**1 - Uvod i pregled OS**

* **Operativni sistem** predstavlja organizovanu kolekciju programa koji upravlja izvršavanjem aplikativnih programa i služi kao interfejs između programa i hardvera računara. Ima tri osnovna cilja: **Pogodnost** - Da omogući lako i pogodno korišćenje računara; **Efikasnost** - Da obezbedi efikasno korišćenje i upravljanje resursima računara; **Mogućnost** **razvoja** - Da obezbedi osnovu za efikasan razvoj, testiranje i uvođenje novih funkcija sistema. Korisnici interaguju sa OS-om pomoću komandi komandnog jezika (shell-a), ili preko elemenata grafičkog korisničkog interfejsa (GUI). OS upravlja resursima računarskog sistema. Računarski resursi su hardverski uređaji (procesor, memorija, štampač, disk, kamera,…) ili strukture podataka (datoteka, semafor, slog u bazi podataka, bafer poruka,…) koji su na raspolaganju korisnicima i programima.
* Više programa koji se konkurentno izvršavaju mogu deliti (zajednički koristiti) resurse. Postoje dva načina deljenja (multipleksiranja) resursa: **U** **vremenu** – različiti programi ili korisnici koriste resurs naizmenično. Na primer, na računaru sa jednim CPU i više programa, operativni sistem dodeljuje CPU jednom programu, zatim drugom, itd. eventualno opet prvom (drugi primer: štampač). Određivanje vremenske raspodele resursa, kome i koliko vremena će resurs biti dodeljen, je zadatak OS-a; **U** **prostoru** – svaki program ili korisnik dobija jedan deo resursa. Na primer, glavna memorija je obično podeljena na više trenutno aktivnih programa, a OS vodi računa o prostornoj raspodeli i zaštiti od pristupa tuđem delu memorijskog prostora. Kod **korisničkog moda** delovi memorije su zaštićeni I neke instrukcije se ne mogu koristiti, dok kod **kernel moda** to nije slučaj.
* **Multiprogramiranje** (multitasking) – Operativni sistem istovremeno smešta u memoriju više poslova; u jednom trenutku samo jedan od poslova se izvršava na CPU, ukoliko se blokira izvršenjem U/I operacije (npr. čitanje podataka sa diska), aktivira se planiranje poslova.
* **Planiranje** **poslova** (Job scheduling) – OS mora da iz skupa svih poslova izabere one koji će biti smešteni u memoriju i odrediti jedan koji će se izvršavati - planiranje CPU (CPU scheduling).
* **Monolitna arhitektura** - OS je implementiran kao jedan proces gde svi elementi procesa dele isti adresni prostor i sve se izvršava u modu kernela. **Mikrokernel arhitektura** – samo neke suštinske funkcije su dodeljene kernelu kao što su adresni prostori, interprocesne komunikacije (IPC) i planiranje poslova. Ostale servise OSa izvršavaju procesi koji rade na korisničkom nivou.
* **Višenitnost** (Multithreading) je tehnika deljenja procesa u niti koje se izvršavaju istovremeno. **Nit** je jedinica izvršenja koja sadrži kontekst procesora i deo podataka za stek. Izvršava se sekvencijalno i može se prekinuti. **Proces** je kolekcija jedne ili više niti i sistemskih resursa kao što su podaci, datoteke i UI uređaji.
* **Simetrično multiprocesiranje** se koristi kod sistema sa više procesora koji koriste istu glavnu memoriju i UI resurse. Svi procesori mogu izvršavati iste funkcije. U odnosu na jednoprocesorski sistem ima sledeće prednosti: Performanse – ako se delovi posla mogu izvšavati paralelno; Raspoloživost – u slučaju otkaza jednog procesora sistem ne mora da prestane sa radom; Inkrementalno povećanje – performanse se mogu povećati dodavanjem dodatnog procesora.
* **Sistemski pozivi** (system calls) obezbeđuju interfejs između aplikativnih/sistemskih programa i operativnog Sistema. Omogućuju pristup funkcijama operativnog sistema od strane korisničkih programa korišćenjem API-ja. U okviru poziva argumenti se smeštaju na stek I poziva se trap instrukcija čiji je argument kod sistemskog poziva. Trap izaziva softverski prekid, OS čuva stanje prekinutog procesa, prelazi u mod krenela I poziva funkciju koja implementira sistemski poziv.

**2 - Upravljanje procesima**

* **Proces** je osnovna jedinica izvršavanja i najmanji entitet koji se može planirati, dodeliti i izvršavati na procesoru. Sadrži više elemenata, ali dva najbitnija su programski kod i skup podataka koji se odnosi na taj kod. Dok se izvršava programski kod, proces sadrži sledeće atribute: identifikator, stanje, prioritet, PC, pokazivačima, podaci o kontekstu, itd.
* **Kontrolni blok procesa** (PCB) se kreira i upravlja od strane OS-a. Sadrži podatke tako da je moguće prekinuti i kasnije nastaviti izvržavanje procesa, tj. Identifikatore procesa (PID, PPID), informacije o stanju procesora (registri, stack pointer, Program Status Word) i informacije za upravljanje procesom (prioritet, informacije vezane za planiranje, privilegija procesa, vlasništvo i korišćenje resursa, pokazivači na tabele koje opisuju memoriju dodeljenu procesu). Kada je proces prekinut, vrednosti PC-a i registra procesora se snimaju u odgovarajuci PCB, a stanje procesa se menja na odgovarajuću vrednost. PCB su povezani u redovima i tabelama koji se često implementiraju kao lančane liste.
* **Praćenje procesa** (trace) je sekvenca instrukcija koje se izvršavaju za određeni proces. **Dispatcher** je program koji prebacuje procesor sa jednog procesa na drugi.
* **Model sa dva stanja** – proces može biti u jednom od dva stanja: izvršava se (1) i ne izvršava se (2). Kada OS kreira proces, kreira se PCB sa stanjem procesa 2. Proces postoji i čeka izvršenje. Nakon određenog vremena proces koji se trenutno izvršava se prekida, status mu se postavlja na 2 i dispatcher bira drugi proces za izvršenje čiji se status postavlja na 1.
* **Model sa pet stanja** – U slučaju da su svi procesi uvek spremni na izvršavanje može se primeniti FIFO red za njihovo izvršavanje, ali to nije uvek slučaj. Bolje rešenje je podela stanja ne izvršava se na dva stanja: **spreman**, koji je u glavnoj memoriji i čekapriliku za izvršenje i **blokiran**, koji je u glavnoj memoriji i ne može da se izvršava dok se ne desi neki događaj. Dodaju se i dva dodatna stanja, novi (proces je kreiran ali još nije primljen od strane OS-a) i završen (izbačen iz skupa izvršnih procesa).
* **Suspendovani procesi** – Često se dešava da zbog razlike u brzini procesora i IO procesi čekaju na završetak IO operacija. Neki od blokiranih procesa se suspenduju, tj. prebacuju iz glavne memorije u sekundarnu memoriju. Da bi se ovo omogućilo dodaju se još dva stanja: **blokiran/suspendovan** (proces je u sekundarnoj memoriji i čeka događaj)i **spreman/suspendovan** (proces je u sekundarnoj memoriji ali je dostupan za izvršenje čim se učita u glavnu memoriju).
* OS konstruiše odgovarajuće tabele za informacije o svakom entitetu kojim upravlja. Mogu biti: **Memorijske tabele** – sadrže podatke o glavnoj i sekundarnoj memoriji; **IO tabele** – za upravljanje IO uređaja; **Tabele fajlova** – sadrže informaciju o postojanju fajlova, njihovoj lokaciji u sekundarnoj memoriji, status, itd; **Tabele procesa –** sadrže sve PCB-ove procesa.
* Većina procesora podržava bar dva moda izvršavanja, **korisnički mod** sa manje privilegija i **kernel mod** sa više privilegija. Određene instrukcije se mogu izvršiti samo u modu sa više privilegija, kao što su čitanje ili menjanje kontrolnih registara kao što su PSW, primitivne IO instrukcije i instrukcije upravljanja memorije. Nekim delovima memorije je moguće pristupiti samo u tom modu.
* **Kreiranje procesa** se vrši u sledećem redosledu: dodela identifikatora procesu, dodela memorijskog prostora za sve elementi slike procesa, inicijalizacija PCB-a procesa, uspostavljanje veza procesa i kreiranje ili proširenje ostalih struktura podataka.
* **Zamena procesa**, jezamena aktivnog procesa jednim od procesa spremnih za izvršenje. Zamenu mogu uzrokovati događaji kao što su: **Prekidi** – dešavaju se zbog eksternih događaja u odnosu na izvršenje procesa. Mogu ih uzrokovati IO događaji, memorijske greške, isticanje dodeljenog procesorskog vremena, itd; **Trap** – dešava se ako je došlo do fatalne greške ili izuzetka prilikom izvršenja tekuće instrukcije procesa; **Sistemski poziv** – zamena uzrokovana zahtevom programa koji se izvršava.
* **Prekid procesa -** procesor postavlja PC na početnu adresu odgovarajućeg hendlera, prebacuje se u kernel režim, čuva kontekst procesa u njegovom PCB-u, izvršava rutinu hendlera i obnavlja kontekst procesa ako nastanak prekida ne zahteva zamenu procesa.
* **Izvršenje OS-a kao kernel bez procesa** – koncept procesa važi samo za korisničke programe. Kod OS-a se izvršava u kernel modu kao poseban entitet u glavnoj memoriji.
* **Izvršenje OS-a unutar korisničkih procesa** – sav softver OS-a se izvršava u kontekstu korisničkog procesa. Kod OS-a i njegovi podaci su u adresnom prostoru koji deli sa svim korisničkim procesima. Kada se izvršava kod OS-a, procesi se izvršavaju u kernel modu.
* **OS zasnovan na procesima** – OS je implementiran kao kolekcija sistemskih procesa. Softver koji je deo kernela se izvršava u kernel modu, ali su glavne funkcije kernela organizovane kao zasebni procesi. Ovakva implementacija je korisna u multiprocesorskom ili višeračunarskom okruženju.

3 – Niti

* **Nit** predstavlja entitet koji se raspoređuje i izvršava na CPU. Svaki proces ima sopstveni adresni prostor i jednu nit izvršenja. U modernim OS-ima jedan proces može imati više niti izvršenja koje se izvode paralelno. **Višenitnost** omogućava podelu posla koji je dodeljen procesu na više niti izvršenja tako da svaka nit preuzima deo posla.
* Svaki proces sadrži virtualni adresni prostor gde se nalazi slika procesa (program, podaci, stek i atributi PCB-a) i zaštićen pristup procesoru i drugim procesima, datotekama i UI resursima. Unutar procesa svaka nit poseduje stanje izvršenja niti, konteskt niti, stek, memoriju za lokalne promenljive i pristup memoriji i resursima procesa koje deli sa svim ostalim nitima.
* Kao i kod procesa, ključna stanja niti su Izvršava se, Spremna i Blokirana. Nema potrebe za suspendovano stanje zato što niti dele adresni prostor procesa. Postoje 4 operacije promene stanja nad nitima: **Spawn** (kreiranje) – kada je proces kreiran, kreira se i nit za taj proces. Nit unutar procesa može kreirati nit unutar istog procesa. Novoj niti se dodeljuje sopstveni kontekst, prostor u steku i ubacuje se u red spremnih; **Block** (blokiranje) – kada nit treba da čeka na neki događaj, blokira se. Procesor zatim prelazi na izvršavanje druge spremne niti unutar istog ili drugog procesa; **Unblock** (deblokiranje) – kada nastane događaj na koju nit čeka, nit se premešta u red spremnih; **Finish** (završetak) – kada se nit završi, njen kontekst i stek se dealociraju.
* **Niti na nivou korisnika (ULT)** – sav posao upravljanja niti izvršava aplikacija i kernel nije svestan postojanja niti. Svaka aplikacija može se implementirati kao višenitna upotrebom odgovarajuće biblioteke koja sadrži rutine za upravljanje ULT. Proces se ne mora prebaciti na kernel mod za upravljanje nitima. Svaka aplikacija može da koristi algoritam planiranja izvršenja koji joj najviše odgovara. ULT se mogu izvršiti na svakom OS-u. Ukoliko jedna nit pozove sistemski poziv koji je blokirajući, sve niti unutuar procesa biće blokirane. Niti se ne mogu istovremeno izvršavati na različitim procesorima.
* **Niti na nivou kernela (KLT)** – kernel upravlja nitima. Ne postoji kod za upravljanje nitima na aplikacionom nivou, već samo API za rad sa nitima. Kernel može istovremeno planirati više niti istog procesa na više procesora. Blokiranje se vrši na nivou niti, ne na nivou procesa. Promena niti unutar procesa zahteva promenu moda na kernel mod, te je i upravljanje KLT niti sporije od ULT.
* **Kombinovana implementacija** – kreiranje niti se izvršava u korisničkom prostoru, kao i većina planiranja i sinhronizacije niti unutar aplikacije. ULT-ovi aplikacije se mapiraju na isti ili manji broj KLT koji podešava korisnik. Ovakav pristup kombinuje prednosti ULT i KLT dok minimizuje nedostatke.

**4 – Uzajamno isključivanje i sinhronizacija**

* **Konkurentnost** se javlja u 3 različita konteksta: **Višestruke aplikacije** - Multiprogramiranje obezbeđuje deljenje vremena obrade između aktivnih aplikacija; **Struktuirane aplikacije** - Aplikacije mogu biti programirane kao skup konkurentnih procesa; **Struktura operativnog sistema** - Operativni sistem se implementira kao skup procesa ili nit. Rešenje za konkurentnost treba da obuhvati komunikaciju među procesima, deljenje resursa, sinhronizaciju više procesa i alokaciju vremena procesa.
* **Interakcije procesa** – postoji tri moguće vrste interakcije procesa u odnosu na njihovo znanje o postojanju drugih procesa: procesi nisu svesni postojanja drugih procesa (procesi koji nisu namenjeni da rade zajedno. OS mora da reguliše pristupe zajedničkim resursima); procesi su indirektno svesni postojanja drugih procesa (procesi koji nisu svesni postojanja drugih procesa preko PID-a, ali dele pristup nekom objektu, kao što je IO bafer); procesi koji su direktno svesni postojanja drugih procesa (procesi komuniciraju preko PID-a i dizajnirani su da rade zajedno).
* **Race condition** (uslov trke) – javlja se kada više procesa ili niti čitaju ili upisuju stavke deljenih podataka tako da rezultat zavisi od redosleda izvršavanja instrukcija.
* **Mutual exclusion** (uzajamno isključivanje) – dešava se kada dva ili više procesa zahtevaju pristup jednom nedeljivom resursu. Takav resurs se naziva kritičan resurs a deo programa koji ga koristi kritična sekcija. Samo po jedan program može da pristupa kritičnoj sekciji.
* **Deadlock** (uzajamno blokiranje) – dva ili više procesa ne mogu da nastave sa radom jer svaki čeka da jedan od preostalih nešto uradi. Neka su data dva procesa P1 i P2 i resursi R1 i R2 gde svaki proces treba da pristupi svakom resursu. Može se doći u situaciju da OS dodeli R1 procesu P2 i R2 procesu P1. Oba procesa čekaju drugi resurs i nijedan neće osloboditi resurs koji mu je dodeljen dok ne dobije resurs koji nedostaje. Ovi procesi su u uzajamnom blokiranju.
* **Starvation** (gladovanje) – neka su data tri procesa P1, P2 i P3 i resurs R gde je prvobitno R dodeljen procesu P1 a P2 i P3 čekaju na R. Kada P1 izađe iz kritične sekcije, ili P2 ili P3 bi trebalo da dobiju pristup resursu R. Pretpostavimo da OS daje pristup R procesu P3 i da P1 ponovo zahteva R pre nego što P3 izađe iz kritične sekcije. Ako OS ponovo dodeli pristup R procesu P1 nakon što P3 završi i potom naizmenično daje pristup samo procesima P1 i P3, P2 nikada neće dobiti pristup resursu.
* **Hardversko rešenje uzajamnog isključivanja** – postoje dva pristupa: **Zabrana prekida** – radi na jednoprocesorskim sistemima zato što konkurentni procesi ne mogu da se izvršavaju istovremeno i procesi će se izvršavati sve dok proces ne pozove servis OS-a ili se ne prekine. Iz tog razloga je za uzajamno blogiranje dovoljno zabraniti prekidanje procesa. Pošto kritična sekcija ne može biti prekinuta, uzajamno isključivanje je zagarantovano. Ovaj pristup ne radi na višeprocesorskom sistemu. **Specijalne mašinske instrukcije** – na hardverskom nivou pristup memorijskoj lokaciji isključuje bilo koji pristup na istoj lokaciji. Implementirane su mašinske instrukcije koje atomično, *kao jedan korak koji ne može biti prekinut*, izvršavaju dve akcije kao što su čitanje i pisanje jedne memorijske lokacije. Izvršenje takve instrukcije je uzajamno isključivo čak i kod multiprocesora, jednostavno je i može se primeniti na više kritičnih sekcija, ali je moguće uzajamno blokiranje i gladovanje.
* **Semafori –** princip rada semafora je da dva ili više procesa mogu da rade zajedno koristeći jednostavne signale tako da proces može biti primoran da stane dok ne primi određeni signal. Semafor može biti inicijalizovan na nenegativnu integer vrednost. Za prenos signala preko semafora s proces izvršava operaciju semSignal(s) koja inkrementira vrednost semafora . Da primi signal preko semafora s proces izvršava operaciju semWait(s) koja dekrementira vrednost semafora i proces postaje blokiran sve dok odgovarajući signal ne bude primljen. Semafori mogu biti binarni i brojački. Može biti teško napraviti korektan program korišćenjem semafora zato što operacije mogu biti rasipane kroz program i može biti teško videti njihov uticaj.
* **Monitori** imaju istu funkcionalnost kao semafori ali su lakši za upravljanje. Monitor sadrži jednu ili više procedura, inicijalizacionu sekvencu i lokalne promenljive kojima se može pristupiti samo iz procedura monitora. Proces može pozvati neku od procedura monitora da bi ušao u njega. U jednom trenutku samo jedan proces se može izvršavati unutar monitora dok svaki drugi proces koji poziva proceduru monitora čeka da monitor postane dostupan. Monitor osigurava uzajamno isključivanje i sinhronizaciju korišćenjem uslovnih promenljivih koje monitor sadriži. Tim promenljivama se upravlja funkcijama cwait(c) kojom se blokira izvršenje procesa koji poziva funkciju uslovom c, i csignal(c) kojom se nastavlja izvršenje procesa (ukoliko ih ima više bira se jedan) blokiranim funkcijom cwait uslovom c.
* **Message passing** (prenos poruka) je metoda komunikacije među procesima. Obično se za realizaciju koriste dve operacije: send(destination, message) i receive(source, message). Ovo je minimalni skup operacija potrebnih da bi procesi komunicirali preko prenosa poruka. Proces šalje poruku u vidu *message* drugom procesu označenim kao *destination*. Proces prima informacuju izvršavanjem *receive* operacije od procesa označenim kao *source*. Za komunikaciju između dva procesa potrebna je neka vrsta sinhronizacije, formata poruke i adresiranja procesa.
* **Format poruke** zavisi od cilja sistema za prenos poruka i od toga da li se sistem izvršava na jednom računaru ili distribuiranom sistemu. Poruke mogu biti fiksne dužine, čime se smanjuje procesiranje i skladištenje poruka, ili promenljive dužine koje se dele na dva dela, header koji sadrži informaciju o poruci (tip, odredište, izvor, dužina, kontrolna informacija) i sadržaj poruke.
* **Sinhronizacija u prenosu poruka** – prenos poruka između dva procesa podrazumeva neki nivo sinhronizacije, na primer da primalac ne može da primi poruku dok je neki proces ne pošalje. Kada se izvrši operacija send postoje dva slučaja, ili je izvorišni proces blokiran dok poruka ne bude primljena ili nije. Kada se izvrši operacija receive postoje dva slučaja, poruka je primljena i izvršenje se nastavlja ili ako ne postoji poruka koja čeka proces je blokiran dok poruka ne stigne ili se proces nastavlja zanemarujući pokušaj da primi poruku. Odatle i izvorišni i odredišni proces mogu biti blokirajući ili neblokirajući. Najčešće su tri kombinacije, mada će sistem obično imati implementiranu samo jednu: **blokirajući send, blokirajući receive** (izvor i odredište se blokiraju dok poruka ne bude primljena), **neblokirajući send, blokirajući receive** (izvor nastavlja da se izvršava, odredište se blokira dok ne primi poruku), **neblokirajući send, neblokirajući receive** (ni izvor ni odredište ne moraju da čekaju na poruku).
* **Adresiranje procesa** omogućava da send operacija zna koji proces treba da primi poruku. U **direktnom adresiranju** send operacija sadrži identifikator odredišnog procesa dok receive operacija ili unapred sadrži identifikator izvorišnog procesa ili se taj identifikator postavlja tek nakon prijema poruke. **Indirektno adresiranje** podrazumeva da se poruke ne šalju direktno od izvora do odredišta, već se šalju u deljenu strukturu podataka sačinjenu od redova koja privremeno zadržava poruke. Kod ovakvog adresiranja odnos izvora i odredišta može biti 1:1, N:1, 1:N i N:M.

**5 – Uzajamno blokiranje i gladovanje**

* **Uzajamno blokiranje** je trajno blokiranje skupa procesa koji zahtevaju iste resurse ili međusobno komuniciraju. Skup procesa ja uzajamno blokiran kada svaki proces u skupu čeka neki događaj koji samo može da aktivira neki od procesa iz tog skupa. Iz tog razloga je uzajamno blokiranje trajno, zato što se nijedan od tih događaje neće dogoditi.
* **Resurs** je bilo koji objekat koji se može dodeliti procesima. Postoje dve kategorije resursa: ponovo upotrebljivi resursi i potrošni resursi.
* **Ponovo upotrebljivi resursi** nakon upotrebe od strane jednog procesa mogu biti korišćeni od strane drugog. Primeri ovakvih resursa su procesori, IO kanali, glavna i sekundarna memorija, strukture podataka kao što su fajlovi, baze podataka, semafori, itd.
* **Potrošni resursi** su oni koji se mogu kreirati i uništiti. Obično ne postoji ograničenje u broju takvih resursa koje proces može da napravi. Primeri ovakvih resursa su prekidi, signali, poruke i informacija u IO baferima.
* **Graf dodele resursa** je orijentisani graf koji modeluje stanje sistema resursa i procesa gde je svaki proces i svaki resurs predstavljen čvorom. Grana usmerena od procesa do resursa govori da taj proces zahteva taj resurs, ali resurs još nije dodeljen. Unutar resursa, svaka tačka predstavlja po jednu instancu resursa. Grana od resursa do procesa govori da je resurs dodeljen procesu. Ako u grafu postoje ciklusi, može se javiti uzajamno blokiranje. Ako svaki tip resursa ima 1 instancu, ciklus u grafu je dovoljan uslov za uzajamno blokiranje.
* **Postoje 4 uslova za uzajamno blokiranje:** uzajamno isključivanje, drži i čekaj (proces drži dodeljene resurse dok čeka dodeljivanje ostalih resursa), bez prekidanja (nijedan resurs ne se ne može oduzeti procesu koji ga drži) i kružno čekanje (petlja u grafu dodele resursa). Da bi nastalo uzajamno blokiranje moraju biti prisutna sva 4 uslova.
* **Prevencija uzajamnog blokiranja** podrazumeva sprečavanje nastanka jednog od 4 uslova uzajamnog blokiranja. **Uzajamno isključivanje** se mora zadržati za nedeljive resurse tako da se ne može sprečiti. **Uslov drži i čekaj** se može sprečiti korišćenjem pristupa da proces pre izvršenja zahteva sve resurse i da se blokira dok ih ne dobije ili da oslobodi sve resurse pre zahtevanja dodatnih. **Uslov bez prekidanja** se može sprečiti pristupom da ako proces drži resurse a zahteva neki drugi resurs koji mu se ne može dodeliti, oslobadja sve resurse i čeka da svi zahtevani resursi postanu slobodni. Alternativno ako proces zahteva resurs koji drži drugi proces, OS može da zahteva od drugog procesa da oslobodi zahtevani resurs. **Uslov kružnog čekanja** se može sprečiti uvođenjem linearnog uređenja za sve tipove resursa, tj. ako proces drži resurs R2 može da zahteva samo resurse R3, R4, itd.
* **Bankarov algoritam** proverava da li dodela resursa vodi u bezbedno stanje, odnosno pronalazi sekvencu alokacije resursa koja ne dovodi do uzajamnog blokiranja (svi procesi se mogu izvršiti do kraja). Dovodi do izbegavanja uzajamnog blokiranja. Neka su date matrica maksimalnih zahteva za resursima C, matrica resursa trenutno dodeljenih procesima A i vektor raspoloživih resursa V. Sekvenca alokacije se određuje nalaženjem matrice C-A i primenom algoritma: Ako je C-Ai<=V onda V=V+Ai. Mora se proći kroz sve redove matrice A tako da uslov bude ispunjen.
* **Otkrivanje uzajamnog blokiranja** – sistem dozvoljava da se uzajamno blokiranje pojavi a onda uključuje mehanizme da otkrije uzajamno blokiranje i oporavi sistem. Provera se može vršiti pri svakom zahtevu za resursima ili povremeno u zavisnosti od verovatnoće pojave. Algoritam označava sve procese koji nisu uzajamno blokirani: označiti sve procese čija vrsta u matrici A sadrži sve nule i inicijalizovati vektor W koji je jednak vektoru V. U matrici C naći neoznačeni proces koji zadovoljava uslov Ci<=W. Ako je uslov ispunjen, označiti proces Pi i vektoru W dodati Ai. Uzajamno blokiranje postoji ako postoje neoznačeni procesi po završetku algoritma.
* **Oporavak iz uzajamnog blokiranja** – prekinuti sve procese koji su uzajamno blokirani, vratiti sve blokirane procese do prethodno definisane kontrolne tačke i restartovati sve procese. Sukcesivno prekidati procese koji su uzajamno blokirani sve dok postoji uzajamno blokiranje (redosled se određuje na osnovu najmanje cene izvršenja). Sukcesivno oduzimati resurse procesima sve dok postoji uzajamno blokiranje.

**6 – Upravljanje memorijom**

* **Upravljanje memorijom** treba da zadovolji sledeće uslove: relokaciju (proces se može po potrebi premestiti iz na disk i ponovo vratiti u memoriju počev od nove memorijske adrese), zaštitu memorije (zaštita pristupa memorijskom prostoru jednog procesa od strane drugog, kao i zabrana pristupa memorijskom prostoru OS-a), zajedničko korišćenje istog dela glavne memorije, logičku organizaciju (organizacija adresnog prostora i podela programa u module) i fizičku organizaciju (glavna i sekundarna memorija).
* **Fiksna podela memorije** je podela memorije na fiksne particije pri startovanju sistema. Particije mogu biti istih ili različitih veličina. Kod particija istih veličina program može biti preveliki da bi stao u jednu particiju, tada se koristi tehnika *overlay* tako da samo deo programa može da bude u glavnoj memoriji. Korišćenje memorije je neefikasno zato što bilo koj program, nezavisno od veličine, zauzima celu particiju. Ovo se zove **interna fragmentacija**. Ovi problemi se rešavaju korišćenjem particija različitih veličina.
* **Algoritam raspoređivanja kod fiksne podele memorije** – kod particija istih veličina nije bitno koja se particija koristi. Kod particija različitih veličina postoje dva načina raspoređivanja: proces se smešta u najmanju particiju u koju može da stane pomoću reda čekanja koji sadrži sve procese koji bi zauzeli tu particiju, ili pomoću samo jednog reda koji sadrži sve procese koji čekaju (uvek se bira najmanja slobodna particija koja može da sadrži proces).
* **Dinamička podela memorije** rešava nedostatke fiksne podele memorije. Particije su promenljive veličine i broja. Kada proces uđe u glavnu memoriju dodeljuje mu se tačno onoliko memorije koliko mu je potrebno. Ovo može dovesti do toga da proces ima dovoljno memorijskog prostora za izvršenje, ali je prostor nekontinualan. Ovo se zove **eksterna fragmentacija**. Rešenje ovoga je kompakcija, odnosno prebacivanje memorijskih delova procesa prema jednom kraju adresnog prostora a slobodnih delova memorije ka drugom. Evidencija slobodnih i zauzetih memorijskih particija se može izvršiti preko Bitmape (memorija podeljena na alokacione jedinice od nekoliko KB: 0 slobodna, 1 zauzeta) ili lančane liste (elementi liste sadrže oznaku da li se radi o procesu ili slobodnoj particiji, početnu adresu, veličinu i link).
* **Algoritmi dodele dinamičkih particija** – postoje tri algoritma, best-fit (bira se najmanja odgovarajuća particija), first-fit (pretražuje se memorija i bira se prva slobodna particija koja može da sadrži proces) i next-fit (isto kao first-fit samo pretraživanje počinje od adrese gde je nađena prethodna slobodna particija).
* **Buddy system** (sistem partnera) je kompromis između fiksne i dinamičke podele memorije. Memorijski blokovi su dostupni u veličinama od 2K reči, L<=K<=U, gde je 2L najmanji blok koji je alociran a 2U najveći (nejčešće veličine cele memorije dostupne za alokaciju). Ako stigne zahtev za alokaciju memorije veličine blok se deli na pola sve dok ne bude ispunjen uslov 2N-1<=k<=2N.
* **Paging** (straničenje) – memorija je podeljena na male delove jednake veličine. Procesi su zatim podeljeni na delove određene veličine koji se zovu **stranice** koje se dodeljuju delovima memorije koji se zovu **okviri**. Usled toga neiskorišćeni delovi memorije za svaki proces nastaju usled interne fragmentacije. OS sadrži tabele stranica za svaki proces da bi omogućio nekontinualnu dodelu okvira. Unutar tabele stranica nalazi se broj okvira u kome se nalazi svaka stranica. Logička adresa se predstavlja kao 2N gde je N = P + O (P je broj bitova za broj stranice, O je broj bitova sa offset). Fizička adresa se dobija zamenom P dela logičke adrese brojem frejma u kome se stranica nalazi.
* **Segmentacija** je šema za upravljanje memorijom u kome se program deli na određeni broj segmenata koji ne moraju biti iste veličine, ali postoji maksimalna dozvoljena veličina. Koristi se tabela segmenata za svaki proces i lista slobodnih blokova memorije. Svaka stavka tabele segmenata sadrži početnu adresu segmenta i dužinu segmenta. Logička adresa sadrži broj segmenta i offset. Fizička adresa se dobija sabiranjem offset-a iz logičke adrese sa početnom adresom segmenta iz tabele segmenata. Može se javiti eksterna fragmentacija.

**7 – Virtuelna memorija**

* **Straničenje u virtuelnoj memoriji** – potrebna je tabela stranica za straničenje u virtuelnoj memoriji za svaki proces. Svaka stavka u tabeli sadrži bit koji specificira da li je stranica u glavnoj memoriji, broj okvira i druge kontrolne bitove**.** Neki procesori koriste straničenje u dva nivoa za organizaciju velikih tabela u kojima se tabela predstavlja u obliku stranica. Tada se u virtuelnoj adresi deo za broj stranice deli na dva dela.
* **Invertovane tabele stranica** – u ovakvoj tabeli postoji jedna stavka za svaki okvir u memoriji. Svaka stavka sadrži virtuelnu adresu stranice koja je smeštena u dati okvir, kao i podatke o procesu kojem ta stranica pripada. Virtuelna adresa se mapira u heš koja predstavlja indeks na invertovanu tabelu stranica čija svaka stavka predstavlja lančanu listu elementa koji su preslikani u istu heš vrednost. Pretražuje se lista i određuje se broj okvira. Da bi se izbeglo dodatno referenciranje memorije uz svaki memorijski pristup koristi se TLB.
* **TLB** (Translation Lookaside Buffer) je keš memorija koja sadrži stavke stranične tabele koje su najskorije korišćene. Kada procesor dobije virtuelno adresu, prvo proverava TLB. Ako je tražena stavka prisutna, TLB vraća broj okvira. Procesor sadrži hardver koji omogućava paralelno pretraživanje svih stavki u TLB-u. Ova tehnika se naziva **asocijativno mapiranje**.
* **Segmentacija sa straničenjem** kombinuje prednosti straničenja i segmentacije. Adresni prostor se deli na broj segmenata koji se dalje dele na broj stranica fiksne veličine koje su jednake u veličini okviru glavne memorije. Logička adresa se sastoji iz broja segmenta, broja stranice i offset-a stranice unutar specificiranog segmenta.
* Softver za upravljanje memorijom OS-a zavisi od tri stvari: da li se koriste tehnike virtualne memorije, da li se koristi straničenje ili segmentacija i koji se algoritmi za upravljanje memorije koriste. Prve dve stavke zavise od hardvera, dok je treća u domenu softvera OS-a.
* **Fetch policy** (politika donošenja stranice) odlučuje kada stranica treba da bude učitana u glavnu memoriju. Postoje dve implementacije: **Demand paging** (straničenje na zahtev) – stranica se učitava u glavnu memoriju samo kada se napravi referenca na memorijsku lokaciju unutar te stranice; **Prepaging** (prestraničenje) – učitava u glavnu memoriju više stranica nego što je trenutno potrebno.
* **Placement policy** (politika smeštanja stranice) odlučuje gde će u memoriji biti smešten deo procesa.
* **Replacement policy** (politika zamene) odlučuje koja će se stranica u glavnoj memoriji zameniti kada se učita nova stranica. Obično se bira stranica čija je najmanja verovatnoća da bude korišćena primenom algoritama LRU, FIFO, optimalni algoritam i algoritam časovnika. Stranica koja je izabrana za zamenu se stavlja u listu slobodnih stranica ukoliko stranica nije menjana ili listu modifikovanih stranica ukoliko jeste. Algoritam časovnika radi slično kao FIFO ali pored starosti uzima u obzir dodatne podatke. Kod prve varijante koristi se jedan *use* bit koji se postavlja na 1 kada se stranica učita u memoriju i kad god je referencirana. Ako je *use* bit 0, stranica je i stara i ne skoro referencirana pa se zamenjuje, ako je 1 bit se postavlja na 0, pokazivač prelazi na sledeći okvir u kružnom baferu i ispitivanje se nastavlja. Ako su bitovi svih stranica 1, algoritam postaje FIFO. Kod druge varijante pored *use* bita postoji bit promene *m* koji se postavlja na 1 kada je stranica modifikovana. Ovaj bit je potreban da kada se sadržaj stranice promeni ona ne bude zamenjena dok se ne upiše na sekundarnu memoriju.
* **Resident set management** (upravljanje rezidentnim skupom) – vodi se računa o veličini rezidentnog skupa (koliko stranica procesa treba učutati u memoriju – vrši se fiksnom alokacijom gde proces dobija fiksni broj okvira u glavnoj memoriji, ili promenljivom alokacijom gde se broj okvira dodeljenih procesu može menjati tokom izvršenja procesa) i opsegu zamene (lokalna zamena – bira samo između stranica procesa koji je napravio grešku; globalna zamena – bira sve nezaključane stranice u glavnoj memoriji kao kandidate za zamenu). Strategije za upravljanje rezidentnim skupom su fiksno dodeljivanje – lokalni opseg, promenljivo dodeljivanje – lokalni opseg i promenljivo dodeljivanje – globalni opseg. Strategija koja se primenjuje u vidu promenljivog dodeljivanja – lokalni opseg je strategija radnog skupa (radni set sa parametrom Δ u virtuelnom trenitku t koji se označava kao W(t, Δ) je skup stranica procesa koji su bili referencirani u poslednjih Δ jedinica virtuelnog vremena. Nadgleda se radni skup svakog procesa, periodično se uklanjaju iz rezidentnog skupa procesa stranice koje nisu u radnom skupu pomoću LRU. Prilikom greške nalazi se stranica koja nije u radnom skupu i zamenjuje se stranicom iz memorije).
* **Cleaning policy** (politika čišćenja) je suprotno od politike donošenja stranica. Određuje kada promenjena stranica treba da bude upisana u sekundarnu memoriju. Koriste se dve tehnike: čišćenje na zahtev (stranica se upisuje samo kada se izabere za zamenu) i predčišćenje (promenjene stranice se upisuju pre nego što su njihovi stranični okviri potrebni). Bolji pristup je baferovanje stranice – čiste se samo stranice koje se mogu zameniti ali se operacije čišćenja i zamene razdvajaju. Stranice se smeštaju na listu promenjenih ili nepromenjenih. Stranice na listi promenjenih se periodično upisuju na sekundarnu memoriju u paketima a zatim pomeraju na listu nepromenjenih.
* **Load Control** (upravljanje učitavanjem) određuje broj procesa koji će biti rezidentni u glavnoj memoriji. Ako je premali broj procesa rezidentno u bilo kom trenutku, može se desiti da su svi procesi blokirani i da je CPU nezaposlen. Tada se troši mnogo vremena na razmenu sadržaja između glavne i sekundarne memorije.

**8 – Jednoprocesorsko planiranje**

* **Dugoročno planiranje** određuje koje programe treba pustiti u sistem za izvršenje. Nakon izbora programa, program postaje proces i dodaje se u red kratkoročnog planera. Ako je više procesa aktivno svaki proces dobija manji deo CPU vremena. Odluka koji program treba sledeći da primi se vrši prema FCFS.
* **Srednjoročno planiranje** odlučuje koje procese treba preneti na disk ili koje uneti u memoriju sa diska za potrebe upravljanja multiprogramiranjem. Sarađuje sa sistemom za upravljanje memorijom.
* **Kratkoročno planiranje** obavlja dispatcher (CPU planer). Odlučuje koji će se proces sledeći izvršavati i alocira procesorsko vreme tako da optimizuje ponašanje sistema. Kada dispatcher izabere proces koji će se izvršavati neophodno je izvršiti promenu konteksta i aktiviranja izabranog procesa od tačke na kojoj je prekinut. Vreme koje dispatcher potroši da bi stopirao jedan proces i startovao izabrani se naziva **dispatch latency**.
* Skup kriterijuma je ustanovljen na osnovu kojih se može evaluisati planiranje. To su: **Vreme odziva** – za interaktivne procese to je vreme koje protekne od slanja zahteva do početka prijema odgovora; **Vreme zadržavanja** (turnaround time) – vreme koje protekne od aktiviranja do završetka procesa; **Propusna moć** – broj kompletiranih procesa u jedinici vremena; **Iskorišćenost procesora** – kreće se u opsegu od 0-100%; **Vreme čekanja** – vreme koje proces provede čekajući u redu spremnih procesa.
* Svakom procesu je dodeljen prioritet i planer uvek bira proces višeg prioriteta. Umesto jednog reda koristi se skup redova u opadajućem redosledu prioriteta. Kada se treba obaviti selekcija prioriteta, planer bira proces iz prvog nepraznog reda najvišeg prioriteta. Ako je red sa najvišim prioritetom prazan, bira se red sa nižim. Može doći do gladovanja procesa u redovima nižih prioriteta.
* **Funkcija selekcije** određuje koji će proces iz reda spremnih biti izabran za izvršenje. Može biti zasnovana na prioritetu, zahtevima za resursima ili karakteristikama izvršenja procesa, u kom slučaju se izražava veličinama *w* (vreme provedeno u čekanju), *e* (vreme provedeno u izvršavanju) i s (ukupno vreme zahtevano od strane procesa za izvršenje uključujući *e*). **Mod odluke** definiše vremenske trenutke u kojima se koristi funkcija selekcije. Može biti bez prekidanja (kada je proces u stanju izvršenja, izvršava se sve dok se ne terminira ili blokira da bi čekao IO ili da zahteva neki servis OS-a) ili sa prekidanjem (proces koji se trenutno izvršava može biti prekinut i pomeren u red spremnih).
* **FCFS** (First Come First Served) je šema koja striktno koristi red. Kada proces postane spreman na izvršenje dodaje se u red spremnih. Kada se završi proces koji se trenutno izvršava, proces koji je najduže bio u redu se bira za izvršenje – funkcija selekcije je max[*w*], mod odluke je bez prekidanja.
* **Round Robin** – proces se izvršava određeni vremenski kvant nakon koga se prekida i vraća u red spremnih. Vremenski kvant mora biti veći od vremena potrebnog za rukovanje prekidom i promenu konteksta - funkcija selekcije je max[*w*], mod odluke je sa prekidanjem (na određenim intervalima).
* **SPN** (Shortest Process Next) – proces koji je spreman na izvršenje i ima najmanje očekivano vreme izvršenja se bira kao sledeći za izvršenje – funkcija selekcije je min[*s*], mod odluke je bez prekidanja.
* **SRT** (Shortest Remaining Time) je verzija SPN sa prekidanjem. Planer uvek bira proces sa najmanjim očekivanim preostalim vremenom izvršenja. Kada se doda novi proces u red spremnih, on može imati kraće preostalo vreme izvršenja od onog koji se trenutno izvršava. Planer može prekinuti trenutni proces kada takav novi postane spreman – funkcija selekcije je min[*s - e*], mod odluke je sa prekidanjem (kada novi proces postane spreman na izvršenje).
* **HRRN** (Highest Response Ratio Next) bira proces sa najvišim odnosom *R = (w+s)/s*. Ako je proces sa ovom vrednošću odmah primljen na izvršenje, *R* je jednak normalizovanom vremenu zadržavanja – funkcija selekcije je max[*(w+s)/s*], mod zadržavanja je bez prekidanja.
* **Feedback** koristi dinamičke prioritete i planiranje sa prekidanjem. Svaki novi proces se dodaje u red spremnih sa najvišim prioritetom R0. Nakon prolaska prvog vremenskog kvanta, prioritet mu se smanjuje i prebacuje se u R1. Prioritet se smanjuje nakon prolaska svakog vremenskog kvanta. Ovo omogućava da noviji i kraći procesi imaju veći prioritet u odnosu na starije i duže procese, mada to može dovesti do gladovanja dužih procesa. Svi redovi osim reda sa najnižim prioritetom koriste FCFS, dok red sa najnižim prioritetom koristi Round Robin – mod zadržavanja je sa prekidanjem (na određenim intervalima).

**9 – Upravljanje IO i planiranje diska**

* Eksterni uređaji koji učestvuju u IO operacijama sa računarskim sistemima mogu da se grupišu u tri kategorije: **čitljivi za čoveka** (koriste se za komunikaciju sa korisnikom), **čitljivi za mašinu** (koriste se za komunikaciju sa računarskom opreemom) i **komunikacioni** (koriste se za komunikaciju sa udaljenim uređajima). Razlike između klasa uređaja mogu biti u brzini prenosa podataka, primeni, složenosti upravljanja, jedinici prenosa podataka, reprezentacije podataka i uslovima greške.
* Postoje tri tehnike za izvođenje IO: **Programirani IO –** procesor izdaje komandu IO modulu od strane procesa, tada je proces u radnom čekanju (busy wait) dok se IO operacija ne završi; **IO vođen prekidima** – procesor izdaje IO komandu od strane procesa. Ako je komanda neblokirajuća, procesor nastavlja da izvršava intrukcije procesa koji je izdao komandu, ako je blokirajuća onda sledeća instrukcija koju procesor izvrši je iz OS-a, tako da će se trenutni proces blokirati; **IO korišćenjem DMA** (Direct Memory Access) – DMA modul kontroliše razmenu podataka između glavne memorije i IO modula. DMA je dominantan oblik prenosa podataka i mora biti podržan od strane OS-a.
* **DMA** je modul koji oponaša procesor i preuzima kontrolu nad sistemskom magistralom. Kada procesor želi da čita ili upisuje blok podataka, šalje odgovarajuće informacije DMA modulu i prosleđuje mu IO operaciju. DMA modul zatim prenosi ceo blok podataka reč po reč direktno u ili iz memorije bez prolazka kroz procesor. Kada je prenos završen, šalje se signal prekida procesoru.
* Pri dizajniranju upravljanja IO treba ispuniti dva uslova: **efikasnost** i **generalnost**. Efikasnost je bitna zato što su IO uređaji daleko sporiji od glavne memorije i procesora. Primenjuje se zamena procesa da bi se obezbedila zaposlenost procesora, što je takođe IO operacija. Generalnost podrazumeva da sa IO uređajima treba raditi na uniforman način, odnosno da procesi treba na uniforman način da vide IO uređaje a OS na uniforman način da njima upravlja.
* **Logički preiferni uređaj** je model logičke strukture IO koji komunicira na jednostavan način. Sadrži sledeće elemente: **Logički IO** - ovaj IO modul upravlja uređajem kao logičkim resursom. Upravlja generalnim IO funkcijama od strane korisničkih procesa; **IO uređaja** – zahtevane operacije i podaci se transformišu u odgovarajuće sekvence IO instrukcija; **Planiranje i upravljanje** – u ovom sloju se izvršava planiranje i uređenje IO operacija kao i kontrola istih., kao i rukovanje izuzecima i očitavanje i prijavljivanje IO statusa.
* **Komunikacioni port** je identičan model kao logički periferni uređaj. Razlika je u tome što je logički IO modul zamenjen komunikacionom arhitekturom koja može da se sastoji od više slojeva.
* **Fajl sistem** predstavlja strukturu upravljanja IO na sekundarnoj memoriji. Sadrži slojeve **IO uređaja** i **Planiranja i upravljanja**, kao i još tri dodatna sloja: **Upravljanje direktorijumima** – prevodi simbolička imena datoteka u identifikatore kojima se referenciraju datoteke direktno ili indirektno kroz deskriptore fajlova ili tabelu indeksa; **Fajl sistem** – obezbeđuje logičku strukturu fajlova i korisničkoh operacija; **Fizička organizacija** – prevodi logičke reference na datoteke u fizičke adrese na sekundarnoj memoriji.
* **Drajver uređaja** predstavlja program prilagođen konkretnom OS-u koji upravlja IO uređajem. Svaki drajver uređaja rukuje jednim tipon IO uređaja ili klasom sličnih IO uređaja. Deo je kernela pa može pristupiti registrima uređaja.
* **Bafer** je oblast u memoriji gde se smeštaju podaci dok se prenose između IO uređaja i memorijskog prostora procesa. Pristup baferovanju može biti **orijentisan na blokove** (podacu su smešteni u blokovima fiksne veličine i transferi se obavljaju blok po blok. Referenciranje podataka se može obaviti na osnovu broja bloka) ili **orijentisan na tokove** (podaci se prenose u tokovima bajtova - stream). Bez bafera OS direktno pristupa uređaju kada je potrebno. **Jedan bafer** funkcioniše tako što omogućava procesiranje jednog bloka podataka dok se sledeći blok učitava. **Dvostruki bafer** funkcioniše tako što proces prenosi podatke iz ili u jedan bafer dok OS ispražnjava ili popunjava drugi – *double buffering****.* Cirkularni bafer** se odnosi na implementaciju koja sadrži više od dva bafera u kom je svaki bafer jedinica u okviru cirkularnog bafera.
* **Parametri performansi diska** određuju vreme potrebno za čitanje i upis disk bloka: **Seek time** (vreme traženja) je vreme potrebno za pozicioniranje glave diska na željenu trasu; **Latency** (rotaciono kašnjenje) je freme potrebno da bi se rotiranjem diska traženi sektor našao ispod glave diska; **Transfer time** (vreme prenosa) je vreme potrebno za izvršenje prenosa. Izračunava se kao *T = b/rN* gde je *b* broj bajtova kojise prenosi, r rotaciona brzina diska, *N* broj bajtova na traci diska. **Access time** (vreme pristupa) je suma vremena traženja i rotacionog kašnjenja. **Ukupno prosečno vreme pristupa** se računa kao *Ts = Ts + 1/2r + b/rN* gde je *Ts* vreme traženja.
* **Strategije raspoređivanja diska** podrazumevaju broj algoritama planiranja za sekvencu IO zahteva**.** Neke od njih su: **FIFO** (zahtevi se obrađuju iz reda sekvencijalno**.** Analogno, postoji i LIFO), **Prioritet** (cilj nije optimizacija iskorišćenosti diska već ostalih ciljeva OS-a. Obično kraći poslovi imaju veći prioritet od dužih), **SSTF** (Shortest Service Time First – bira se IO zahtev kome je potrebno najmanje kretanje glave diska sa njegove trenutne pozicije),  **SCAN** (algoritam koji sprečava gladovanje. Glava se pomera samo u jednom pravcu sve dok ne dostigne poslednju trasu u tom pravcu ili nema više zahteva**,** tada se okreće i pomera u drugom pravcu), **C-SCAN** (SCAN kome je ograničeno kretanje u samo jednom pravcu), **N-step-SCAN** (segmentira se red zahteva za diskom u podredove dužine N. Podredovi se obrađuju jedan po jedan korišćenjem SCAN) i **FSCAN** (koriste se dva reda, u jedan se stavljaju svi zahtevi dok je drugi prazan. Tokom skeniranja svi novi zahtevi se stavljaju u drugi red).
* **RAID** (Redundant Array of Independent Disks) je skup fizičkih drajvova koje OS vidi kao jedan logički drajv. Podaci su distribuirano po fizičkim drajvovima u šemi koja se naziva *striping*. Redundantan kapacitet diska se koristi za smeštanje podataka o parnosti što garantuje da se podaci mogu oporaviti u slučaju greške ili otkaza diska.
* **RAID 0** nije potpuni RAID zato što ne sadrži redundantnost. Korisnički i sistemski podaci su raspoređeni preko svih diskova u RAID-u u strip-ovima. Strip-ovi mogu da budu fizički blokovi, sektori ili neka druga jedinica. Mapiraju se pomoću Round Robin algoritma na fizičkim diskovima u RAID-u. U RAID-u sa *n* diskova prvih *n* strip-ova se skladište na prvom strip-u svakog diska.
* **RAID 1** se razlikuje od RAID-ova 2 do 6 u načinu u kome se dostiže redundantnost koja se postiže dupliciranjem podataka. Koriste se strip-ovi kao i kod RAID 0, ali u ovom slučaju svaki strip se mapira na dva fizička diska tako da za svaki disk u RAID-u postoji još jedan disk sa istim podacima. Može se implementirati i bez striping-a.
* **RAID 2**, kao i RAID 3, koristi tehniku za paralelni pristup u kome svi diskovi učestvuju u izvršenju svakog IO zahteva. Glave individualnih diskova su sinhronizovane tako da je svaka glava u istoj poziciji na svakom disku u bilo kom trenutku. Striping se koristi i ovde ali su strip-ovi veoma mali, obično veličine bajta ili reči. Za korigovanje grešaka na jednom bitu i detektovanje grešaka na dva bita koristi se Hamming kod.
* **RAID 3** je organizovan slično kao RAID 2 sa razlikom da RAID 3 zahteva samo jedan redundantni disk nebitno od veličine RAID-a. Umesto Hamming koda koriste se bitovi parnosti za skupove individualnih bitova u istoj poziciji za podatke na svim diskovima.
* **RAID 4**, kao i RAID 5 i 6, koriste tehniku nezavisnog pristupa tako da svaki disk radi nezavisno što omogućava paralelnu obradu IO zahteva. Koristi se striping, mada su strip-ovi relativno veliki. Računaju se strip-ovi parnosti na osnovu svih odgovarajućih strip-ova na svakom disku i skladište se u odgovarajućem strip-u na disku parnosti.
* **RAID 5** je organizovan slično kao RAID 4 sa razlikom da umesto što se svi bitovi parnosti nalaze na posebnom disku parnosti u ovom slučaju se bitovi parnosti raspoređuju na svim diskovima. Bitovi parnosti se obično raspoređuju Round Robin algoritmom.
* **RAID 6** sadrži dva izračunavanja parnosti čiji se rezultati skladište u odvojenim blokovima na različitim diskovima tako da ako je za skladištenje podataka potrebno N diskova, za RAID 6 će biti potrebno N+2 diskova. Prvo izračunavanje parnosti je XOR operacija kao kod RAID 4 i 5, a drugo je nezavistan algoritam provere podataka. Ovo omogućava oporavak podataka čak i kod otkaza dva diska.
* **Keš memorija diska** je bafer u glavnoj memoriji u kome se čuvaju kopije sektora sa diska. Kada stigne zahtev za određeni sektor prvo se proverava da li je taj sektor u keš memoriji.

**10 – Upravljanje datotekama**

* **Datoteka** je imenovana kolekcija povezanih podataka zapisana na sekundaron memoriji. **Datotečni sistem** je deo OS-a zadužen za upravljanje datotekama koji obezbeđuje interfejs za korisnike datoteka. Omogućuje korisnicima kreiranje datoteke kao kolekcije podataka koje imaju svjostva dugoročnog postojanja, deljivost između procesa i poseduju odgovarajuću internu strukturu.
* **Polje** predstavlja elementarni podatak koji sadrži vrednost, ime vrednosti i u nekim slučajevima dužinu vrednosti. **Slog** je kolekcija polja koja se tretiraju kao celina od strane aplikacionih programa. **Datoteka** je kolekcija povezanih slogova koja se može referencirati imenom i tretira se kao celina od strane korisnika i aplikacija. **Baza podataka** predstavlja kolekciju podataka u povezanim datotekama.
* **Gomila** je najprostiji tip organizacije datoteka. Podaci su organizovani u redosledu u kome su stizali. Uloga gomile je da sakuplja određenu količinu podataka i snimi je bez organizacije. Pristup podacima zahteva proveru svakog sloga sve dok se ne nadje traženi.
* **Sekvencijalna datoteka** je najčešći tip organizacije datoteka. Koristi se fiksni format slogova pri čemu se slogovi sastoje od istog broja polja fiksnih veličina u određenom redosledu. Slogovi sadrže polje koje predstavlja ključ koji jedinstveno identifikuje dati slog. Datoteka može biti sortirana po ključu (A-Z ili numerički).
* **Indeksna sekvencijalna datoteka** funkcioniše slično kao sekvencijalna s tim što se dodaje indeks datoteci za podršku random pristupu i dodaje se overflow datoteka. Indeks omogućava brzu pretragu u okolini traženog sloga, dok je overflow datoteka integrisana tako da se slog u overflow datoteci može naći preko pokazivača roditeljskog sloga.
* **Indeksna datoteka** je struktura u kojoj se slogovima pristupa samo preko njihovih indeksa. Koristi se višestruki indeks za svako polje po kome se vrši pretraživanje. Indeks može biti iscrpni (sadrži jedan ulaz za svaki slog u datoteci) ili parcijalni (ukazuje na grupu slogova u okviru kojih postoji traženi slog).
* **Direktna/heširana datoteka** se zasniva na karakteristikama diska da direktno pristupi bilo kom bloku sa poznatom adresom. Svaki slog ima ključ pa se direktan pristup realizuje heširanjem vrednosti ključa.
* **Direktorijum je** datoteka koja služi za organizovanje datoteka i kojasadrži informacije o datotekama. Operacije koje mogu da se primene na njemu su traženje, kreiranje datoteka, brisanje datoteka, listanje sadržaja i ažuriranje. Mogu biti organizovani u jednom nivou, dva nivoa ili hijerarhiski u vidu stabla (master direktorijum sa korisničkim direktorijumima ispid njega, svaki direktorijum može imati poddirektorijume i datoteke). Putevi između datoteka u direktorijuma mogu biti apsolutni (put u odnosu na koren – master direktorijum) ili relativni (put u odnosu na tekući direktorijum).
* **Kontinualna dodela** je metoda dodeljivanja datoteke u kojoj se svakom fajlu prilikom kreiranja dodeljuje kontinualni skup blokova. Tabeli alokacije je potrebna samo po jedna stavka za svaki fajl koja sadrži početnu adresu i dužinu fajla. Prednosti su mimalno vreme traženja i mogućnost čitanja celog fajla samo jednom komandom, ali može se javiti eksterna fragmentacija i fajlovi moraju biti fiksne veličine. Eksterna fragmentacija se može rešiti kompakcijom.
* **Dodela ulančavanjem** je pristup dodele blok po blok. Svaki blok sadrži pokazivač na sledeći blok datoteke. Tabeli alokacije (FAT – File Allocation Table) je potrebna samo jedna stavka za svaki fajl koja sadrži početni blok i dužinu fajla. Dodela blokova se vrši po potrebi tako što se bilo koj slobodan blok može dodati u ’lanac’ tako da ne dolazi do eksterne fragmentacije. Pristup individualnim blokovima fajla zahteva obilazak lančane liste dok se ne naiđe na traženi blok.
* **Dodela indeksiranjem** je slučaj u kome FAT sadrži poseban indeksni blok za svaki fajl koji sadrži po jednu stavku za svaki blok dodeljen fajlu. Indeksni blok se čuva u posbnom bloku a stavka u FAT-u pokazuje na taj blok. Alokacija može biti na osnovu blokova fiksne veličine (eliminiše eksternu fragmentaciju) ili na osnovu grupa promenljivih veličina (poboljšava lokalnost).
* **Bit tabela** (bit vektor) je metod koji koristi vektor sa jednim bitom za svaki blok na disku u kome 0 predstavlja slobodan blok a 1 zauzet. Relativno je lako naći jedan ili grupu susednih slobodnih blokova. Bit tabela zahteva prostor koji se računa kao *veličina diska u bajtovima/veličina bloka u bajtovima.* Sistemi koji koriste ovaj metod održavaju pomoćne strukture podataka koje dele bit vektore na podopsege za koje se pamti broj slobodnih blokova i maksimalni broj kontinualnih slobodnih blokova.
* **Ulančavanje slobodnog prostora** koristi ulančavanje slobodnih blokova ili grupa blokova. Pokazivač na prvi blok ove liste se čuva na specijalnoj lokaciji na disku i kešira u memoriji. Kod ove metode disk može postati fragmentisan i veliki broj grupa blokova će biti dužine od samo jednog bloka.
* **Indeksiranje** – slobodni prostor se tretira kao fajl i koristi tabelu indeksa. Postoji po jedna stavka u tabeli za svaku slobodnu grupu blokova.
* **Lista slobodnih blokova** – svakom bloku se dodeljuje broj sekvencijalno i lista brojeva svih slobodnih blokova se održava u posebnom delu diska. Ako je lista prevelika za čuvanje u glavnoj memoriji, i dalje se u glavnoj memoriji može čuvati mali deo liste tako što se tretira kao stek ili FIFO red.