnički, zna tačnu veličinu fajlova (u bajtovima). Pogledamo u slogu adresu prvog bloka, a adrese ostalih tražimo u FAT-u.

3. UNIX

Najjednostavnije rešenje: čuvamo samo adresu i-čvora i ime fajla. Atributi fajla se nalaze unutar i-čvora.

broj bajtova	**2**	**14**
opis	adresa i-čvora	ime fajla

Primer: kako OS nalazi putanju **/usr/ast/mbox** ? Objašnjenje ćemo dati pomoću *UNIX*ove realizacije: krenemo od korenskog direktorijuma (i-čvor korenskog direktorijuma je na fiksiranom mestu – bloku), otvorimo fajl direktorijuma – sadrži slogove oblika (adresa i-čvora, ime fajla). Potražimo slog sa imenom fajla **usr** i uzmemo adresu i-čvora. Odemo do tog i-čvora – sadrži atribute za usr i adrese blokova gde je sadržaj tog fajla-direktorijuma. U tim blokovima tražimo slog sa imenom fajla **ast** i uzmemo adresu i-čvora. Odemo do tog i-čvora i u navedenim blokovima (sadržaj fajla-direktorijuma ast) tražimo slog sa imenom fajla **mbox** da bismo dobili adresu i-čvora.

Linkovi

Link predstavlja sredstvo pomoću kojeg možemo iz više direktorijuma pristupiti istom fajlu (isti fajl možemo "videti" ispod više direktorijuma). Pod *Unix*om linkove delimo na **tvrde** i **meke** (**simboličke**).

Tvrdi link: pri kreiranju tvrdog linka, pravi se samo novi slog u direktorijumu sa imenom fajla za koji pravimo link i stavimo adresu i-čvora originalne datoteke. Znači, više imena pokazuju na isti i-čvor. Da bismo vodili računa o broju linkova, i-čvorovi u skupu atributa fajla treba da čuvaju i taj podatak. Ako u nekom folderu izdamo komandu za brisanje fajla, OS najpre pogleda broj tvrdih linkova – ako je 1, briše se slog iz direktorijuma i oslobađaju se blokovi zauzeti od strane fajla. Ako imamo više linkova, briše se samo slog iz direktorijuma a brojač linkova se smanji za jedan.

Meki link (simbolički link): pri kreiranju mekog linka, pravi se novi fajl tipa **link** – (pravimo i-čvor za novi fajl, i eventualno zauzmemo još jedan blok za sadržaj fajla – a sadržaj je jedino putanja do fajla na koji link pokazuje, zatim ubacimo novi slog u direktorijum sa adresom novog link-fajla). Brojač linkova originala se ne menja. Ako izdamo komandu za brisanje originalnog fajla, OS ne vodi računa o simboličkim linkovima, briše fajl (ako nema više tvrdih linkova). Ako sada pokušamo pristupiti originalu preko mekog linka, OS neće naći putanju, pa će javiti da fajl ne postoji. Ako izbrišemo simbolički link, to nema nikakvog uticaja na original.

Problem mekih linkova: sporost – da bi OS došao do sadržaja original, mora najpre naći i-čvor link-fajla, zatim blok sa sadržajem link-fajla, pa da izčita putanju, da potraži original u hijerarhiji direktorijuma i konačno, na kraju da otvori originalnu datoteku za rad. Mogli bismo malo poboljšati performanse ako putanju stavimo direktno u i-čvor linka.

Prednost mekih linkova: meki link – pošto sadrži putanju originala a ne adresu i-čvora – može pokazati na bilo koji fajl bilo gde na mreži, čak i na Internetu.

Možemo imati još neke probleme pri korišćenju linkova: npr. imamo fajl `PERA.TXT` u direktorijumu `/usr/pera` i tvrdi link u folderu `/usr/pera/doc`. Kopiramo `/usr/pera` zajedno sa svim poddirektorijumima. Treba dodatna pažnja da ne pravimo dve kopije od `PERA.TXT`.

Pouzdanost i performanse fajl sistema

Pouzdanost fajl sistema

Loši blokovi

Mediji za čuvanje podataka mogu imati loše blokove – fizički oštećene delove medije. Za izbegavanje loših blokova imamo dva rešenja:

Hardversko – kontroler diska vodi računa o lošim blokovima, ako želimo pristupiti oštećenom bloku (radi pisanja), kontroler će naš zahtev preusmeriti ka drugom, neoštećenom (i nekorišćenom) bloku.

Softversko – korisnik ili OS pravi poseban fajl koji sadrži sve loše blokove na disku. Na taj način ti blokovi su izbačeni iz skupa slobodnih (neiskorišćenih) blokova.

Sigurnosne kopije (backup)

Pored eliminisanja loših blokova, radi povećanja pouzdanosti sistema, poželjno je u određenim vremenskim intervalima napraviti sigurnosne kopije važnih fajlova. To može uraditi i sam korisnik ili OS. Neke tehnike:

1. Koristimo dva diska – A i B. Oba diska delimo na dva dela: deo za podatke i deo za sigurnosne kopije. Kada dođe vreme za backup, prekopiramo podatke diska A u deo diska B za sigurnosne kopije, a podatke diska B u deo za sigurnosne kopije diska A. Na taj način, ako se pokvari bilo koji disk, na drugom disku imamo sve podatke. Nedostatak: trošimo jako mnogo mesta za sigurnosne kopije i jako mnogo vremena za backup.
2. Nećemo prekopirati sve podatke, samo fajlove koji su izmenjeni od poslednjeg backup-a. *MS-DOS* npr. ima podršku za ovaj način arhiviranja: svaki fajl ima atribut bit sa imenom **archive** – kada izvšimo *backup*, ovaj bit se resetuje. Ako izmenimo sadržaj fajla, bit se setuje.

Konzistentnost (file system consistency)

Mnogi sistemi pročitaju sadržaj nekih blokova, rade sa njima u memoriji i pišu ih natrag na disk kasnije. Ako padne sistem pre nego što uspemo vratiti promenjene blokove, može doći do problema. Ako je reč o blokovima koji sadrže i-čvorove ili opise direktorijuma, greška može biti vrlo ozbiljna. Da bismo izbegli takve situacije, potrebno je obezbediti neke pomoćne programe za ispravljanje nastalih grešaka. Npr. *scandisk* pod *DOS*-om ili *Windowsima*. Imamo dve vrste provere konzistentnosti:

1. **Konzistentnost blokova**: pravimo tabelu za blokove. Za svaki blok stavimo po dva brojača: prvi pokazuje koliko puta se dati blok javlja u listi zauzetih blokova, a drugi u listi slobodnih blokova. Tada idemo redom po i-čvorovima i gledamo koje blokove koriste fajlovi – pri tome povećavamo prvi brojač odgovarajućih blokova. Zatim gledamo listu slobodnih blokova (ili bitnu mapu) i isto tako povećamo drugi brojač. Na kraju dobijamo rezultat:
    - ako je vrednost jednog brojača 1 a drugog 0, onda je sve u redu – blok je ili zauzet ili slobodan

- ako je vrednost oba brojača 0, znači da blok nije ni zauzet, ni slobodan – greška!, našli smo blok koji je izgubljen: nije iskorišćen a ne može biti ni u budućnosti jer nije ni evidentiran kao slobodan. Rešenje: proglasimo ga slobodnim
- ako je vrednost prvog brojača 0 a drugog veći od 1 – to znači, da je blok evidentiran više puta kao slobodan (može biti samo u slučaju da koristimo povezanu listu, u slučaju bitne mape to se ne može desiti). Rešenje: popravimo listu slobodnih blokova – izbacimo duplikate
- ako je vrednost prvog brojača veći od 1 a vrednost drugog nula – blok pripada više od jednog fajla. Rešenje: ako blok pripada n fajlu, zauzmemo jos n-1 blokova, prekopiramo sadržaj u te blokove i ažuriramo i-čvorove. Posle, za svaki slučaj javimo i korisniku šta smo uradili.
- ako je vrednost oba brojača veći od 0 – to znači da je blok evidentiran i kao zauzet i kao slobodan. Rešenje: izbacimo iz liste slobodnih blokova.

2. **Konzistentnost fajlova**: Idemo po folderima od korenskog direktorijuma i za svaki i-čvor dodeljujemo po jedan brojač. Svaki put kad naiđemo na isti fajl, povećamo brojač. Na kraju idemo redom po i-čvorovima i upoređujemo tako dobijen brojač sa brojačem tvrdih linkova unutar i-čvora. Rezultat može biti:
   - vrednost brojača odgovara broju tvrdih linkova nutar i-čvora – nema grešaka
   - vrednost brojača je veći od broja tvrdih linkova – greška!! Ako izbrišemo fajl, može se desiti da imamo link na neiskorišćen blok. Posle neko iskoristi taj blok i sad imamo link koji pokazuje na ko zna šta... Rešenje: brojač tvrdih linkova preuzima vrednost drugog brojača
   - broj tvrdih linkova je veći od vrednosti brojača – greška!! Čak i ako svi korisnici izbrišu sve tvrde linkove na fajl, blok neće biti prebačen u skup slobodnih blokova. Rešenje: brojač tvrdih linkova uzima vrednost drugog brojača.

Performanse fajl sistema

Pristup disku je mnogo sporiji od pristupa memoriji, pa su mnogi sistemi dizajnirani tako da se smanji broj pristupa disku. Za ostvarenje tog cilja, najčešće se koristi **bafer ili keš (buffer, cache)**. Keš se sastoji od skupa blokova koji logički pripadaju disku, ali se, radi bržeg pristupa, čuvaju u memoriji. Ako se javlja zahtev za pristup nekom bloku, najpre proverimo da li je u kešu, ako jeste, uzmemo podatke iz keša. Ako nije, učitamo sa diska u keš, i podatke čitamo iz keša.

Ako treba učitati neki blok sa diska u keš, potrebno je izabrati neki blok koji će biti izbačen iz keša. Za izbor možemo koristiti sve algoritme koje smo koristili kod straničenja (paging). Ako je blok, koji se izbacuje izmenjen, moramo njegov sadržaj vratiti na disk.

Uzmimo na primer LRU algoritam.

Pri izboru bloka za izbacivanje treba uzeti u obzir i sledeće stvari:

1. **Da li će blok uskoro ponovo biti potreban ?**
   Blokove delimo u grupe: i-čvororovi, SIB,DIB,TIB, deo direktorijuma, popunjen blok, parcijalno popunjen blok. Parcijalno popunjene blokove stavimo na kraj liste jer ćemo verovatno još upisivati podatke u njih, popunjeni blokovi idu na početak liste da bi se što pre izbacili na disk.

2. **Da li je blok od esencijalne važnosti za konzistentnost fajl sistema?** (i-čvorovi, direktorijumi)
   Ako je blok bitan za konzistenciju fajl sistema, treba ga što pre prebaciti na disk. Time ćemo smanjiti mogućnost grešaka ako dođe do pada sistema. U ovu grupu spadaju blokovi i-čvorova, SIB,DIB,TIB,direktorijumi.

3. **Briga o korisnicima.**
   Vodimo računa o tome da očuvamo konzistentnost fajl sistema, pa guramo bitne blokove ka početku liste da ih što pre prebacimo na disk. Korisnik piše izveštaj u text editoru. Napiše 10 stranica, čak i ako periodično kaže editoru da prebaci dokument na disk, postoji dobra šansa da ostane u kešu – dajemo prednost bitnijim blokovima. Ode struja na trenutak, nemamo *UPS*, odu tih 10 stranica sa strujom. Fajl sistem je konzistentan, a korisnik je besan, lupa po tastaturi i baca monitor preko prozora.

Neka rešenja:

UNIX: imamo proces koji se zove **sync**. Njegov zadatak je da svakih 30 sekundi protrči kroz keš i prebaci sve izmenjene blokove na disk. Na taj način, ako dođe do pada sistema, gubimo samo rad poslednjih 30 sekundi. To je efikasno, ali nepouzdano rešenje

DOS: svaki izmenjen blok odmah se prebacuje na disk. Time se popravlja pouzdanost, ali se kvari efikasnost – praktično ubrzanje imamo samo kada čitamo sadržaj bloka koji je u kešu.

Pored keša, i sami korisnički programi mogu imati svoje interne bafere za skladištenje podataka što dodatno komplicira stvari.

Keširanje nije jedini način za poboljšanje performanse fajl sistema. Druga tehnika je da blokove kojima ćemo verovatno pristupati istovremeno, stavimo što bliže što možemo (da smanjimo vreme pozicioniranja).

Optimizacija diska

Prednosti diska u odnosu na internu memoriju:

- mnogo veći kapacitet
- mnogo manja cena
- podaci se ne gube nakon isključivanja računara

Disk možemo zamisliti koa niz magnetnih ploča, pri čemu pišemo na obe strane ploče. Ispod i iznad svake ploče nalazi se po jedna glava za čitanje/pisanje. Glava diska može da se kreće horizontalno, a ploče se rotiraju. Površinu ploča delimo na sledeće delove:

Staza (track) – deo površine ploče koje se može obići rotiranjem ploče bez pomeranja glave (koncentrični krugovi).

Cilindar (cylinder) – sve staze iste veličine na svim pločama (uzimajuci u obzir obe strane)

Sektor (sector) – staze delimo u sektore – sektori predstavljaju najmanju jedinicu za prenos podataka sa i na disk.

Prilikom pristupa disku radi čitanja/pisanja čekamo na:

1. vreme pozicioniranja glave na odgovarajući cilindar (na odgovarajuću stazu – **seek time**) – horizontalno pomeranje glave
2. vreme koje je potrebno da rotiranjem traženi sektor dovodi ispod glave (**latency time**)
3. vreme koje je potrebno za prenos podataka (**transfer time**)

Ako imamo multiprogramirano okruženje, možemo imati i više procesa koji čekaju da bi dobili pristup podacima na disku. Ovi procesi se povezuju na neku listu čekanja i kontroler diska redom zadovoljava zahteve. Ako su zahtevani podaci jako rasuti po disku, gubimo mnogo vremena za pozicioniranje glave, zato se koriste algoritmi za izbor procesa. Ideja je da biramo proces koji želi pristup sektoru koji je najbliže trenutnom položaju glave. Pri tome moramo voditi računa o sledećim stvarima:

- što veća propusnost (što veći broj prenosa u jedinici vremena)
- što manje prosečno vreme čekanja
- što manja varijansa (razlika između prosečne i pojedinačnih vremena čekanja)

Algoritmi za optimizaciju pomeranja glave

1. **Prvi u redu (FCFS – First Come First Served)**: zahtevi se zadovoljavanju po redu nastavnka. Nema nikakve optimizacije.
2. **Najbliži zahtev (SSTF – Shortest Seek Time First)**: Zadovoljavamo onaj proces koji potražuje podatak koji se nalazi najbliže trenutnoj poziciji glave. Nedostatak je što može doći do pojave izgladnjivanja (starvation) – radimo sa procesima koji traže nešto u blizini, a neki proces, koji želi čitati sa udaljenog mesta čeka nepredvidljivo dugo.
3. **Skeniranje (SCAN)**: glava se neprestano pomera u određenom pravcu (npr. prema sredini diska). Zadovoljavamo onaj zahtev koji je najbliži glavi u datom pravcu. Kada glava stigne do centra ili nema više zahteva u tom pravcu, vraća se unazad. Nedostatak: sektori na sredini se obilaze u relativno pravilnim ali većim intervalima, dok sektore na ivici ploče i u centru ploče posetimo dva puta unutar malog vremesnkog intervala (najpre u jednom pravcu, pa u drugom pravcu) a onda čekamo duže vreme da se glava vrati.
4. **N-SCAN**: radi isto kao SCAN, sa razlikom da uzima u obzir samo n onih zahteva koji su bili već prisutni kada se glava počela pomerati u datom pravcu. Zahtevi koji stignu u međuvremenu, moraće da čekaju.
5. **Skeniranje u jednom pravcu (C-SCAN – Circular Scan)**: radi slično kao scan, sa razlikom da skenira samo u jednom pravcu: ide od ivice ka centru diska i zadovoljava zahteve, kada stigne do centra, prebaci se nazad na ivicu i ponovo ide ka centru. Time je obezbeđeno da glava provodi istu kolučinu vremena i nad središnjim i nad spoljašnjim trakama.

ZAGLAVLJIVANJE
(deadlock)

Resursi (resources)

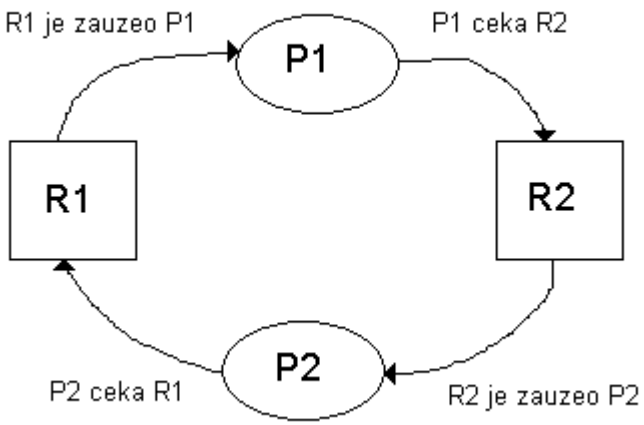
Resurs je sve što je nekom procesu potrebno za izvršavanje: procesor,memorija, fajlovi, štampač, skener itd. Resursi se dele u dve velike grupe:

1. Resursi koji mogu biti oduzeti od procesa a da to nema uticaja na sam proces (**preemtive resource**). Npr. procesor, memorija.
2. Resursi koji ne mogu biti oduzeti od procesa (**nonpreemtive resource**). Npr. štampač. Ako proces A preda zahtev za neki ovakav resurs koji je već zauzet od strane procesa B, postaje blokiran (ode na spavanje) dok dati resurs ne postane ponovo slobodan.

Zaglavljivanje (deadlock)

Zaglavljivanje je situacija kada dva ili više procesa čekaju na neki događaj koji se nikada neće dogoditi. Npr. proces A čeka da ga probudi proces B a proces B čeka da ga probudi proces A – tako lepo obojica spavaju čekajući onog drugog za buđenje. Drugi primer: procesi A i B žele nešto da štampaju sa magnentne trake. Kreće proces A i zauzme štampač. Tada dolazi proces B i zauzme traku. Sada A pogleda na traku – vidi da je zauzeta i blokira se. Proces B pogleda štampač, vidi da je zauzeto i blokira se. Proces A čeka na događaj da se traka oslobodi i ne pušta štampača. Proces B čeka na događaj da se štampač oslobodi i ne pušta traku. Niko ništa ne pušta, svi čekaju.

Vizuelno:



Zaglavljeni: proces P1 je zauzeo resurs R1, ali mu je za dalji rad potreban i R2, koji je zauzet od strane procesa P2, a za dalji rad tog procesa treba resurs R1.

Coffman (1971) je dokazao da postoje 4 neophodna uslova za mogućnost javljanja zaglavljivanja:

1. **Međusobno isključivanje (mutual exclusion)**. Procesi imaju ekskluzivnu kontrolu nad resursima: proces P1 zauzme resurs R1 i niko drugi ne može koristiti taj resurs, čak ni ako P1 ne koristi (npr. zauzme štampač ali ne štampa).
2. **Zauzimanje i čekanje (hold and wait)**. Procesi čuvaju svoje resurse dok čekaju na druge resurse: proces P1 čuva resurs R1 a čeka na R2.
3. **Resursi ne mogu biti silom oduzeti (nonpreemtive resourses)**. Resursi zauzeti od strane procesa ne mogu biti oduzeti od strane OS-a. Proces mora da završi svoj rad da bi on sam oslobodio zauzete resurse: proces P1 je zauzeo R1 i ne možemo ga oduzeti dok P1 ne završi svoj rad i sam ne oslobodi resurs.
4. **Cirkularna veza između procesa (circular wait)**. Proces P1 traži resurs R2 koji je zauzet od strane procesa P2, a P2 traži R1 koji je zauzet od strane P1.

Važno je napomenuti da je istovremeno prisustvo sva četri uslova neophodno za pojavu zaglavljivanja. Ako neki uslov nije ispunjen, zaglavljivanje nije moguće.

Moguće strategije borbe protiv zaglavljivanja :

0. **Ne radimo ništa** (najjednostavnije rešenje: procesi su blokirani? Resetujemo računar.)
1. **Prevencija** – OS dizajniramo tako da ne može doći do zaglavljivanja
2. **Izbegavanje** – OS je takav da postoji mogućnost zaglavljivanja, ali implementiramo neke algoritme za izbegavanje
3. **Detekcija** – OS ne zna ni za prevenciju ni za izbegavanje, ali ume da otkrije zaglavljivanje i da o tome obavesti korisnika
4. **Oporavak** – znamo da su neki procesi zaglavljeni i pokušamo rešiti problem

Prevencija (prevention)

Prevencija predstavlja najbolje rešenje i podrazumeva da pravimo OS tako da nikada ne može doći do zaglavljivanja. To ćemo postići ako onemogućimo bar 1 od navedena 4 uslova.

Međusobno isključivanje: OS koji ne pruža mogućnost međusobnog isključivanja neće biti popularan: Pera zauzme štampač i počne da štampa cenovnik komponenata. Dolazi Milica i preuzme štampač za štampanje horoskopa. Rezultat: cenovnik sa horoskopom. Znači prvi uslov nećemo dirati, ako procesu treba ekskluzivna kontrola, dobija ekskluzivnu kontrolu.

Zauzimanje i čekanje: Treba onemogućiti da proces zauzme neki resurs i da čuva taj resurs dok čeka na neke druge resurse. Jedno rešenje je da zahtevamo da procesi predaju listu svih potrebnih resursa pre nego što počne njihovo izvršavanje. Ako neki resursi nisu dostupni, proces će čekati. Ako su svi resursi dostupni, izvršavanje može početi. Imamo više problema: neki procesi ne znaju unapred koje sve resurse će koristiti, resursi neće biti optimalno korišćeni: proces traba da pročita neke podatke sa diska, zatim tri sata nešto izračunava i onda štampa rezultat. Kako zahtevamo da svi resursi budu zauzeti istovremeno, štampač će biti zauzet tri sata bez korišćenja.

Resursi ne mogu biti silom oduzeti: Imamo dva rešenja: Kod prvog rešenja, ako proces javi zahtev za neki resurs, a resurs je nedostupan, OS oduzima sve resurse koje je proces do sada zauzeo i proces može nastaviti sa radom tek kada su svi potrebni resursi na raspolaganju. Može se doći do izgladnjivanja (starvation). Drugo rešenje: ako proces javi zahtev za neki resurs i nema slobodnih primeraka tog resursa, OS pokuša oduzeti od procesa koji su takav resurs već zauzeli ali čekaju na druge resurse. Ako ni to ne pomaže, proces ode na spavanje. I tu se može doći do izgladnjivanja.

Cirkularna veza između procesa: svakom resursu dodelimo jedinstveni broj, a od procesa očekujemo da poštuju sledeće pravilo: resurse mogu zahtevati po rastućem redosledu. Znači ako je proces dobio resurs sa brojem 5, može zahtevati samo resurse iznad 5. Tada sigurno nećemo imati cirkularnu vezu među procesima: proces A drži resurs R*i*, proces B drži resurs R*j*. Tada ako je i>j, A neće moći tražiti resurs Rj. Ako je i<j, onda B neće moći tražiti resurs Ri. Nedostaci:

- teškoće oko numerisanja resursa
- ako sistemu dodamo nove resurse, moramo ponovo izvršiti numerisanje

Izbegavanje (avoidance)

Pretpostavka je da OS nismo dizajnirali tako da zaglavljivanja budu nemoguća, ali ćemo potražiti neke algoritme i tehnike za izbegavanje. Da bismo izbegli zaglavljivanja, sistem mora biti u stanju da svaki put kada dobije zahtev za zauzimanje resursa, odluči da li je zauzimanje bezbedno ili nije. Ako zauzimanje nije bezbedno (vodi do zaglavljivanja), zahtev se odbija.

Bankarov algoritam

Bankarov algoritam potiče od **Dijsktre** (1965) i radi na sledeći način: imamo banku i veći broj korisnika. Korisnici traže zajam od banke radi realizovanja svojih projekata. Svaki korisnik mora javiti bankaru, koja je maksimalna količina novca koji je njemu potreban. Banka ima konačne resurse. Pozajmljeni novac mogu vratiti samo oni korisnici koji su uspeli da završe svoje projekte, a da bi završili svoje projekte moraju dobiti prijavljenu maksimalnu količinu novca. Ako bankar nije dovoljno pametan, može se desiti da ostane bez novca a da ni jedan korisnik ne završi sa radom – korisnici su se zaglavili.

Pretpostavke za korišćenje bankarovog algoritma:

- imamo fiksni broj resursa: M
- imamo fiksni broj procesa: N
- svaki proces unapred javi maksimalnu količinu potrebnih resursa
- broj najavljenih (potrebnih) resursa od strane jednog procesa mora biti manji od M (bankar mora imati dovoljno novca)
- operativni sistem garantuje da će svaki proces dobiti potrebne resurse unutar konačnog vremenskog intervala (niko neće beskonačno čekati – ako bankar nema dovoljno novca, kaže korisniku da malo sačeka i garantuje da će u konačnom vremenu dati potrebnu količinu novca)
- proces ne sme da laže: broj trenutno zauzetih resursa mora biti manji ili jednak od najavljenog maksimalnog broja resursa (korisnik ne sme da krade novac iz banke)
- proces garantuje, da će kada dobije sve potrebne resurse, u konačnom vremenu završiti sa radom i vratiće sve zauzete resurse (korisnik mora garantovati bankaru da će vratiti sav pozajmljeni novac u konačnom vremenu nakon realizovanja projekta)

Trenutno stanje OS zovemo:

- **SIGURNO STANJE** – ako OS može omogućiti svakom procesu da završi svoj rad u konačnom vremenu (ako bankar može garantovati da će svi korisnici dobiti potrebnu količinu novca u konačnom vremenu).
- **NESIGURNO STANJE** – ako OS ne može garantovati da će svaki proces dobiti najavljen broj resursa.

Bankarov algoritam radi na sledeći način: ako neki proces javi da mu je potreban neki resurs, OS simulira budućnost: pogleda šta će se desiti ako zadovolji zahtev. Ako zadovoljavanje zahteva vodi do sigurnog stanja, proces dobija resurs. Ako zadovoljenje zahteva vodi do nesigurnog stanja, zahtev se odbija.

Primer sigurnog stanja: Neka je broj resursa sistema 12.

PROCES	BROJ ZAUZETIH RESURSA	MAKSIMALNO TRAŽENO
A	1	4
B	4	6
C	5	8

Tada na raspolaganju imamo još 2 slobodna resursa.

Stanje je sigurno: ako proces B dobija 2 preostala resursa, imaće sve potrebne resurse i završiće svoj rad u konačnom vremenu. Tada ćemo imati 6 slobodnih resursa. Tada i proces A i proces C mogu dobiti potrebnih 3 resursa i mogu završiti svoj rad.

Prelazak iz sigurne u nesigurno stanje: pp. da C traži 1 resurs i OS daje taj resurs. Tada ćemo na raspolaganju imati još jedan slobodan resurs. Procesu A su potrebna 4, procesu B 2 a procesu C još 2 resursa. OS ima 1. To je nesigurno stanje. Niko ne može garantovati da će neki od procesa moći vratiti neke resurse (ili da će neki proces uspeti da završi svoj rad i sa manjim brojem resursa). **Nesigurno stanje ne vodi obavezno u zaglavljivanje, ali mogućnost postoji. NESIGURNO STANJE ZNAČI DA NE MOŽEMO GARANTOVATI DA NEĆE DOĆI DO ZAGLAVLJIVANJA.** Zato će OS, kad vidi da zadovoljavanje zahteva vodi u nesigurno stanje, odbiti zahtev i proces C će morati da čeka.

Primer nesigurnog stanja: Neka je broj resursa sistema 12.

PROCES	BROJ ZAUZETIH RESURSA	MAKSIMALNO TRAŽENO
A	8	10
B	2	5
C	1	3

Tada na raspolaganju imamo još jedan slobodan resurs.

Stanje je nesigurno: Nemamo dovoljno slobodnih resursa da bilo koji proces dobija maksimalnu količinu. To ne znači da ćemo obavezno doći do zaglavljivanja, samo nas informiše o tome da je zaglavljivanje moguće.

Nedostaci Bankarovog algoritma:

- fiksan broj resursa
- fiksan broj procesa
- u praksi nije lako ostvariti garanciju OSa da će svaki proces dobiti sve potrebne resurse
- u praksi nije lako ostvariti garanciju procesa da će ako dobije sve potrebne resurse završiti rad u konačnom vremenu
- postoje situacije kada procesi ne mogu unapred znati maksimalan broj potrebnih resursa (npr. interaktivni sistemi – razvoj događaja zavisi i od korisnika)

Detekcija (detection)

Pretpostavimo da OS nema mogućnost prevencije ni izbegavanja, ali želimo imati mogućnost detekcije – ako dođe do zaglavljivanja, barem da budemo svesni toga i javimo korisniku da nešto ne valja.

Jedno rešenje je da OS vodi računa o odnosima procesa i resursa pomoću usmerenog grafa. Ako je resurs R1 zauzet od strane procesa P1, strelica vodi od R1 do P1. Ako proces P1 čeka na resurs R2, strelica vodi od P1 do R2. Do zaglavljivanja dolazimo ako se javljaju ciklusi u grafu. Znači algoritam za detekciju pretražuje usmeren graf resursa i procesa. Ako nađe ciklus, javi korisniku da su se odgovarajući procesi zaglavili.

Pitanje je koliko često treba pokrenuti algoritam za datekciju? OS bi mogao nakon svake dodele, pri svakom zahtevu za resursom izvršiti proveru. To bi obezbedio najraniju detekciju zaglavljivanja, ali bi jako usporavao rad sistema. Kao bolje rešenje, mogli bismo algoritam za detekciju pokrenuti u određenim vremenskim intervalima. Time ćemo manje ugroziti performanse sistema, malo ćemo kasniti sa detekcijom, ali pošto nemamo druge mehanizme za borbu protiv zaglavljivanja, nije preterano bitno da ćemo par sekundi zakasniti.

Oporavak (recovery)

Kod ove strategije, malo smo unapredili OS: ume da detektuje zaglavljivanja a pored toga pokušava i da se oporavi od toga. Imamo više načina:

Mogli bismo zaglavljene procese suspendirati. Time ćemo oduzeti sve resurse tih procesa, više neće takmičiti za resurse, neće trošiti procesorsko vreme, ali će i dalje ostati procesi. Obavestimo korisnika da smo ih iz tog i tog razloga suspendirali, pa neka korisnik izabere šta će sa njima.

Drugi način je da ih ubijemo. Možemo ih sve odjednom ubiti ili jedan po jedan, pa videti šta će se desiti, ako su ostali još uvek zaglavljeni ubijemo još jedan itd. Ovaj algoritam može se lako implementirati i efikasno rešava problem zaglavljivanja, ali baš i ne pruža neki oporavak.

Prilikom izbora procesa koji će biti ubijen, trebalo bi uzeti u obzir sledeće parametre:

- broj zauzetih resursa – možda imamo veću šansu za pobedu ako ubijemo proces koji drži veći broj resursa
- broj traženih resursa – trebalo bi izbaciti proces koji čeka na veći broj resursa, jer možemo pretpostaviti da će ponovo upasti u zaglavljivanje
- količina obavljenog posla (potrošeno vreme procesora, rad sa IO uređajima) – bolje da ubijemo proces koji je tek počeo svoj rad a ne neki koji je već pri kraju

- prioritet procesa – treba birati proces što manjeg prioriteta (važniji procesi imaju prednost za život)

Treći način je da pokušamo privremeno oduzeti neke resurse od nekog procesa i damo ih drugom procesu. Znači, oduzmemo resurs od procesa A, dajemo ga procesu B, kada se resurs oslobodi vratimo ga procesu A i očekujemo da bez nekih ozbiljnih posledica A ume da nastavi svoj rad. Ovaj način oporavka je često vrlo teško implementirati ili čak nemoguće – zavisi od prirode resursa.

Četvrti način je da pokušamo proces prebaciti u neko prethodno stanje. Ovde pretpostavljamo da sistem operator vodi dnevnik nevaljalih procesa. Znači operator zna koji procesi su skloni zaglavljivanju, pa za te procese postavlja **tačke provere (checkpoint)**. Svaki put kada proces dođe do te tačke, sačuva se na disku. Ako dođe do zaglavljivanja, pogledamo koji resursi su potrebni i vratimo proces u neko prethodno stanje a resurse dajemo drugom zaglavljenom procesu. Ako sada prvi proces ponovo traži te resurse, moraće da čeka. Kod izbora procesa koji će biti vraćen u neko prethodno stanje možemo uzeti u obzir parametre iz drugog načina oporavka.

Izgladnjivanje (starvation)

Izgladnjivanje je pojava slična zaglavljivanju, ali nije tako opasna, jer do zaglavljivanja i ne dolazi. Ako imamo multiprogramirano okruženje, procesi se takmiče za resurse (čekaju u redu). OS mora imati neku strategiju za izbor procesa koji će sledeći dobiti određeni resurs. Ako ta politika izbora nije dovoljno demokratska, može se desiti da neki proces nikada ne dođe do traženog resursa. Svaki put dođe neki drugi proces koji na osnovu politike OS-a ima veći prioritet i nosi resurs sa sobom. Naš proces nije zaglavljen već mirno čeka u redu deset godina, jer ne može ništa protiv politike OS-a.

SIGURNOST I ZAŠTITA

Sigurnost

Postoji potreba da operativni sistem onemogući neautorizovani pristup podacima svakog korisnika. Sigurnost i zaštita su usko vezani za fajl sistem (o operativnoj memoriji smo ranije pričali). Dakle, potrebno je onemogućiti pristup nekim fajlovima.

Sigurnost se odnosi na opšti - filozofski pojam, dok zaštitu predstavljaju usvojeni principi sigurnosti koji se realizuju na nekom operativnom sistemu.

Kada sigurnost fajl sistema može biti ugrožena?

- viša sila (udar groma, požar, zemljotres...)
- hardverska i softverska greška
- ljudske greške

Jedan od načina na koji se branimo od gubitka važnih podataka je pravljenje rezervnih kopija (*backup*a) koje se potom čuvaju na sigurnom mestu.

Pored ovih slučajnih postoje i namerni napadi na sigurnost fajl sistema. Lica koja žele pristupiti zabranjenim fajlovima mogu biti:

- **laici**, koji nisu zlobni ali ako im se pruži prilika da "zavire" u tuđu poštu to će i uraditi.
- oni za koje zaobilaženje mehanizama zaštite predstavlja **intelektualni izazov**.
- oni koji žele da izvuku **materijalnu korist** iz toga (prevarom, ucenom,...)
- **špijuni** (vojni, privredni,...)

Nekoliko poznatih grešaka u ranijim operativnim sistemima

UNIX u fajlu **/etc/passwords** čuva spisak svih korisnika sistema u obliku

<korisničko_ime kriptovana_lozinka identifikacoini_broj grupa ime prezime ...>

Ovaj fajl je svima dostupan za čitanje. Postoji komanda UNIX-a **lpr** koja se koristi za štampanje fajlova. Ova komanda između ostalih ima i opciju brisanja fajla pa se njom može obrisati i fajl passwords. Brisanjem ovog fajla ni jedan korisnik se više ne može prijaviti na sistem.

Takođe u UNIX-u. Proces koji je pokrenuo neki program ima ista prava kao i njegov vlasnik. Postoji bit koji kontroliše prava i on se zove **setuid**. Ako je **setuid** setovan tada proces koji ga je pokrenuo ima sva prava. Korisnik (nasilnik) iz svog procesa pokrene proces koji ima **setuid** postavljen na

1 (na primer mkdir) i zatim ga nasilno prekine. Pod UNIX-om postoji standardni fajl **core** koji operativni sistem kreira i popunjava kada dođe do greške pri izvršavanju programa. Ako korisnik predhodno napravi u svom direktorijumu fajl **core** kao link na fajl **/etc/passwords** neki sadržaj će biti direktno upisan u njega. Ovo je moguće zato što je u trenutku prekida proces imao postavljen **setuid** na 1 tj. imao je sve privilegije. Uz malo muke, ono što se upisuje može ličiti na sadržaj **passwords** pa se korisnik na taj način dopiše.

MULTICS je imao loše zaštićene mehanizme za paketne obrade. Bilo je moguće učitati podatke sa trake, odnosno kartice i snimiti ih bilo gde. Jedini uslov je bio da se to uradi paketno a ne interaktivno. Ljudi su tada pravili svoje editore koje su snimali u direktorijum odakle žele ukrasti fajl. Kada bi korisnik pokrenuo takav editor ne bi bio svestan da dok on kuca, program mu paketno krade podatke. Ovakvi programi, koji osim što rade ono čemu su namenjeni rade i nešto "krišom" se nazivaju **trojanski konji**.

TENEX - Digital DEC10 je takođe imao neke mane:

- za svaki Page Fault se mogla "zakačiti" proizvoljna procedura
- pristup svakom fajlu se kontrolisao lozinkom i to interaktivno
- lozinka se proveravala slovo po slovo. čim bi naišao na slovo koje ne odgovara, javio bi grešku

Neka neki program želi da pristupi zaštićenom faju. On pokušava sa nekom lozinkom koja se napravi tako da prvo slovo leži u jednoj stranici memorije a ostatak u drugoj. Memorija se popuni (nebitnim stvarima) tako da se u njoj nađe samo prva stranica. Ovo je potrebno da bi došlo do Page Faulta nakon što se obradi prvo slovo lozinke. Na Page Fault se "zakači" procedura koja obaveštava o svakom Page Fault-u. Tako, ako se pogodi prvo slovo procedura će odreagovati na Page Fault i obavestiti korisnika, a ako ne operativni sistem će javiti da je lozinka pogrešna. Kada se pogodi prvo slovo onda se lozinka tako "šteluje" da prva dva slova budu u jednoj stranici a ostatak u drugoj i postupak se ponavlja. Za lozinku od 6 od 30 mogućih znakova grubom silom bi trebalo izvršiti 30^6 provera, a na ovaj način samo $30 \cdot 6 = 180$.

Operativni sistem **OS/360** je omogućavao da se u toku računanja u pozadini nešto čita. Lozinka se pri kopiranju proveravala u dva navrata i to prvi put se proveravalo smemo li uopšte kopirati sa trake u zadati fajl a drugi put kada kopiranje zaista i počne (ali tada se nije proveravalo i ime fajla). Ako se između ove dve provere promeni ime fajla, bilo je moguće kopirati preko nekog fajla bez obzira da li se imala njegova lozinka ili ne.

Šta treba da radi osoba koja provaljuje ?

- Treba da gleda deo diska sa izbačenim stranicama iz memorije. Operativni sitem ih nekad ne obriše posle swap-ovanja pa se tu mogu kriti vredni podaci.
- Treba da poziva nepostojeće sistemske pozive ili postojeće ali sa pogrešnim parametrima. Postoji mogućnost da će se operativni sistem zbuniti pa se iz takvih situacija može nešto korisno zaključiti.
- Pri procesu prijavljivanja nekako ga nasilno prekinuti. Tako se možda može zaobići proveravanje lozinke.
- Modifikacija strukture podataka koja se koristi za pozivanje servisa operativnog sistema može zbuniti operativni sistem pa se opet može nešto korisno saznati.

- Može da napravi program koji simulira prijavljivanje. Pokrene takav program i napusti računar. Sledeći korisnik seda za računar i unosi svoju lozinku koju program negde zapamti pa zatim pozove pravi program za prijavljivanje.
- Raditi sve što uputstva kažu da nije dozvoljeno, da je opasno ...
- Šarmirati sekretaricu ?

Principi za ostvarivanje sigurnosti

- Dizajn zaštite treba da bude javan. Treba dobro realizovati principe (pa se hvaliti).
- Podrazumevajući način pristupa svakom objektu - fajlu je "bez pristupa" tj niko ne sme da mu pristupi. U praksi nikad nije baš tako ali je nešto logično, na primer vlasnik ima sve privilegije a ostali nikakve.
- Uvek se traže tekuća prava pristupa..
- Svaki proces treba da započne rad sa što manje privilegija. Ako su potrebne veće, onda se naknadno eksplicitno povećaju.
- Mehanizam zaštite treba da bude jednostavan, uniforman, od početka zamišljen.
- Mehanizam zaštite treba da bude psihološki prihvatljiv.

Provera identiteta

Ni jedan korisnik se ne bi smeo lažno predstaviti. Korisnik ne sme koristiti resurse ako nije ovlašćen. Uobičajeni način provere identiteta je putem lozinke. Pri prijavljivanju na UNIX sistem potrebno je uneti **login** (korisničko ime pod kojim smo poznati operativnom sistemu) i **password** (lozinka). UNIX tada proverava u fajlu **etc/passwords** da li postoji odgovarajuće ime i da li lozinka odgovara. Lozinka je jednostrano šifrirana što znači da ne postoji način da se nakon kriptovanja vrati u prvobitno stanje.

Kako izabrati lozinku i da li je ona dovoljna?

Teoretski je moguće napraviti 95^7 lozinki dužine sedam karaktera što je i više nego dovoljno. Ali korisnici prave kraće lozinke i ne koriste sve moguće karaktere. Takođe, korisnici su skloni standardnim, predvidivim lozinkama. Tokom nekog istraživanja pokušale su se provaliti lozinke koje ljudi koriste. Upotrebljavana su imena i prezimena korisnika, imena njihovih žena, dece, brojevi automobilskih tablica, datumi rođenja i slično. Kada su tako napravljene lozinke upoređene sa pravim više od 85% je bilo pogođeno.

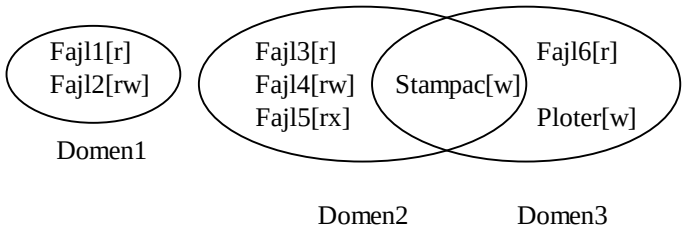
Zato se uvode dodatne provere.

- Generatori slučajnih brojeva. Lozinka se dodatno šifrira 12-bitnim brojem koji je jedinstven za svakog korisnika?
- Jednokratne lozinke. Korisnik ima knjižicu sa lozinkama, pročita koja mu treba, iskoristi je pa dobije drugu.
- Lična pitanja. Pri prvom prijavljivanju se korisniku postavljaju lična pitanja, pa se ponekad proverava identitet korisnika pomoću njih.

- Korisniku se pri prvom prijavljivanju dodeli algoritam, na primer x^2. Pri kasnijim prijavljivanjima od korisnika se zahteva ime, lozinka i da mu se neki broj. Korisnik tada unosi ime, lozinku i rešenje.
- Pri odabiranju lozinke se mogu postavljati uslovi tako da lozinka na kraju mora ispasti nelogična. Na primer, korisnik u lozinci mora da iskoristi šest znakova od kojih jedan mora biti specijalan, jedan mora biti veliko slovo, jedan mora biti cifra,...

Ako je provera identiteta jako bitna onda se koristi i fizička provera (koja nije baš jako popularna od strane korisnika). Može se proveravati zenica, krv, otisci prstiju,...

Zaštita



Svakom objektu se dodele tri slova koja označavaju prava pristupa. Na primer (objekat,rwx). Domen je skup parova (objekat,prava).Kada se proces počne izvršavati dodeli mu se domen i tada on može da pristupa samo objektima iz tog domena i to kako su prioriteti zadani.Oni koji dele domene sa vremena na vreme mogu menjati domen.

Domeni se predstavljaju matricama.

Na primer:

	Fajl 1	Fajl 2	Fajl 4	Fajl 4	Fajl 5	Stampac	Fajl 6	Ploter	D1	D2	D3
Dom 1	R	RW								Enter	
Dom 2			R	RW	RX	W					
Dom 3						W	R	W			

Poslednje tri kolone služe za prelazak iz jednog domena u drugi.

Kako su ovakve matrice prilično retko popunjene, obično se predstavljaju drugačije i to po vrstama ili kolonama pri čemu se pamte samo korisni podaci. Ako se matrica pamti po kolonama onda se naziva Acces Control List (ACL), a ako se pamti po vrstama C-lista.

Na primer, po kolonama. Domen je korisnik+grupa:

Fajl0 - (Pera,*,rwx)
Fajl1 - (Djoka,system,rwx)
Fajl2 - (Pera,*,rw)(Mika,osoblje,r)(Aca,nastavnici,rx)
Fajl3 - (Pera,*,)(*,studenti,rwx)

Fajlu Fajl0 može da pristupa Pera iz bilo koje grupe i da radi sve.
Fajlu Fajl1 može da pristupa Djoka iz grupe system i da radi sve.
Fajlu Fajl2 mogu da pristupaju Pera iz bilo koje grupe (da čita i piše), Mika iz grupe osoblje (da čita) i Aca iz grupe nastavnici (da čita i izvršava).
Fajlu Fajl3 Pera iz bilo koje grupe ne može da pristupi ni na koji način dok svi korisnici iz grupe studenti mogu da rade sve.

U **UNIX**-u su svakom fajlu pridruženi 9 bitova kojima se određuju prioriteti. Prva tri se odnose na vlasnika, druga tri na grupu kojoj vlasnik pripada, a poslednja tri na sve ostale korisnike. **r** bit označava pravo čitanja, **w** bit označava pravo pisanja, a **x** bit označava pravo izvršavanja fajla.

Na primer, 111101100 znači da:

- vlasnik može da čita i piše u fajl i da ga izvršava fajl
- grupa kojoj vlasnik pripada može da čita i izvršava fajl
- svi ostali mogu samo da čitaju fajl.

rwx	rwx	rwx
vlasnik	grupa	ostali