1. Vrste multiprogramiranja.

Osnovna ideja multiprogramiranja bila je da se u radnu memoriju smesti više programa (procesa) kako bi se poboljšala iskorišćenost procesora. Memorija bi se podelila na delove (particije) u koje bi se učitavali programi. Programi koji imaju potrebu da rade na procesoru bi se smenjivali na njemu tako da on uvek bude zaposlen, a dok se jedan program izvršava na procesoru, drugi imaju mogućnost da vrše ulazno/izlazne operacije. Glavni cilj je maksimalno povećanje iskorišćenosti sistema, ali je sa druge strane poželjno da vreme izvršavanja programa bude manje.

Deljenje vremena (Time sharing) - koncept koji se zasniva na deljenju računara između više korisnika. Procesor nije bilo moguće podeliti fizički, pa se došlo na ideju da se podeli "vremenski", tako što bi svaki korisnik dobio određeno vreme u kojem bi imao procesor na raspolaganju. Po isteku dodeljenog vremena, procesor bi dobio sledeći korisnik. Mali vremenski intervali koje korisnik često dobija stvarali su iluziju da na raspolaganju ima procesor koji sve vreme radi samo za njega.

Multitasking - "moderniji" pristup u odnosu na multiprogramiranje. Multitasking podrazumeva da je jedinica izvršavanja na procesoru posao (task) koji ne mora nužno da obuhvata izvršavanje procesa na procesoru između dve ulazno/izlazne operacije. Proces po izvršavanju posla oslobađa procesor za sledeći posao procesa koji je na redu.

Multiprocesiranje - doživljava ekspanziju u poslednjoj generaciji računara. Obično se odnosi na postojanje više procesora u računarskom sistemu koji hardverski omogućavaju istovremeno izvršavanje više poslova (različitih ili istih) procesa.

2. Uporediti multiprogramiranje, multitasking i paketnu obradu.

Paketna obrada podrazumeva da se programi nadovezuju jedan na drugi tj. da se blokovi kartica korisnickih programa redjaju jedan za drugim. Ovakav pristup je omogucavao da se u trenutku izvrsavanja jednog programa ucitava sledeci koji je na redu.

Osnovna ideja *multiprogramiranja* je da se u radnu memoriju smesti vise procesa kako bi se poboljsala iskoriscenost procesora. Memorija racunara se deli na particije i u odgovarajuce particije se smestaju programi (procesi) koji ce se smenjivati u svom izvrsavaju. Glavni cilj multiprogramiranja je maksimalno povecanje iskoriscenosti sistema, ali je pozeljno da vreme izvrsavanja programa bude sto manje.

Multitasking predstavlja moderniji i efikasniji nacin implementacije ideje multiprogramiranja. Procesi se ne smenjuju pri pojavi U/I operacija (za razliku od multiprogramiranja). Jedinica izvršavanja na procesoru je posao (task) koji ne mora nužno da obuhvata izvršavanje procesa na procesoru između dve ulazno-izlazne operacije.

Multiprocesiranje se obicno odnosi na postojanje vise procesora u sistemu koji hardverski omogucavaju istovremeno izvrsavanje vise poslova. Vise se odnosi na hardverske mogucnosti sistema.

3. Vrste operativnih sistema u zavisnosti od jezgra.

Monolitni sistemi

→ U jezgru se nalaze svi servisi operativnog sistema zajedno sa drajverima, integrisani u jedan program. Svi delovi se pokrecu u istom trenutku, izvrsavaju u sistemskom rezimu i u istom delu memorije. Pri pokretanju ovakvog OS-a, jezgro se u memoriju ucitava u celosti kao jedan izvrsni program. F-je jezgra mogu jedna drugu pozivati bez ogranicenja. Veoma velika povezanost f-ja na niskom nivou obicno doprinosi brzini i efikasnosti ovakvih jezgara. Aplikativnim programima na raspolaganje se stavljaju sistemski pozivi kroz koje su implementirane usluge OS-a. Jedna od najvecih mana je losa otpornost na greske. U slucaju da dodje do greske u nekom od podsistema, dolazi do problema koji mogu uticati na ceo sistem i najcesce je jedino resnje njegovo ponovno podizanje. MS-DOS, BSD, AIX, Windows 98 i GNU/Linux.

Slojeviti sistemi

→ OS je izgradjen od zasebnih slojeva koji se nadogradjuju jedan na drugi. Svaki sloj ima odredjene funkcije koje su opisane kroz njegov interfejs ka visem sloju. Slojevi se implementiraju tako da mogu da koriste iskljucivo usluge prvog sloja ispod sebe. Iz tog razloga projektovanje slojeva je veoma zahtevan posao jer se mora voditi racuna o raspodeli funkcija. Problem ovakvih sistema je neefikasnost. Sistemski poziv prolazi kroz vise slojeva, a pri svakom prolazu se prosedjuju podaci, menjaju parametri itd. sto dovodi do usporenja. Slojevi ne moraju biti u jezgru operativnog sistema, vec zavisno od implementacije, odredjeni slojevi mogu funkcionisati u korisnickom rezimu. Najnizi sloj cini hardver ili njegova apstrakcija, na najvisem sloju je podrska za aplikativne programe, a izmedju moze biti proizvoljan broj slojeva. THE, Multics.

Mikrojezgro

→ Podrazumeva minimalno jezgro u kojem se nalaze samo najosnovnije funkcije.Deo funkcija koje bi u prethodnim slucajevima bile deo jezgra se izmestaju u korisnicki prostor i to u zasebne prostore u memoriji tako da sa bezbednijeg nivoa pristupaju jezgru. Usluge OS-a koje su izmestene iz jezgra se njemu obracaju kako bi ostvarile svoje ciljeve, a preko njega i komuniciraju sa drugim uslugama. Usluge koje obavljaju slicne zadatke u okviru OS-a se grupisu u procese koji se nazivaju serverski procesi, a cesto i samo serveri. Drajveri se cesto ne nalaze u mikrojezgru, vec pripadaju odgovarajucim serverima. Arhitekturu mikrojezgra odlikuje veci stepen sigurnosti u odnosu na monolitne sisteme, ali su oni obicno sporiji. Promene memorijskog prostora dovode do kasnjenja i manje propusnosti u poredjenju sa sistemima sa monolitnim jezgrom. One su neophodne jer se serveri i jezgro ne nalaze u istom adresnom prostoru i ne izvrsavaju u istom rezimu. Mach, Minix, QNX i L4.

Hibridni sistemi

→ Predstavljaju kompromis izmedju monolitne i arhitekture koja se zasniva na mikrojezzgru. Neke veoma bitne, kao i f-je koje se cesto izvrsavaju spustaju se u jezgro kako bi se povecala brzina i efikasnost, ali se dobar deo f-ja zadrzava u nivoima iznad jezgra. Hibridna jezgra se koriste u vecini komercijalnih OS-a: Apple Max OS X, Microsoft Windows NT 3.1, NT 3.5, NT 3.51, NT 4.0, 2000, XP, Vista, 7, 8, 8.1

Egzojezgro

→ Ideja je da jezgro obezbedi osnovne resurse i da aplikacijama prepusti rad sa njima. Ovo se postize premestanjem apstrakcije iznad hardvera u posebne biblioteke koje obezbedjuju minimalne apstrakcije uredjaja. Programeri imaju slobodu u izboru nivoa apstrakcije kada je pristup hardveru u pitanju. Ovakav nacin rada moze da doprinese znatnom ubrzanju i poboljsanju performansi. XOK, EXOS.

4. Prednosti koriscenja niti. Primeri.

Niti omogucavaju znacajne ustede memorijskog prostora i vremena. Niti dele memoriju i neke resurse koji pripadaju istom procesu tako da zauzimaju manje prostora nego kada su u pitanju nezavisni procesi. Takodje, iz istog razloga, niti se kreiraju mnogo brze od procesa, a i prebacivanje konteksta izmedju niti istog procesa je brze od prebacivanja konteksta izmedju procesa jer se prebacuju samo resutrsi koji su jedinstveni za nit – *stek, registri i programski brojac*. Niti pruzaju mogucnost aplikacijama da nastave rad u situacijama kada se izvrsavaju dugotrajne operacije koje bi bez podele poslova procesa na niti privremeno zaustavile izvrsavanje ostalih delova procesa. Slicno, koriscenjem niti omogucava se rad procesa ciji su delovi potpuno blokirani. Razvojem i ekspanzijom viseprocesorskih sistema do izrazaja narocito dolaze prednosti koje donosi koriscenje ovog koncepta jer vise niti mogu da se izvrsavaju istovremeno.

Primer serverske aplikacije (koja ima zadatak da opsluzi veliki broj klijenata za kratko vreme):

Tradicionalni pristup - podrazumevao je da serverski proces u petlji ceka da se pojavi klijent, a onda se za svakog novog klijenta kreira novi proces kojem ce prepustiti opsluzivanje klijenta.

Koriscenjem niti - umesto razlicitih procesa koristi se samo jedan u okviru kojeg se kreira veliki broj niti. Jedna nit procesa je zaduzena da "osluskje" mrezu tj. da prihvata klijente pri cemu se za svakog novog klijenta kreira nova nit. Kreiranje niti je brze i zhteva manje prostora nego kada su u pitanju procesi, tako da ovakav pristup omogucava da se mnogo brze odgovori na zahteve, ali i da se opsluzi mnogo veci broj klijenata.

5. Sistemski pozivi.

Programi uz pomoc sistemskih poziva komuniciraju sa jezgrom i pomocu njega dobijaju mogucnost da izvrse osetljive operacije u sistemu. *Sistemski pozivi* su skup funkcija koji predstavlja interfejs ka operativnom sistemu. Oni su implementirani tako da dozvole samo operacije koje ne mogu biti stetne po racunarski sistem. Sistemskim pozivom se jasno definise koje su dozvoljene operacije kada je odgovarajuca usluga operativnog sistema u pitanju. Na ovakav nacin se pristup hardveru stiti od potencijalno stetnih operacija korisnika.

Procesori savremenih racunarskih sistema imaju mogucnost rada u dva razlicita rezima: *korisnickom* i *sistemskom*. U sistemskom je moguce izvrsiti sve instrukcije, dok je broj dozvoljenih instrukcija u korisnickom rezimu redukovan. Aplikativni programi se veci deo vremena izvrsavaju u korisnickom rezimu dok je sistemski rezim predvidjen za posebno osteljive operacije koje izvodi operativni sistem. Pri koriscenju sistemskog poziva se prelazi iz korisnickog u sistemski rezim i dalju kontrolu preuzima operativni sistem. Kljucni deo OS-a koji reaguje u ovim situacijama je jezgro. Sistemski pozivi koriste jezgro da bi omogucili razlicite servise OS-a. Svi programi, cesto ukljucujuci i sistemske, funkcionisu na nivou iznad jezgra u korisnickom rezimu rada. Zbog osetljivosti poslova kojima se jezgro bavi, ono se ucitava u poseban, zasticeni deo memorije i time cuva od nezeljenih promena.

Pri dizajniranju OS-a cesto se tezi da se vise akivnosti odvija u korisnickom rezimu umesto u sistemskom jer se na taj nacin povecava stabilnost sistema. U tom slucaju eventualne greske pri izvrsavanju programa ili problemi uglavnom ne mogu da ugroze funkcionisanje sistema. Kada aplikativni programi izvrse sistemski poziv, parametri sistemskog poziva se postave na predvidjene lokacije u memoriji. Zatim, menja se rezim rada u sistemski i jezgro preuzima kontrolu i na osnovu parametara sistemskog poziva izvrsava zeljenu operaciju. Po zavrsetku operacije rezim rada se menja nazad u korisnicki, a rezultati se vracaju programu koji je izvrsio.

6. Fragmentacija: sta je, koje vrste postoje i gde se javlja?

Jedna od nezeljenih pojava kod sistema sa particijama je fragmentacija. Ona nastaje kada u racunarskom sistemu postoje delovi memorije koji su slobodni, ali ih sistem ne moze iskoristiti. Fragmenti predstavljaju neiskoriscen slobodan memorijski prostor, iako postoje procesi koji cekaju da se izvrse. Fragmentacija moze biti interna i eksterna.

Interna fragmentacija se odnosi na prostor koji ostaje neiskoriscen kada se proces smesti u deo memorije veci od njegovih memorijskih potreba. Interna fragmentacija se skoro sigurno javlja kod statickog particionisanja memorije.

Eksterna fragmentacija podrazumeva da u sistemu postoje delovi memorije koji su slobodni ali da nisu dovoljno veliki da se u njih smesti neki proces. Iz tog razloga ovi delovi memorije su neupotrebljivi. Procesu se od slobodne memorije dodeljuje onoliko koliko mu je potrebno. Ostatak particije formira novu slobodnu particiju, za koju se moze desiti da je nedovoljno velika da sadrzi proces. Eksterna fragmentacija moze nastati kod dinamickih particija, ali i kod statickog particionisanja (kada nijedna slobodna particija nije dovoljna za izvrsavanje procesa). Kod dinamickih particija, jedan od nacina da se resi eksterna fragmentacija je *kompakcija*. Kompakcija podrazumeva da operativni sistem zaustavi procese i da se izvrsi relokacija procesa tako da se oni sabiju na poceka ili kraj memorije. Pri tome se slobodna memorija grupise na drugi kraj memorije i formira jedinstvenu particiju. Ova operacija je vremenski dosta zahtevna sto dovodi do toga da sistem gubi na efikasnosti ako se cesto koristi, tako da upravljanje memorijom ne bi trebalo zasnivati na kompakciji.

7. Algoritmi planiranja.

FCFS algoritam

→ Algoritam za raspoređivanje procesa, koji se zbog svoje jednostavnosti i pravednosti obično prvi nameće. Zasnovan je na ideji da se procesor dodeljuje procesima po redu kako su ga zatražili. Jednostavno se implementira korišćenjem FIFO strukture. Kada se proces pokrene i prijavi za korišćenje procesora, on se tada postavlja na kraj reda. U trenutku kada se procesor oslobodi, tada se prvi sledeći procesor izbacuje iz reda i dobija procesor na raspolaganje. Ovaj algoritam nema verziju koja bi dozvoljavala prekidanje jer bi to bilo u suprotnosti sa glavnim principom na kojem je zasnovan. Prednost ovog algoritma je laka implementacija, ali je vreme čekanja često veliko. Može dovesti do takozvanog "efekta konvoja", situacije kod koje dosta procesa čeka da se izvrši jedan vremenski zahtevan proces.

SPF algoritam

→ Zasniva se na ideji da se favorizuju procesi čiji je (sledeći) zahtev za procesorom (procesorskim vremenom) manji. Svakom procesu se dodeljuje očekivano vreme za njegovo sledeće izvršavanje (posao) na procesoru, a procesor se dodeljuje redom procesima koji zahtevaju manje vremena. Ako se dogodi da dva procesa imaju isto očekivano vreme, obično se primenjuje FCFS algoritam za odluku koji će proces dobiti prednost. Ovaj algoritam se može implementirati i sa prekidanjem i bez. Ako se dozvoli prekidanje procesa, proces P i se može prekinuti kada se pojavi novi proces P j čija je sledeća operacija na procesoru vremenski manje zahtevna od operacije procesa P i koja se trenutno izvršava. Optimalan je kada se posmatra srednje vreme čekanja za bilo koji skup poslova. Glavni problem je procena dužine trajanje sledeće aktivnosti procesa na procesoru. Kod dugoročnog planiranja, kao procena se može uzeti unapred zadata granica vremena potrebnog da se posao završi, dok je kod kratkoročnog planiranja teže primeniti ovaj algoritam, iz razloga što nema pouzdanog načina da se precizno odredi trajanje sledeće aktivnosti. Obično se za predviđanje dužine sledeće aktivnosti koristi *eksponencijalna sredina* prehodnih aktivnosti i prethodnih procena potrebnog vremena. Sledećim izrazom se može

aproksimirati dužina trajanja sledeće aktivnosti: $T_{n+1} = \alpha T_n + (1 - \alpha)t_n$

pri čemu je t_n vreme koje je bilo potrebno da se izvrši n-ta aktivnost procesa na procesoru, a T_n očekivano vreme koje je bilo predviđeno za izvršavanje n-te aktivnosti procesa na procesoru, dokje α parametar koji reguliše uticaj procenjenog vremena i vremena koje je stvarno bilo potrebno za izvršavanje prethodne aktivnosti u proceni dužine sledeće aktivnosti.

Algoritam sa prioritetima

→ Podrazumeva da se svakom procesu pridruži veličina koja predstavlja njegov prioritet i da se kao sledeći za izvršavanje bira onaj proces koji ima najviši prioritet. Ako se dogodi da dva procesa imaju isti prioritet, bira se onaj koji je prvi došao u sistem. Prioriteti se mogu definisati interno ili eksterno. Interno definisani prioriteti koriste neku merljivu veličinu za izračunavanje prioriteta, npr. vremenske granice, memorijski zahtevi, prosečno trajanje aktivnosti na procesoru, dok se eksterni prioriteti definišu u odnosu na neke spoljašnje kriterijume, kao što su vremenski rokovi, važnost klijenta, vrednost projekta itd. Primer jednog pristupa prioritetima, od najvišeg ka najnižem je: sistemski procesi, interaktivni procesi, paketni procesi, korisnički procesi. Glavna mana je potencijalno "izgladnjivanje" procesa. Može se dogoditi da procesi višeg prioriteta konstantno uzimaju procesor ispred onih sa nižim prioritetom. Ovaj problem se može rešiti povećavanjem prioriteta procesa sa porastom njegovog vremena provedenog u redu za čekanje.

Kruzni algoritam

→ Zasnovan na ideji sličnoj kao kad je deljenje vremena u pitanju. Svaki proces dobija procesor na unapred zadati vremenski interval. Kada to vreme istekne ili ako se proces izvrši do kraja, on se prekida, a procesor na kvantum vremena dobija sledeći proces iz reda spremnih procesa. Prekinuti proces, u slučaju da se nije izvršio do kraja, stavlja se na kraj reda spremnih procesa. Ako ima n spremnih procesa, a kvantum vremena je dužine q, onda svaki proces dobija najmanje 1/n procesorskog vremena, a najviše po q u jednom prolazu. Pri tome, nijedan proces ne čeka duže od (n-1)q vremena. Performanse ovog algoritma veoma zavise od q. Ako q teži beskonačnosti, onda se ovaj algoritam ponaša kao FCFS algoritam, a ako je q veoma malo, ovaj algoritam se naziva deljenje procesora i teoretski izgleda kao da svaki od n procesa u redu ima sopstveni procesor koji je n puta sporiji od procesora koji se nalazi u sistemu. Na kraju svakog kvantuma vremena vrši se prebacivanje konteksta procesa, što je izuzetno skupa operacija. Poželjno je da se kvantum vremena izabere tako da bude znatno duži od vremena koje je potrebno za prebacivanje konteksta, ali ako je kvantum preveliki, onda algoritam više liči na FCFS algoritam.

Redovi u vise nivoa

→ Pristup koji je zasnovan na ideji da se red spremnih procesa podeli u više redova sa različitim prioritetima. Poslovi koji pristižu svrstavaju se u odgovarajući red na osnovu nekog kriterijuma. Svaki red može da ima sopstveni algoritam planiranja, a između samih redova postoji jasno definisana razlika u prioritetu. Posao koji je prvi u svom redu može dobiti procesor na korišćenje isključivo ako su redovi višeg prioriteta prazni.

Redovi u vise nivoa sa povratnom vezom

→ Pristup koji predstavlja modifikaciju malopre navedenog algoritma, tako da se eliminiše problem izgladnjivanja procesa. Dozvoljeno je kretanje procesa između redova, tako što se u toku izvršavanja izdvoje procesi tako da se oni koji koriste previše procesorskog vremena premeste u redove sa nižim prioritetom. Svaki proces u višim redovima dobija određeni kvantum vremena i kada ga iskoristi, premešta se u red na nižem nivou.

8. Sta su planeri i cemu služe?

Odabir procesa i njihovo rasporedjivanje po redovima, ali i organizaciju cekanja vrse *planeri*. Postoje dve vrste: dugorocni, kratkorocni. Cesto se implementira i srednjorocni planer.

Dugorocni planer od skupa svih procesa koji bi trebalo da se izvrse iz reda poslova, biraju one koji ce da se aktivno ukljuce u sistem i pocnu sa izvrsavanjem. Oni bi trebalo da naprave dobar odabir procesa za red spremnih procesa.

Kratkorocni planeri imaju zadatak da donose odluke o tome koji ce se od spremnih procesa izvrsavati i koliko dugo ce dobiti procesor. Kratkorocni planer se cesto naziva i planer procesora. Kratkorocni planeri se cesto pozivaju pa je potrebno da budu implementirani tako da sto brze donose odluke.

Sa druge strane dugorocni planeri se ne pozivaju tako ucestalo kao kratkorocni pa se moze dozvoliti njihov nesto sporiji rad. U praksi, za dugorocne planere je bitnije da naprave dobru kombinaciju poslova nego da ustede na vremenu, dok je kod kratkorocnih brzina rada vaznija od optimalnosti donete odluke.

Srednjorocni planer vrsi prebacivanje procesa ili u cilju poboljsanja kombinacije izabranih poslova ili da bi se oslobodila memorija u slucaju zagusenja. Posto se nekad desava da je skup izabranih procesa u memoriji takav da jedni druge ometaju i zaglavljuju cime se umanjuje efikasnost celokupnog sistema. U takvim situacijama je cesto korisnije ukloniti neke od procesa iz memeorije nego forsirati visi stepen multiprogramiranja na ustrb efikasnosti. Uklonjeni procesi se mogu ponovo vratiti u memoriju kada se steknu uslovi za to.

9. Shortest process first (algoritmi planiranja).

SPF (Shortest Process First) bira se proces cije ce izvrsavanje krace trajati (ne citav proces, vec neki njegov posao). Ukoliko postoji vise procesa sa istim ocekivanim vremenom, onda se primenjuje FCFS algoritam nad njima. SPF algoritam se moze implementirati sa prekidanjem ili bez. Ako postoji prekidanje, proces Pi se moze prekinuti jer se pojavio proces Pj cije je vreme izvrsavavanja krace. Algoritam SPF ima optimalno vreme cekanja. Glavni problem jeste kako odrediti vreme po kojem se vrsi biranje procesa. Kod dugorocnog planiranja se kao procena moze uzeti unapred zadata granica vremena potrebnog da se posao zavrsi, dok je kod kratkorocnog planiranja teze proimeniti ovaj algoritam. Tada se koristi pristup koji se zasniva na aproksimaciji gde se obicno za predvidjanje duzine sledece aktivnosti koristi eksponencijalna sredina prethodnih aktivnosti i prethodnih procena potrebnog vremena.

10. Sta su sistemski prekidi i cemu sluze?

Prekidi su sistem koji je osmisljen kako procesor ne bi morao da trosi vreme na aktivno cekanje U\I komponenti. Ranije, procesor trazi podatke a onda ceka dok se oni ne posalju, ali sa prekidima, procesor zahteva, radi nesto drugo a ui uredjaj kada zavrsi salje signal prekida kako bi obavestio procesor da je posao zavrsen. Prekidi mogu biti maskirajuci i nemaskirajuci. Nemaskirajuci imaju veci prioritet i koriste se za prijavljivanje ozbiljne greske ili vrlo bitne informacije i ne mogu se ignorisati. Maskirajuci prekidi su prekidi koji se mogu odloziti(npr obavljamo kriticnu sekciju, maskirajuci ceka dok se ona ne obradi, ne maskirajuci se odmah obavlja).

11. Sta je, kako nastaje i kako se otklanja deadlock?

Situacija u kojoj dva ili više međusobno zavisnih procesa blokirani čekaju na resurse koje nikad neće dobiti naziva se *zaglavljivanje*. Ovakvo zaglavljivanje je trajno i za posledicu često ima zaglavljivanje čitavog sistema. Ako sistem sadrži procese koji su zaglavljeni, smatra se da je i on zaglavljen. Do zaglavljivanja dolazi pri radu sa resursima koji, kada se dobiju, moraju biti iskorišćeni bez prekidanja, tj. oduzimanja tog resursa i predavanja drugom procesu, pre nego što se potpuno iskoriste. Primer ovakve vrste resursa je *kritična sekcija*. Proces koji je dobio priliku da joj pristupi ne sme biti izbačen iz nje pre njenog završetka, jer bi dopuštanje pristupa drugom procesu moglo da proizvede pogrešan rezultat. Da bi došlo do zaglavljivanja, neophodno je da su u sistemu istovremeno ispunjena sledeća četiri uslova (Kofmanovi uslovi):

- *Uzajamno isključivanje* - u jednom trenutku tačno jedan proces može da koristi resurs, dok ostali moraju da čekaju njegovo oslobađanje.
- *Čekanje i držanje* - dok proces drži resurse sa kojima radi, može da zahteva nove koji su mu potrebni za dalje izvršavanje i da čeka one koji nisu trenutno raspoloživi.
- *Nemogućnost prekidanja* - operativni sistem nema pravo da oduzme resurse koje je dodelio procesu i kasnije mu ih vrati, kako bi te resurse u međuvremenu dodelio drugim procesima.
- *Kružno čekanje* - u sistemu može da postoji lanac procesa koji čekaju jedni na druge, tj. na resurse koji su im dodeljeni, čineći time krug.

Pošto je za zaglavljivanje potrebno da važe svi pomenuti uslovi, do zaglavljivanja neće doći ako se bilo koji od ovih uslova ukloni. Postoje razni pristupi kada je u pitanju odnos prema zaglavljivanju, ali se na osnovu mera koje se primenjuju, mogu svrstati u naredne četiri grupe:

- *Sprečavanje* - sistemske mere koje se implementiraju u operativni sistem i kojima se isključuje mogućnost zaglavljivanja
- *Izbegavanje* - dinamičke mere koje se preduzimaju kako bi se sistem vodio kroz stanja koja obezbeđuju da ne dođe do zaglavljivanja, ali ne garantuje se da do zaglavljivanja neće doći
- *Detekcija i oporavljanje* - dozvoljava se da do zaglavljivanja dođe, ali postoje mehanizmi za otkrivanje da li je do zaglavljivanja došlo, tretman procesa koji su zaglavljeni i eventualno spašavanje dela rezultata koji su dobijeni pre zaglavljivanja u slučajevima kada je to moguće
- *Nepreduzimanje bilo čega* - prihvatljivo na sistemima na kojima do zaglavljivanja retko dolazi ili u situacijama kada se ne radi o veoma važnim poslovima, ne pružaju se nikakve mogućnosti i ne vodi se računa o zaglavljivanju, a kada do njega dođe, najčešće se problem rešava tako što se ponovo pokrene proces ili čitav sistem.

12. Hardverski algoritmi za kriticne sekcije i primeri.

Hardverska rešenja za zaštitu kritične sekcije podrazumevaju upotrebu posebnih instrukcija procesora. Ideja je da se naprave mašinske instrukcije koje su u stanju da urade bar dve operacije bez mogućnosti prekida, tj. atomično. Najčešće se za potrebe zaštite kritične sekcije koriste sledeće tri instrukcije: TAS (Test and Set), FAA (Fetch and Add), SWAP (zamena).

TAS:
→ TAS operise sa dve promenljive *A=TAS(B)* i funkcionise tako sto se vrednost koja se nalazi u promenljivoj B prebacuje u promenljivu A, a u promenljivu B se upisuje 1.

Deklarise se globalna promenljiva *zauzeto* i postavi na nulu, dok svaki proces ima lokalnu promenljivu *ne_moze*. Promenljiva *ne_moze* se na pocetku izvrsavanja procesa postavi na 1, a onda se u petlji stalno proverava da li je *zauzeto* postavljeno na 0, tj. da li je ulaz u kriticnu sekciju

slobodan. Na ovaj nacin, kada se oslobodi ulaz (*zauzeto* postane 0) atomicnom operacijom ce tacno jedan proces biti obavesten da moze da udje u kriticnu sekciju (*ne_moze* ce postati 0) i kriticna sekcija ce se zatvoriti za ostale procese jer TAS operacija *zauzeto* postavlja na 1.

Mana ovakvog resenja je cinjenica da je to resenje zasnovano na hardverskoj podrsci, odnosno da je potrebno da procesor ima ugradjenu tu instrukciju. Sa druge strane, ovo resenje se bez bilo kakave modifikacije moze primeniti na proizvoljan broj procesa, ali ne postoji garancija koliko ce neki proces da ceka. Izgladnjivanje procesa je moguce, ali se retko dogadja.

```
Proces i:
    ne_moze = 1;
    WHILE (ne_moze == 1)
        ne_moze = TAS(zauzeto);
     ENDWHILE
     //Kriticna sekcija
    zauzeto = 0;
```

Modifikacija (da bi se obezbedila garancija da ce proces konacno dugo cekati na ulazak u kriticnu sekciju): Potrebno je deklarisati globalni niz *ceka[n]* ciji se clanovi inicijalizuju na nulu i promenljivu celobrojnog tipa *zauzeto* koja se takodje postavlja na nulu. Modifikacija se sastoji u tome da po zavrsetku rada u kriticnoj sekciji, proces koji je zavrsio pronadje sledeci od procesa koji cekaju i da ga prozove da udje u kriticnu sekciju tako sto ce mu promenljivu *ceka[j]* postaviti na 0 i time ga osloboditi od aktivnog cekanja. Ukoliko nema procesa koji cekaju da udju u kriticnu sekciju, promenljiva *zauzeto* se postavlja na 0 cime se ulaz u kriticnu sekciju oslobadja.

```
Proces i:
    ceka[i]=1;
    WHILE (ceka[i]==1 AND TAS(zauzeto)==1)
        // Aktivno cekanje dok je kriticna sekcija zauzeta
    ENDWHILE
    ceka[i]=0;
    // Kriticna sekcija
   // Trazi se sledeci proces
    j = (i+1) mod n;
    WHILE (ceka[j]==0 and j!=i)
         j = (i+1) mod n;
    ENDWHILE
    IF (i==j)
        zauzeto=0;
    ELSE
        ceka[j]=0;
```

```
ENDIF
```

FAA:

→ Instrukcija FAA takođe operiše sa dve promenljive. Njena sintaksa je FAA(A, B) i pri njenom korišćenju se vrednost promenljive B prebacuje u A, a vrednost B + A u promenljivu B. Kod druge operacije, u izrazu B + A, uzima se stara vrednost promenljive A, pre promene vrednosti.

SWAP:

→ SWAP operiše sa dve promenljive, SWAP(A, B) i rezultat je atomična zamena vrednosti promenljivih A i B. Primera radi, zamena se, uz ili bez korišćenja pomoćne promenljive, sastoji od bar tri operacije. Zastita kriticne sekcije:

```
Proces i:
  kljuc = 1;
  WHILE (kljuc != 0)
      SWAP(brava, kljuc);
  ENDWHILE
  // Kriticna sekcija
  brava = 0;
```

Procesi koji se nadmecu za pristup kriticnoj sekciji imaju vrednost *kljuc = 1*. Procesi u petlji pokusavaju da kroz SWAP ubace kljuc u bravu tj. da ugrabe trenutak kada je vrednost *brava = 0* pa da zamenom te vrednosti sa 1 dobiju uslov za izlazak iz ciklusa i pristup kriticnoj sekciji. Istovremeno se *brava* zakljucava. Ovako se ulazak u kriticnu sekciju blokira sve dok proces koji je u njoj ne zavrsi i postavi *brava=0* cime se otvara mogucnost da jedan od procesa koji cekaju udje u kriticnu sekciju.

13. Izbegavanje aktivnog cekanja kod procesa.

Postoje rešenja koja su efikasnija, tj. ne sadrže cikluse koji bespotrebno troše procesorsko vreme. Ove metode se obično zasnivaju na tehnikama koje podrazumevaju zaustavljanje procesa, bez aktivnog čekanja i njihovo pokretanje u odgovarajućem trenutku. Ideja je da se proces koji ne može da uđe u kritičnu sekciju na neki način blokira, pa da se, po sticanju uslova da uđe u kritičnu sekciju, "probudi", odnosno "prozove" i odblokira. Ovakva rešenja su bolja od pristupa sa aktivnim čekanjem, ali je njihova implementacija komplikovanija. Najpoznatiji metodi kada je ovakav pristup u pitanju, zasnivaju se na korišćenju: semafori, kriticni regioni i monitori.

14. Brojacki semafor pomocu binarnih.

```
P(C)
// Ekskluzivni pristup
P(S1);
// Umanjenje brojacke promenljive
C--;
IF(C < 0)
   // Blokira se proces
   V(S1);
    P(S2);
ENDIF
V(S1);
```

```
V(C)
// Ekskluzivni pristup
P(S1);
// Uvecanje brojacke promenljive
C++;
IF(C <= 0)
   // Oslobadjanje procesa koji ceka
   V(S2);
ELSE
   // Oslobadja se semafor koji stiti ekskluzivan pristup
  V(S1);
ENDIF
```

15. Kada se radi suspendovanje procesa?

Osim osnovnih stanja, operativni sistemi često podržavaju i suspendovana stanja u koje procesi mogu da pređu na zahtev korisnika ili operativnog sistema. Procesi koji su u stanju Spreman ili Čekanje se mogu suspendovati i onda oni prelaze u stanja Suspendovan i spreman, odnosno Suspendovan i čeka. Suspendovani proces oslobađa resurse koje je zauzimao pre suspenzije i prestaje da konkuriše za druge resurse koji su mu potrebni za izvršavanje, ali i dalje ostaje proces. Često se suspendovani procesi prebacuju na disk do prestanka suspenzije, čime se oslobađa i deo radne memorije koju su zauzimali i na taj način se ostavlja više prostora za druge procese. Do suspendovanja spremnog procesa najčešće dolazi zbog preopterećenosti sistema zbog velikog broja spremnih procesa, pa je privremeno suspendovanje nekog od procesa poželjno kako bi se operativnom sistemu olakšao rad. Takođe, do suspendovanja može doći da bi se izbeglo zaglavljivanje sistema ili ukoliko korisnik želi da privremeno zaustavi izvršavanje procesa kako bi dobio mogućnost da proveri međurezultate. Suspendovanje procesa koji je u stanju Čekanje se obično vrši da bi se oslobodili resursi kojima on raspolaže i time sprečilo zaglavljivanje ili ubrzao rad sistema. Proces koji je u stanju Suspendovan i čeka prelazi u stanje Suspendovan i spreman ako se stvore uslovi, tj. oslobode se resursi na koje čeka (zbog kojih se i našao u stanju Čekanje), tako da po povratku iz suspenzije može da bude u stanju Spreman. Prekidanje suspenzije procesa koje inicira korisnik ili operativni sistem realizuje se njegovim prelaskom iz stanja Suspendovan i čeka u stanje Čeka i iz stanja Suspendovan i spreman u stanje Spreman.

16. Potrošač i proizvođač preko semafora, pseudokod i diskutovati.

Primer potrošača i proizvođača se često koristi kao ilustracija trke za resurse. Dva procesa (proizvođački i potrošački) imaju zadatak da kontrolišu stanje u magacinu. Proces (funkcija) koji simulira rad proizvođača ima zadatak da povećava broj proizvoda koji se nalaze u magacinu u trenutku kada se novi proizvod donese u magacin. Analogno, proces koji simulira rad potrošača umanjuje zajedničku promenljivu koja sadrži stanje u magacinu kada se proizvod iznese iz magacina. Problem je u istovremenom pristupu promenljivoj *brojac* koja bi trebalo da nosi informaciju o trenutnom stanju u magacinu.

→ Za resavanje ovog problema koriscenjem semafora potreban je jedan binarni semafor koji bi trebalo da obezbedjuje rad sa magacinom. Potrebna su i dva brojacka semafora koji ce cuvati informaciju o broju praznih i punih mesta u magacinu. Semafor *mutex* obezbedjuje ekskluzivan pristup, a brojacki semafori su *pun* i *prazan.*

Proizvodjacki proces, prvo testiranjem brojackog semafora *prazan*, proverava da li u megacinu ima proizvoda. Zatim preko semafora *mutex* pokusava da u magacin smesti proizvod. Ako je neki proces u magacinu, ovaj proces ce sacekati blokiran na semaforu *mutex.* Slicno, potrosacki proces proverava stanje testiranjem semafora *pun.*

Glavni program:

```
// Inicijalizacija magacina
unutra = 0;
van = 0;
brojac = 0;
// Inicijalizacija semafora
mutex = 1;
pun = 0;
prazan = velicina_magacina;
PARALLEL WHILE(true)  // Procesi se izvrsavaju paralelno
    proizvodjac();
    potrosac();
```

```
    ENDPARALLEL WHILE

proizvodjac():
    P(prazan);   // Da li ima mesta u magacinu?
    P(mutex);   // Zastita ekskluzivnog pristupa magacinu
    magacin[unutra] = novi_proizvod;
    unutra = (unutra + 1) mod velicina_magacina;
    brojac++;
    V(mutex);
    V(pun);

potrosac():
    P(pun);      // Da ima ima proizvoda u magacinu?
    P(mutex);  // Zastita ekskluzivnog pristupa magacinu
    prodaja = magacin[van];
    van = (van + 1) mod velicina_magacina;
    brojac--;
    V(mutex);
    V(prazan);
```

17. Algoritam za uklanjanje aktivnog čekanja korišćenjem semafora.

P(S)	**V(S)**

S.vrednost = S.vrednost – 1;
IF(S.vrednost < 0)
dodati trenutni proces u S.lista;
blokirati proces;
ENDIF
	S.vrednost = S.vrednost + 1;
	IF(S.vrednost <= 0)
	izbaciti proces iz S.lista;
	odblokirati taj proces;
	ENDIF

18. Detaljno opisati koncept monitora.

Monitori predstavljaju najvisi nivo apstrakcije kada je zastita kriticne sekcije bez aktivnog cekanja u pitanju. Oni su konstrukcije programskih jezika u okviru kojih su implementirani mehanizmi za zastitu kriticne sekcije, ali i za sinhronizaciju. Predstavljaju konstrukcije programskog jezika, slicne klasama u objektno orjentisanom programiranju, koje mogu da sadrze procedure i promenljive. Glavna odlika monitora je da u okviru njega u jednom trenutku moze da bude aktivan samo jedan proces. Za potrebe sinhronizacije implementirane su specijalne uslovne promenljive, nad kojima su definisane 2 operacije: *wait* i *signal*, cija sintaksa moze da bude *x.wait()* i *x.signal()*. Ovim mehanizmima se obezbedjuje blokiranje procesa i kasnije ponovno pokretanje kada do potrebnog signala dodje, omoguceno je potpuno resenje problema pristupa kriticnoj sekciji i sinhronizacije procesa. Uslovne promenljive monitora nisu ekvivalentne brojackim semaforima jer kod njih cekanje mora da nastupi pre slanja signala. Funkcije koje se nalaze u okviru monitora moze pozvati tacno jedan proces, dok ostali ne mogu pozvati ni jednu funkciju iz monitora dok proces koji je pristupio monitoru ne zavrsi sa radom u njemu. Pristup kriticnoj sekciji se uz pomoc monitora veoma lako resava jer je dovoljno kriticnu sekciju ubaciti u monitor.

Dilema koja se javlja pri implementaciji je sledeća: ako u trenutku kada neki proces P signalizira *x.signal()*, a pri tome postoji proces Q koji je blokiran na uslovnoj promenljivoj *x*, tada treba doneti odluku da li da se suspendovanom procesu Q dozvoli da obnovi svoje izvšenje, a da proces P sačeka ili da se dozvoli obrnuto. U suprotnom, procesi P i Q bi bili istovremeno aktivni u monitoru, što narušava uzajamnu isključivost koja je najvažniji uslov kada je zaštita kritične sekcije u pitanju. Toni Hor je prihvatio princip da P čeka dok Q ne napusti monitor, dok je Brinč Hansen izabrao da

kada proces P izvrši operaciju *signal*, odmah napusti monitor, tako da Q odmah posle signala nastavlja svoje izvršenje. Pri tome, insistira se da poziv *signal* bude poslednje što se nalazi u okviru funkcije, tako da u svakom slučaju proces P završi pre ponovnog pokretanja procesa Q. Ovaj pristup uslovno je slabiji od Horovog, pošto na ovaj način proces može da signalizira najviše jednom.

19. Overlay i dinamicko punjenje memorije.

Prekrivači predstavljaju jedan od prvih pristupa za upravljanje memorijom, zasnovanih na ideji da se omogući izvršavanje programa koji nisu kompletno učitani u memoriju. Osnovni zadatak je da se identifikuju moduli programa koji su relativno nezavisni. Ovi moduli se nazivaju prekrivači. Oni obično sadrže delove programa koji se koriste u različitim vremenskim trenucima, delove koji se retko koriste, ali i one za koje postoji šansa da ne budu potrebni tokom izvršavanja procesa. Oni se odvajaju od glavnog dela programa i smeštaju u sekundarnu memoriju. Prilikom alociranja memorije za glavni program, rezerviše se jedan slobodan deo koji je dovoljno velik za prihvatanje najvećeg prekrivača. Kada je, za dalje izvršavanje programa, potreban deo programa iz nekog prekrivača, taj prekrivač se učitava u rezervisani deo memorijskog prostora. Kasnije, kada se zahteva deo programa iz drugog prekrivača, taj prekrivač će biti učitan u rezervisani deo memorije, tj. prepisaće se njen prethodni sadržaj (prekrivač). To znači da će se prekrivači po potrebi smenjivati u rezervisanom delu memorije. Naprednije verzije pristupa za upravljanje memorijom zasnovane na prekidačima omogućavaju prisustvo više prekrivača u memoriji i samim tim imaju komplikovaniji način njihovog raspoređivanja u rezervisanom delu memorije. Najveći nedostatak je to što operaciju deljenja programa na delove precizno i dovoljno dobro može da uradi jedino programer, odnosno, ona se ne može automatizovati. Dinamičko punjenje je zasnovano na sličnim principima na kojima se zasnivaju i prekrivači. Osnovna ideja kod ovog pristupa je da se funkcije i procedure ne smeštaju u memoriju sve do trenutka dok ne budu potrebne, odnosno pozvane. Funkcije i procedure se nalaze u sekundardnoj memoriji, a glavni program se smešta u memoriju i izvršava. U trenutku kada se pozove neka funkcija ili procedura, sistem prvo proveri da li je ona već u memoriji. Ako to nije slučaj, vrši se njeno prebacivanje sa sekundardnog medijuma. Punilac se poziva da bi obavio ovo punjenje memorije, ali i ažurirao odgovarajuće tabele kako bi odgovarale novom stanju u sistemu. Nakon učitavanja, nova funkcija počinje da se izvršava. Glavna prednost dinamičkog punjenja je to što se funkcije koje nisu potrebne za izvršavanje procesa nikada ne pune u memoriju. Ovakav pristup posebno dolazi do izražaja u programima koji imaju dosta koda koji se odnosi na slučajeve koji se retko ili skoro nikada ne dešavaju. Dinamičko punjenje ne zahteva podršku operativnog sistema tako da korisnici mogu relativno lako da ga implementiraju.

20. Bankarev algoritam.

Veoma efikasan, a pri tome i ne previše komplikovan algoritam za izbegavanje zaglavljivanja. Ime je dobio po principu davanja zajmova u bankama. Ovaj algoritam podrazumeva da je pre početka izvršavanja svakog od procesa poznat maksimalni broj resursa svakog tipa koji će mu biti potreban. Ideja je da se dinamički ispituje pokušaj dodele resursa kako bi se obezbedilo da nikada ne dođe do kružnog čekanja. Stanje dodele resursa je definisano brojem raspoloživih, dodeljenih resursa i maksimalnim zahtevima procesa. Smatra se da je stanje sistema bezbedno ako sistem može u određenom redosledu da dodeljuje resurse kako bi svaki proces mogao da završi sa radom, tj. da ne dođe do zaglavljivanja, a da se pri tome svi procesi izvrše. Redosled kojim se vrši ovakva dodela resursa procesima naziva se bezbedna sekvenca.

Definicija: Sekvenca procesa <P1, P2, ..., Pn> je bezbedna za tekuće stanje dodele resursa akko se potrebe svakog procesa P i mogu zadovoljiti slobodnim resursima i resursima koje trenutno drže procesi Pj (j < i).

Ako potrebni resursi za proces P i nisu odmah dostupni, tada proces P i mora da sačeka dok se svi

Pj (j < i) ne izvrše. Kada procesi Pj (j < i) završe sa radom, P i dobija potrebne resurse, izvrši se, a zatim vrati sve resurse koje je koristio. Kada završi proces Pi , tada Pi+1 može da dobije potrebne resurse i tako dalje.

Definicija: Sistem je u bezbednom stanju akko postoji bezbedna sekvenca.

Ako stanje nije bezbedno, to ne znači da će sigurno doći do zaglavljivanja, već da postoji mogućnost da do njega dođe.

Osnovna ideja Bankarevog algoritma je da se resursi procesima dodeljuju tako da se sistem sprovodi kroz niz bezbednih stanja kako bi se izbegla mogućnost da dođe do zaglavljivanja. Pri dodeli resursa, vodi se računa o tome da sistem uvek bude u bezbednom stanju, odnosno, po zahtevu procesa za dobijanje resursa, vrši se provera da li postoji mehanizam kojim će se novododeljeni zajam kasnije (kada proces kojem je dodeljen završi svoj rad) vratiti i biti na raspolaganju drugim procesima. Ovaj algoritam je teorijski dobar, ali u praksi ima ozbiljna ograničenja od kojih je najveće to što se za sve procese unapred mora znati koji su njihovi maksimalni zahtevi, a to u realnim sistemima najčešće nije poznato.

Algoritam provere da li je stanje bezbedno:

```
radni = raspolozivo;

gotov[i]=0, za i=0,1,...,n-1

sledeci_proces_u_sekvenci:
IF (ne postoji i takvo da je gotov[i]==0 i potrebno[i]<=radni)
   idi na kraj;
ENDIF
Odaberi prvo i takvo da je gotov[i]==0 i potrebno[i]<=radni;
radni = radni + dodeljeno[i];
gotov[i]=1;
Idi na sledeci_proces_u_sekvenci;
kraj:
IF (gotov[i]==1 za svako i)
   bezbedno_stanje=true;
ELSE
   bezbedno_stanje=false;
ENDIF
```

Algoritam za dodelu resursa:

```
IF (potrebno[i]=zahtev[i])

    Doslo je do greske;

ENDIF

IF (raspolozivo<zahtev[i])

   Resursi nisu raspolozivi, proces ce morati da saceka;

ENDIF

raspolozivo = raspolozivo - zahtev[i];

dodeljeno[i] = dodeljeno[i] + zahtev[i];

potrebno[i]  = potrebno[i] - zahtev[i];
```

```
IF (provera stanja sistema je bezbedna)
    Transakcija je zavrsena.
ELSE
    Proces ce morati da saceka zahtevane resurse, a sistem se vraca na staro stanje dodele resursa;
ENDIF
```

21. Algoritam sata.

Algoritam sata je modifikacija algoritma druge šanse koja se pokazala bolje u praksi. Naime, kod algoritma druge šanse, stranica se često premešta na kraj povezane liste stranica. Pri tome, nije izvesno da će to premeštanje poboljšati efikasnost izvršavanja procesa i smanjiti nastanak promašaja stranica. Prilikom primene algoritma, svi memorijski okviri jednog procesa se posmatraju kao povezana kružna lista. Za njegovu implementaciju se koristi jedan pokazivač, koji u početku pokazuje na prvi memorijski okvir. I ovde se svakoj stranici pridružuje bit referisanosti čija vrednost ima isto značenje kao kod algoritma druge šanse. Kada su svi okviri zauzeti, pretraga za stranicom koju treba izbaciti kreće od memorijskog okvira na koji pokazuje pokazivač. Ako je bit referisanosti te stranice 1, on se postavlja na 0, a pokazivač se pomera za jedno mesto. Kada se naiđe na stranicu čiji je bit referisanosti 0, stranica se izbacuje iz memorije, a na njeno mesto se učitava zahtevana stranica. Na kraju, pokazivač se pomera za jedno mesto unapred, odakle će krenuti naredna potraga za stranicom koja treba da bude žrtvovana. Kod ovog algoritma se, takođe, javlja problem kada su svi bitovi referisanosti postavljeni na 1.

22. Asocijativna memorija. Sta je i gde se koristi?

Asocijativna memorija je veoma brza memorija koja se koristi za potrebe straničenja. Malo je sporija od registara, ali je većeg kapaciteta, pa se deo tabele stranica može smestiti u nju. Čuva podatke u parovima (*ključ, vrednost*), a pretraga se vrši simultano nad svim ključevima. Ključevi su redni brojevi stranica, a vrednosti su odgovarajuće adrese memorijskih okvira.

Kada procesor pokuša da pristupi nekoj logičkoj adresi, najpre određuje broj stranice koja sadrži tu adresu. Zatim se vrši pretraga asocijativne memorije. Ako je pretraga bila uspešna, procesor može odmah da pristupi lokaciji u memoriji na koju se logička adresa odnosi. U suprotnom, kaže se da je došlo do promašaja, tako da je potrebno da se pretraži deo tabele stranica koji je u memoriji.

Procenat uspešnosti pronalaska željenih podataka u asocijativnoj memoriji naziva se nivo pogotka (hit ratio).

23. Prodiskutovati LFU i MFU.

Osnovna ideja *LFU* (Least Frequently Used) algoritma je da se iz memorije izbaci najmanje često korišćena stranica, odnosno stranica koja od svih stranica u memoriji ima najmanju frekvenciju korišćenja. Motivacija za takav postupak leži u težnji da se stranice koje se često koriste drže u memoriji, a one koje se ne koriste toliko često drže u sekundardnoj memoriji. Implementacija ovog algoritma je hardverski zahtevnija jer treba voditi računa o frekvenciji pojavljivanja, odnosno stalno ažurirati i sortirati podatke. Takođe, ostaje otvoreno pitanje u kolikom vremenskom periodu treba meriti referisanje na stranice. Velika mana ovog pristupa leži u lošem tretmanu stranica koje su tek ušle u sistem, ali su veoma potrebne i često će se koritiiti u budućnosti. Njihova frekvencija je mala, pa će one biti kandidati za izbacivanje. Drugi problem ovog algoritma javlja se u slučajevima kada

se pojedine stranice često koriste u određenom vremenskom periodu, a kasnije neće biti potrebne. Njihova frekvencija će biti velika, pa će one ostati u memoriji još neko vreme, iako neće biti potrebne procesu u budućnosti.

MFU (Most Frequently Used) algoritam podrazumeva da se iz memorije izbacuju stranice sa najvećom frekvencijom korišćenja. On se zasniva na ideji da će u budućnosti biti potrebnije stranice koje su tek ušle u sistem i imaju manju frekvenciju korišćenja. U nekim situacijama ovaj algoritam daje odlične rezultate, međutim, susreće se sa različitim problemima. Jedan od njih nastaje kada program tokom celog svog izvršavanja često koristi neku stranicu, pa se tada ta stranica često izbacuje.