Uvod

**Jezgro ili kernel** (kernel): deo operativnog sistema koji je uvek učitan u operativnu memoriju ili se u nju učitava pri uključivanju računara i tu ostaje stalno do isključenja, izvršava osnovne funkcije operativnog sistema i pruža usluge programima koji se na računaru izvršavaju.

**Sistemski poziv** je određena funkcionalnost koju operativni sistem pruža korisničkim programima. Korisnički programi mogu pozvati sistemski poziv, često korišćenjem mehanizma softverskih prekida, kako bi od operativnog sistema zatražili neke osnovne usluge kojima najčešće iz korisničkog režima i nemaju pristup. Primer sistemskog poziva bi bio fork() na *Unix-like* sistemima.*Sistemski poziv je metod kojim program koji se izvršava na nekom OS-u traži određenu uslugu od tog OS-a.*

**Sistemski program** je program koji obavlja osnovne operacije sa korisničkim sistemom, na primer brisanje fajlova (rm na *Unix-like* sistemima).   
*To su programi koji se izvršavaju kao i svi ostali, samo što se isporučuju kao sastavni deo operativnog sistema, jer obavljaju neke opšte radnje.*

**Korisnički interfejs** (user interface): deo za interakciju sa korisnikom (čovekom).

**Proces** (process) je jedno izvršavanje nekog programa na računaru, koje potencijalno teče uporedo sa drugim takvim izvršavanjima istog ili drugih programa.

**Računarski sistem** može biti:   
**monoprogramski**: izvršava samo jedan program, a ne bilo koji, proizvoljan program koji mu se zada; primer su ugrađeni (embedded) sistemi koji izvršavaju samo program koji je ugrađen u sistem, a ne bilo koji program koji zada korisnik (npr. jednostavni kućni uređaji - veš mašine, šporeti, usisivači itd.)   
**multiprogramski**: mogu da mu se zadaju različiti, proizvoljni programi pisani za taj sistem

**Operativni sistem** može biti:   
**monoprocesni**: omogućava samo jedno izvršavanje programa u datom trenutku, sledeći program može da se pokrene tek kad se prethodni završi  
**multiprocesni**: omogućava uporedno izvršavanje više procesa dok neki čekaju na U/I operacije itd.

Ovaj koncept uporednog izvršavanja procesa, pri čemu se procesor vremenski mulitpleksira između različitih procesa (u jednom intervalu izvršava se deo jednog procesa, pa onda nekog drugog itd.) naziva se **multiprogramiranje** (multiprogramming).

**Time sharing** je koncept gde kad OS da procesor nekom procesu, ne dozvoljava mu izvršavanje do samog kraja naleta, dok ne zatraži sistemski poziv ili sam ne pusti procesor, već mu da odredjeno vreme (**time slice**) nakon kog mu oduzima procesor (**preemption**), menja kontekst I daje procesor drugom procesu.

Treba se zaštititi od situacije u kojoj neki proces nikada ne pozove sistemski poziv (npr. zbog greške ili loše namere) ili ga  
ne poziva jako dugo - treba mu preoteti procesor (**preemption**).

**Mehanizam prekida** omogućuje preotimanje procesora - procesor dobija signal da prekine trenutni proces i pređe na kod kernela. Preotimanjem se postiže multiprogramiranje sa boljom raspodelom vremena i vremenom odziva.

OS može **preusmeriti standardni ulaz ili izlaz** pri pokretanju procesa iz komandne linije:

u tekstualni fajl ili iz tekstualnog fajla - redirekcija (redirection), > i <  
ls -al > listing.txt (Izlaz komande/procesa ls -al preusmerava se u fajl listing.txt)

izlaz jednog procesa prosleđuje se kao ulaz drugog procesa - cevovod (pipe), |

cat listing.txt | less (Izlaz komande/procesa cat usmerava se na ulaz komande/procesa less)

**Multiprocesorski sistem** (multiprocessor): računarski sistem sa više procesora koji imaju zajedničku (deljenu) operativnu memoriju; procesori mogu da pristupaju deljenoj memoriji (čitaju iz nje ili u nju upisuju), npr. preko zajedničke magistrale na koju su svi povezani.

**Distribuiran sistem** (distributed system): sistem sa više procesora koji nemaju zajedničku operativnu memoriju (već svaki procesor ima svoju), a koji su povezani komunikacionom mrežom preko koje mogu razmenjivati poruke. LAN, WAN, Internet.

**Multiprocesorski sistem** može biti, na hardverskom nivou:

**simetričan**: svi procesori su opšte namene, jednaki, imaju isto vreme pristupa operativnoj memoriji

**asimetričan**: neki procesori su specijalizovani za posebne namene (npr. floating point unit - FPU, graphical processing unit - GPU), ili imaju različito vreme prisupa operativnoj memoriji (non-uniform memory access, NUMA)

**OS** za multiprocesorski sistem može biti:

**simetričan**: svi procesori su ravnopravni, u smislu da svi mogu izvršavati i kod procesa i kod kernela

**asimetričan**: jedan procesor je “gazda” (master) i on izvršava raspoređivanje procesa na druge procesore, kao i kernel kod za ostale sistemske usluge; ostali procesori su “robovi” (slave) i samo izvršavaju kod korisničkih procesa u skladu sa onim što im master dodeli.

Kompajler I linker

**Prevodilac (compiler)** je program koji prevodi tekstualni zapis izvornog programa na višem programskom jeziku u binarni, mašinski zapis.

Kompajler radi: .cpp->.obj/.o odnosno iz izvornog koda pravi **object file**. Bilo koja greška u prevodjenju => neće se napraviti object file (ipak prevodilac nastavlja prevodjenje I prijavljuje greške). Svako ime u programu mora biti **deklarisano** pre upotrebe.

U objektnom fajlu se nalazi: binarni mašinski kod I alociran prostor za statičke podatke.

Kad naidje na deklaraciju, kompajler to ime stavlja u **tabelu simbola**, strukturu koju izgradjuje tokom prevodjenja I koja sadrži ime I atribute vezane za jezičku kategoriju, tip I ostala svojstva definisana jezikom.

Prilikom prolaska kompajler za jedan identifikator/ime proverava da li je deklarisan i dostupan, po pravilima jezika, i da li je upotrebljen u skladu sa pravilima jezika. U suprotnom javlja **grešku**.

U zaglavlju objektnog fajla prevodilac pravi **tabelu izvezenih simbola**, **external linking**, i ti simboli se mogu koristiti u drugim fajlovima dok, imena sa internim vezivanjem, **internal linking**, se ne mogu koristiti u drugim fajlovima.

U tabeli izvezenih simbola se nalazi ime i realtivna adresa u koju se to ime preslikava.

Int n; je i definicija ako je globalna promenljiva, I alociraće prostor za nju iako nije inicijalizovana.

Extern int n; (kao ni extern ili bez extern, void f();) nije definicija, neće se alocirati prostor.

Na mestima uvezenih simbola stavlja se nedefinisana vrednost.

**BSS segment** - segment sa statičkim podacima inicijaliziovanim nulama, tzv. bss segment dovoljno je da prevodilac ostavi informaciju u exe fajlu o poziciji i veličini tog segmenta; prilikom kreiranja memorijskog konteksta procesa, OS će u prostor u memoriji alociran za ovaj segment samo upisati sve nule; zbog ovoga neki jezici, poput C/C++ definišu posebnu semantiku za statičke objekte sa podrazumevanom inicijalizacijom na 0 (tzv. *zero initialization*), a takve statičke objekte grupišu u isti segment, odvojen od statičkih objekata koji su statički inicijalizovani drugim vrednostima.

**Linker** je program koji više objektnih fajlova povezuje u jedan izvršiv fajl.

Linker radi: [.obj, .obj,…,.obj] ->.exe.

**Prvi prolaz linkera**: u prvom prolazu analizira ulazne fajlove, veličinu njihovog binarnog sadržaja (prevoda), i pravi mapu exe fajla; osim toga, sakuplja infomacije iz tabela simbola obj fajlova i izgrađuje svoju tabelu simbola; u tu tabelu simbola unosi izvezene simbole iz obj fajlova, za koje odmah može da izračuna adresu u odnosu na ceo exe fajl.

**Drugi prolaz linkera**: u drugom prolazu generiše binarni kod, i ujedno razrešava nerazrešena adresna polja mašinskih instrukcija na osnovu informacija o adresama u koje se preslikavaju simboli iz njegove tabele simbola.

Primer: Recimo da imamo dva objektna fajla, a.obj i b.obj. a.obj izvozi simbol f i uvozi simbol c. Linker će iz prvog objektnog fajla dodati u tabelu simbol f. b.obj uvozi simbol f i izvozi simbol c, pa će linker u svoju tabelu dodati simbol c. U drugom prolazu će linker razrešiti c polje u a.obj i polje f u b.obj.

Fajl .**lib** je nastao povezivanjem više .obj fajlova, dok .obj prevodjenjem jednog izvornog fajla.  
**Library** služi da bi se kod za jednu biblioteku spakovao u jedan .lib fajl. Biblioteka svakako izvozi simbole koje definiše  
(upravo one koji predstavljaju njene usluge, ono za šta se ona i koristi), ali može i da ih uvozi, ukoliko zavisi od neke druge biblioteke (koristi simbole iz druge biblioteke).

Naravno, posao pravljenja **lib** fajla se **razlikuje** od posla pravljenja **exe** fajla iz nekoliko razloga:

**exe** može imati **drugačiji** **format** od lib (i obj) fajla

**exe sadrži** i informacije koje lib (i obj) fajlovi ne sadrže, a koje su potrebne operativnom sistemu za pokretanje programa; na primer, **barem početnu adresu prve instrukcije od koje operativni sistem treba da počne izvršavanje programa**

kada pravi exe, linker mora da završi posao bez nerazrešenih simbola; **lib može da ima simbole** koje uvozi i koji ostaju nerazrešeni.

Vrste grešaka:

Simbol nedefinisan – ako se uvozi simbol koji nijedan fajl nije izvezao.

Simbol višestruko definisan – kad neki fajl izvozi ime koje već postoji u tabeli simbola jer ga neki drugi fajl već izvozi. To je **name clash**, sukob imena.

Organizacija I alokacija memorije

U monorpocesnom sistemu u memoriji je samo jedan proces I on može da je koristi celu.  
Stek se često alocira da raste naniže, jer ako bi bio odmah iznad podataka, dinamičke strukture bi ga pregazile.

**Particionisanje** – raspoloživ prostor se deli na N disjunktnih, jednakih delova. Svaku particiju može zauzimati samo jedan proces ili može biti slobodna.

Problemi:

Ograničen stepen multiprogramiranja.

**Stepen multiprogramiranja** – broj aktivnih procesa koji su u memoriji i mogu se izvršavati.

Veličina particija mora biti onolika koliko je velik virtuelni/logički adresni prostor, a fizička memorija značajno veća.

Interna fragmentacija.

**Interna fragmentacija** je pojava da proces zauzima više memorije nego što mu je potrebno. Zove se interna jer je slobodna memorija zarobljena unutar logičkog bloka u memoriji.

**Kontinualna alokacija** – proces zauzima memorije tačno koliko mu je potrebno. Fizička adresa se dobija sabiranjem logičke i bazne adrese.

Kod vodjenja evidencije o slobodnim blokovima se javlja **eksterna fragmentacija**, to je pojava da ostane mali deo slobodne memorije koji možda ni ne može evidentirati.  
Moguće rešenje je da se da procesu taj delić ili da se memorija alocira na fiksne blokove neke veličine.  
Ova rešenja opet dovode do interne fragmentacije!

**First fit** – alocira se prvi blok koji je dovoljno velik.  
**Best fit** – alocira se najmanji od onih koji odgovaraju.  
**Worst fit** – alocira se najveći blok.

Ako nakon dužeg rada sistema slobodni fragmenti budu previše mali, može se vršiti **kompakcija**, premeštanje procesa tako da se slažu jedan iza drugog.

**Segmentna organizacija** – razlog uvodjenja je to što kod kontinualne alokacije, izmedju ostalog, postoji problem gde kada bi proces alocirao stek na vrhu svog adresnog prostora, izmedju podataka i steka može potencijalno biti mnogo neiskorišćenog prostora, interna fragmentacija. Takodje, ako više procesa izvršava isti kod, ima više kopija koda. Sada virtuelna adresa ima **VA() = SEGMENT()OFFSET()** => a fizicka adresa se dobija korišćenjem segmenta iz virtuelne kao indeksa u nizu, odnosno u, **Segment Map Table (SMT)**. Jedan ulaz u SMT je deskriptor i on sadrži   
**SMT\_DESCR() = BASEADDR()LIMIT()**. Ako je OFFSET > LIMIT desilo se prekoračenje. Fizička adresa se dobija kao **PA() = BASEADDR()+OFFSET()**.

void\* mmap (void\* base\_addr, size\_t size); - alocira size memorije od base\_addr, ako je base\_addr = NULL, OS će sam pronaći slobodan segment i vratiće virtuelnu adresu početka.

int munmap (void\* base\_addr, size\_t size); - dealokacija.

U PMT mogu se nalaziti biti za pravo pristupa stranici **RWX**, pa se u mmap još dodaje I int prot argument kojim se najavljuje pravo pristupa toj stranici.

Problem **eksterne fragmentacije**. Kontinualna alokacija je specijalan slučaj segmentne kad proces ima samo jedan segment. Kod segmentne se kontinualno alociraju segmenti.

**Segmentno-stranična** – **VA() = SEGMENT()PAGE()OFFSET() => PA() = FRAME()OFFSET()**. Frame se dobija tako što se prvo preko segmenta u SMT nadje ulaz u kom je pokazivač na **Page Map Table (PMT)**, pa na isti način kao pre, korišćenjem page polja kao indeksa u ovom nizu, pronalazimo frame koji konkateniramo sa offsetom. Ovakva organizacija je ista kao stranična samo što se još jednom grupišu stranice u logičke segmente, za šta nema potrebe.

**Eksterne fragmentacije nema** jer se alociraju blokovi iste veličine dok **interne ima**.

**Stranična organizacija** - **VA() = PAGE()OFFSET() => PA() = FRAME()OFFSET()**. Preskače se korak traženja pokazivača na PMT već direktno se nalazi u nizu frame.

Postoji **interna fragmentacija**. Ona se može rešiti, smanjivanjem veličine stranice.

**Manje stranice** uzrokuju veće tabele preslikavanja, manja stranica znači da ih ima više pa mora imati I više ulaza u PMT da pokrije svaki frame. Takodje, smanjena je efikasnost zbog režijskih troškova.  
**Veće stranice** uzrokuju manje tabele preslikavanja, a U/I operacije su više efikasne za veće blokove prenosa.

Budući da procesi koriste mali deo svog adresnog prostora može se raditi sledeće.

Da PMT pokriva prvih n ulaza a da se ostali ne alociraju. Ovo može raditi samo ako proces alocira prostor u nižem delu adresnog prostora. Ako alocira podatke sa velikom razdaljinom, moraće da se alociraju svi ulazi u PMT izmedju.

Upotreba heš tabele. Nije dobro zbog efikasnosti.

**Straničenje u više nivoa**. PMT se organizuje u više nivoa i alociraju se samo one strukture koje se koriste.

Problem je što za svaku stranicu mora da se pristupa više puta memoriji prilikom SVAKOG preslikavanja.

Rešenje je **Translation Lookaside Buffer (TLB)**. On sadrži deskriptore koji su nedavno korišćeni. Kad se slika VA u PA, MMU traži u TLB deskriptor, ako ga nema, traži u memoriji i upisuje u TLB tako da sledeći put bude brži pristup.

*Zadatak: U nekom sistemu sa straničnom organizacijom memorije i straničenjem u dva nivoa vreme pristupa dinamičkoj RAM memoriji je 100 ns, vreme pristupa TLB-u je 10 ns, a TLB ima učestanost pogotka od 90%. Koliko je efektivno vreme pristupa memoriji? Izvesti račun.*

9/10\*(10+100)ns + 1/10\*(10+100+100+100)ns = 130ns

Prvi sabirak kaže da u 9 od 10 slučajeva će biti pogodak odnosno, pristupaće se jednom TLB i jednom operativnoj memoriji kako bi se pristupilo stranici. U drugom slučaju, kada se desi 1 od 10 da se ne pogodi, prvo je pokušano da se pristupi TLB i desio se neuspeh, pa se pristupa PMT1, PMT2 i konačno operativnoj memoriji.

**Zaštita**:

Privilegovani i korisnički režim gde neke instrukcije nisu dozvoljene korisniku, dok kernelu jesu, kao i registri kao što je PMTP.

Da proces ne bi adresirao kod kernela, može dva pristupa:

Da se kernel kod ne mapira u virtuelne adresne prostore za procese pa procesu nije moguće da pristupi tom delu memorije kad se nijedna adresa ne slika tu. Kad kernel hoće da pristupi svom delu memorije, on menja PMTP na tabelu koja preslikava jedan na jedan, virtuelnu u identičnu fizičku, pa može pristupiti svakom delu memorije.   
Problem je baš to preslikavanje svakog parametra sistemskog poziva.

Kernel u svaki virtuelni adresni prostor procesa, preslikava svoj kod na neki odredjeni deo memorije, gore ili dole, a pristup tim stranicama zabranjuje iz korisničkog režima.

Deljenje memorije



**Dinamičko učitavanje** – procesi ne koriste neke strukture podataka ili potprograme, pa oni se ne moraju alocirati. Tek kada su zaista potrebni se traži njihova alokacija. Primer su moduli.   
Odgovornost programa je da presretne u nekom getteru/setteru pristup objektu potrprogramu ili strukturi podataka koja još nije učitana, da je učita, pa tek onda izvrši zadati poziv.

**Preklopi** – moduli se koriste naizmenično, u memoriji se jedan preko drugog prepisuju. Odgovornost je takodje na procesu da vodi evidenciju o učitanim modulima. OS treba da obezbedi učitavanje binarnog sadržaja exe fajla i alokaciju potrebne memorije, a za šta će se ta memorija koristiti, to zna proces.

**Logičko deljenje memorije** – sve dok dva procesa dele jedan deo memorije, deskriptori u PMT ukazuju na isti kod u fizičkoj memoriji. U deskriptorima logičkih segmenata piše da je upis i čitanje dozvoljeno.  
Kada jedan proces pokuša da upiše nešto u taj deo memorije, dešava se stranična greška jer u PMT piše da je dozvoljeno samo čitanje. Tada se proverava da li je zapravo dozvoljen upis, ako jeste, kopira se deo memorije i preusmerava se pokazivač a u PMT svih procesa koji su delili, Write bit se postavlja na 1.

**Copy on write** je opisana tehnika, gde kad se pokuša upis u lokaciju koja je deljena, prvo se kopira, preusmeri pokazivač iz PMT pa onda promeni memorija.

*Zadatak: Neki sistem primenjuje tehniku kopiranja pri upisu (copy on write). Jedan proces roditelj zauzima sledeću memoriju: 12KB za programski kod, 24KB za stek, 36KB za podatke koji se samo čitaju i 40KB za podatke koji se menjaju. Ako ovaj proces pokrene proces-dete sistemskim pozivom fork, koliko najviše operativne memorije ova dva procesa ukupno zauzimaju? Objasniti.*

Najviše mogu da zauzimaju kad proces dete promeni podatke. Tada se ponovo alocira stek i prostor za podatke koji se menjaju. 12KB + 2X24KB + 36KB + 2X40KB

void\* mmap (…); može služiti za ovu namenu.

**Deljene biblioteke** – kada bi se biblioteke statički vezivale u exe fajlovima svih programa koji ih koriste, onda bi postojale višestruke kopije istog sadržaja. Sadržaj se deli izmedju procesa isto kao kad se deli ceo program, za procese nad istim programom samo što:

Sad se deli samo kod jedne biblioteke

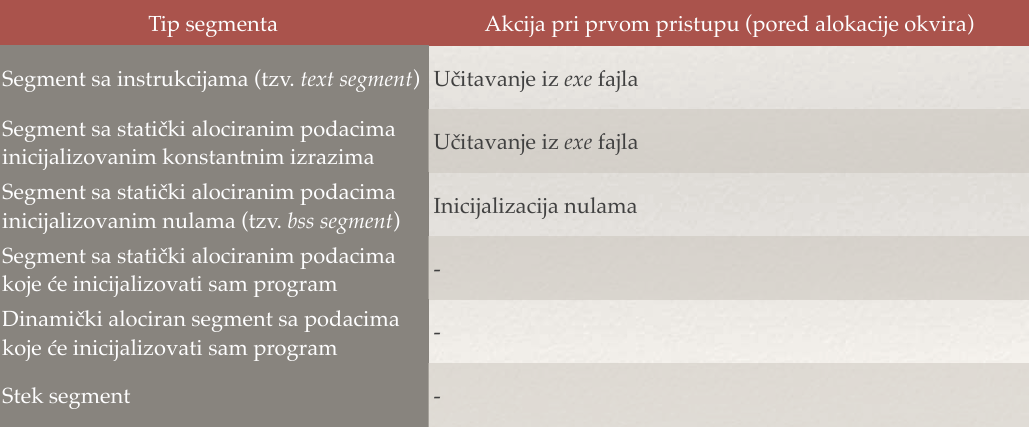
Kod se dinamički uvezuje sa ostatkom programa koji koristi usluge biblioteke, u vreme izvršavanja procesa

Odatle nastaju **DLL – Dynamic Linking Libraries**. Proces koji koristi usluge neke biblioteke može da pozove neki potprogram na standardan način. Taj poziv ustvari nije poziv potprograma, jer je taj poziv **stub**, patrljak, koji je statički uvezan I on traži od OS da mapira DLL. Ako DLL nije učitan u memoriju, OS će ga učitati I mapirati u virtuelni adresni prostor procesa I vraća pokazivač na tabelu simbola pozivajućem procesu. Patrljak sad koristi tu tabelu simbola kako bi pozvao ciljni potprogram DLL-a. Tabelu simbola kao linker, formira OS prilikom učitavanja u DLL u memoriju.

**Zamena procesa** – jedan proces se izbacuje, snima na disk, a drugi ubacuje. Pošto je ovo vremenski zahtevna operacija, OS ovo rade samo kad baš ne mogu da stanu svi procesi ili se može koristiti kao tehnika za sprečavanje **batrganja, thrashing.**

Sistem može da uđe u režim u kom se stranične greške dešavaju izuzetno često, procesi često i dugo bivaju suspendovani, pa procesor nema šta da radi, **iskorišćenje mu je izuzetno nisko**, dok **ulazno-izlazni podsistem za operacije sa diskom postaje preopterećen**, ne može da postiže da opsluži sve zahteve, pa zahtevi dugo čekaju - disk podsistem postaje usko grlo celog računara. To je **thrashing**.

**Učitavanje stranica na zahtev (demand paging)** – OS formira memorijski kontekst, PMT I deskriptore logičkih segmenata. U PMT ulaze upiše NULL. Svaki put kad proces pokuša da pristupi nekoj stranici koja nije učitana, desiće se page fault. OS proverava da li virtuelna adresa koja je generisala page fault pripada regularno alociranom segmentu. Ako jeste, OS učitava stranicu, upisuje u PMT pokazivač na nju I ponovo izvršava instrukciju.

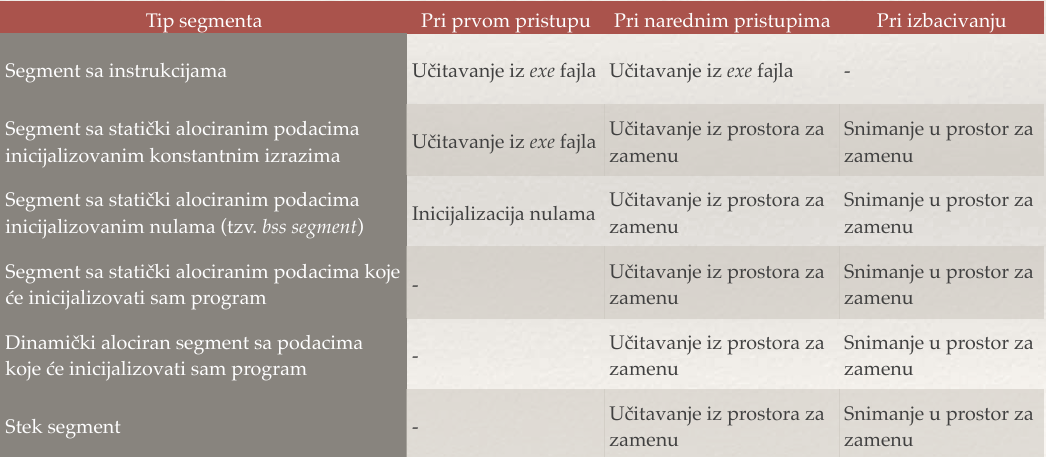


**Zamena stranica** – pri straničnoj grešci, kad se traži da se stranica učita, ako mesta nema, OS mora naći žrtvu za izbacivanje. Tu stranicu čuva na disku na **swap space ili raw partition**, a drugu stranicu upisuje. Proces ovako može da vidi i koristi daleko veći adresni prostor od nego što računar zaista poseduje.

Prilikom izbacivanja stranice u njen deskriptor upisuje NULL, tako da će je posle učitati opet.

U deskriptoru stranice može postojati **dirty bit**. Ovo je hardverska podrška od MMU. On setuje ovaj bit na 1 kad god se upisuje nešto u ovu stranicu. Prednost je da ako je bit 0 stranice koja se izbacuje, on je ne mora snimiti na disk, jer sadržaj nije menjan.

**LRU – least recently used** je algoritam koji će izbaciti stranicu koja je najdavnije korišćena.



Procesi I niti

**Proces** (process) je *jedno* izvršavanje nekog programa sa *jednim* (virtuelnim) adresnim prostorom (address space).  
**Interaktivni procesi** – mogu pomoću GUI da interaguju sa korisnikom.  
**Pozadinski procesi** – nemaju interakcije sa korisnikom već uporedo sa njim izvršavaju neke radnje.

**Nit** (thread): tok kontrole koji teče uporedo sa drugim tokovima kontrole, ali koji *deli* virtuelni adresni prostor sa nekim drugim tokom ili tokovima kontrole (nitima).

**Tok kontrole** – redosled sekvencijalnog izvršavanja instrukcija, jedna za drugom, odma iza jedne se nalazi druga u memoriji, osim ako se ne radi neke od instrukcija skoka, uključujući I skok I povratak iz potprograma.  
**Stanje** – predstavlja stanje regstara procesora I memorije koje adresira proces. Stanje koje zatekne jedna instrukcija, prethodna je to ostavila za sobom.

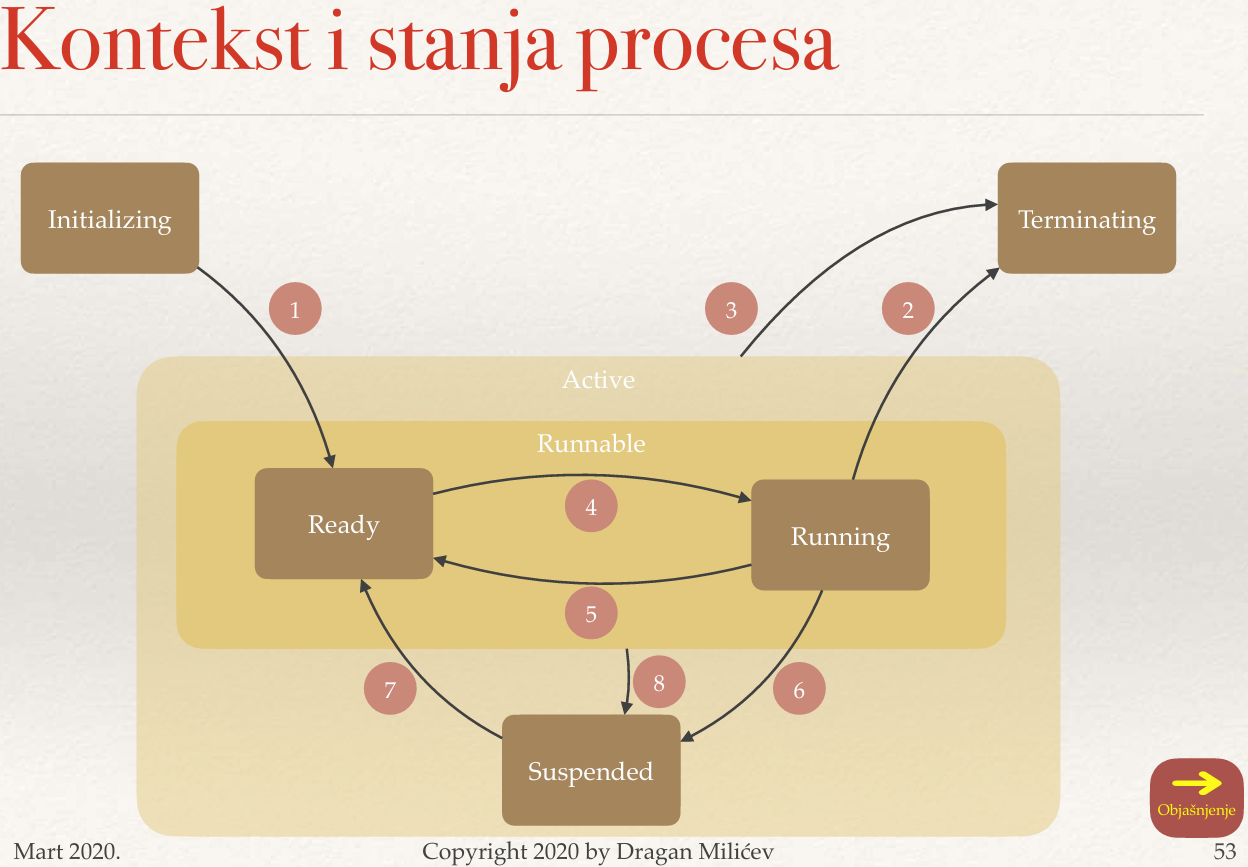
Kreiranje procesa pomoću **pid\_t pid();**u kontekstu deteta vraća 0, u kontekstu oca >0, greška <0.  
Suspendovanje tekućeg procesa sa **int wait(pid\_t pid, unsigned timeout); ili int wait(int\* status);** pid je na koje se dete čeka, timeout milisekundi. Ako je sa stausom, onda je to pokazivač na povratnu vrednost procesa deteta. Ako je pid NULL, onda se čeka na bilo koje dete. Ako hoćemo da čekamo na svu decu, mora se u for petlji pozivati wait.

**Int execlp(const char\*);** odbacuje trenutni kontekst I prelazi na exe fajl dat argumentom.

**Int kill(pid\_t pid, int signal);** šalje signal pid procesu. SIGKILL – 9, nasilno se gasi proces. SIGTERM – 15, blagonaklono gasi proces, uputi zahtev da se ugasi sam.

**kill -KILL 1234 ili kill -9 1234** prvi argument je signal drugi pid.

Sistemskim pozivom kill pozivajući proces drugom procesu šalje signal. Signal je prosta informacija, identifikacija nekakve proste poruke, obično celobrojna vrednost. Operativni sistem prenosi tu infomaciju odredišnom procesu, a reakcija na taj signal je poziv potprograma koji se izvršava u kontekstu i adresnom prostoru odredišnog procesa. Za svaki signal operativni sistem definiše podrazumevanu reakciju, uglavnom je to samo gašenje procesa sistemskim pozivom exit. Međutim, odredišni proces za neke signale može reakciju da preusmeri na svoje potprograme odgovrajućim sistemskim pozivom.



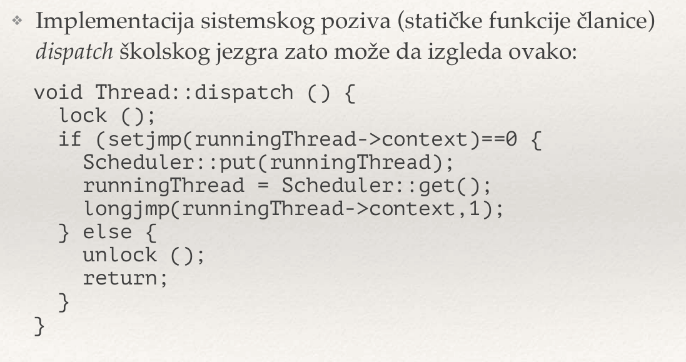
**Initializing** – kreiranje procesa. **Terminating** – oslobadjanje resursa procesa i brisanje procesa. **Running** – proces koji se trenutno izvršava na procesoru. **Ready** – imaju sve što je potrebno da mogu da se izvršavaju ali čekaju na red. **Suspended** – čeka na ispunjenje nekog uslova, ulazno-izlazna operacija, logički uslov, page fault obrada, swapping.

Promena konteksta:

**Sinhrona** – *voljna* promena, kad proces pozove sistemski poziv, blokirajući poziv. *Nevoljno*, instrukcija izazvala izuzetak.

**Asinhrona** – nepredvidiva, uglavnom spoljni hardverski prekidi.

Prilikom svakog prekida ili sistemskog poziva OS radi sledeće – čuva kontekst trenutnog procesa, izvršava potrebnu odbranu, bira nov proces za izvršavanje, restaurira njegov kontekst.

dispatch vrši jednostavnu promenu konteksta.   
lockgasi prekide i pomoću hardverske podrške instrukcijama test\_and\_set ili swap vrši zaključavanje sekcije.   
setjmp čuva kontekst niti koja je data I vraća nulu.  
longjmp restaurira dati kontekst I u setjmp postavlja return vrednost drugog argumenta.

**Algoritmi rasporedjivanja**:

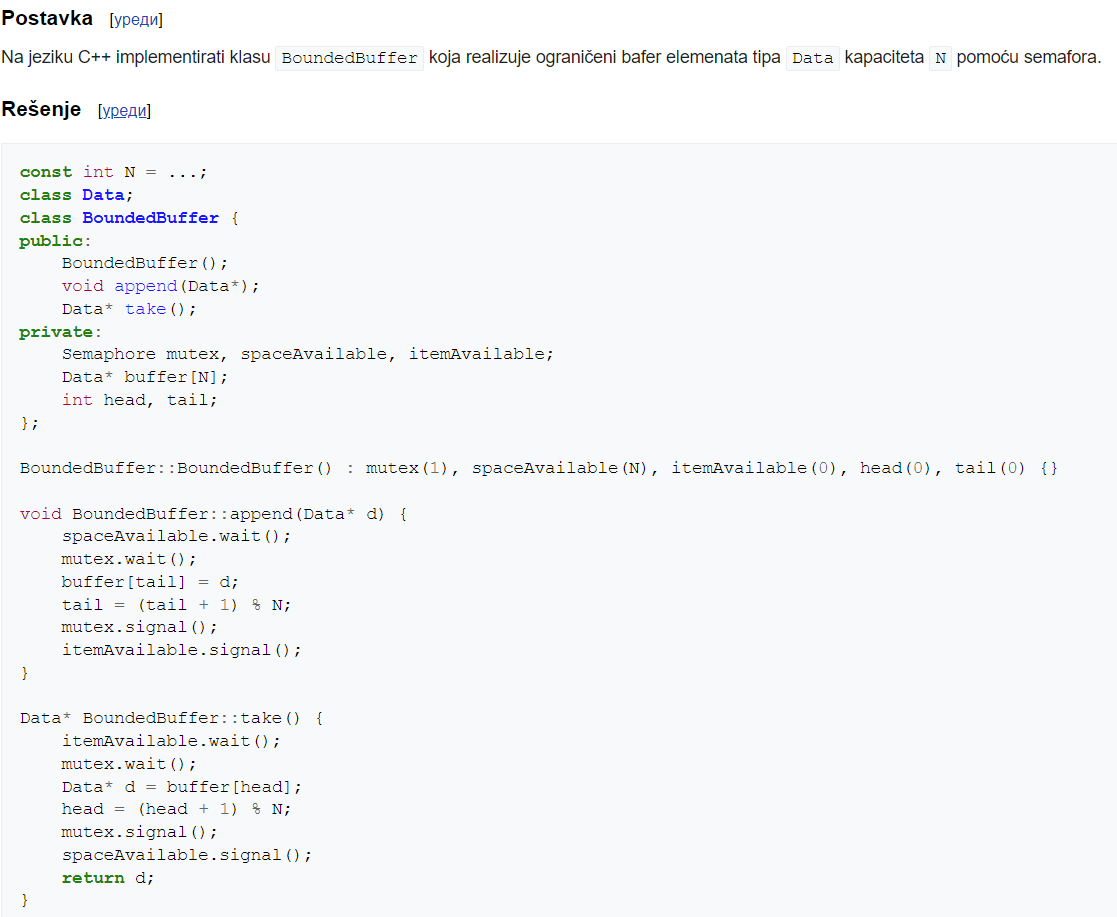
**FIFO/FCFS** – realizovan kao neprioritetni red.

Problemi: vreme čekanja i ukupnog čekanja na procesor može biti nerazumno dugačko – kraći procesi moraju čekati na procese koji se dugo izvršavaju. Bolje kada bi kraći procesi imali prioritet (kao na kasi).   
**Convoy efekat** – iza dugotrajnog procesa je kratkotrajni u redu čekanja. Dugotrajni se izvršava i traži operaciju sa diskom. Kratkotrajni dodje na red, izvršava se i recimo i on traži operaciju sa diskom, ali opet mora da čeka da dugotrajni završi svoju operaciju sa diskom, pa opet mora da udje u red spremnih pa se situacija ponavlja.   
Kao šleper, dugotrajni proces vuče i usporava saobraćaj.

**SJF (shortest job first)** – prioritet imaju procesi koji se kraće izvršavaju.

**Round robin** – isto kao FIFO samo sa vremenskim odsečkom.

Sinhronizacija procesa



Podrazumevano se procesi izvršavaju uporedo (**concurrently**), što znači da se sekvence njihovih instrukcija izvršavaju proizvoljno prepleteno ili čak fizički paralelno.

Jedan tip sinhronizacije predstavlja **uslovna sinhronizacija** (conditional synchronization): neki proces ne sme da nastavi izvršavanje iza neke tačke, tj. ne sme da izvršava neke akcije ukoliko neki drugi proces nije uradio nešto, ili ukoliko nije ispunjen neki uslov, ili ukoliko neki proces ili podatak nije u nekom potrebnom stanju i slično.  
Primer je join koji roditelj poziva za dete, čeka da se dete završi.  
Primer je i **uposleno čekanje** kod ograničenog bafera.

**Kritične sekcije** su sekcije u kojima više niti pristupa nekom deljenom resursu i menjaju ga. Mora se obezbediti **medjusobno isključenje** ovih sekcija tako da samo jedna nit može izvršavati taj kod dok druge cekaju.

Pokušaji rešavanja ovog problema:

*Obe niti imaju svoj flag* koji postavljaju pre ulaska u kritičnu sekciju a uklanjaju po izlasku. Nakon postavljanja flega uposleno se čeka da fleg druge niti bude uklonjen.

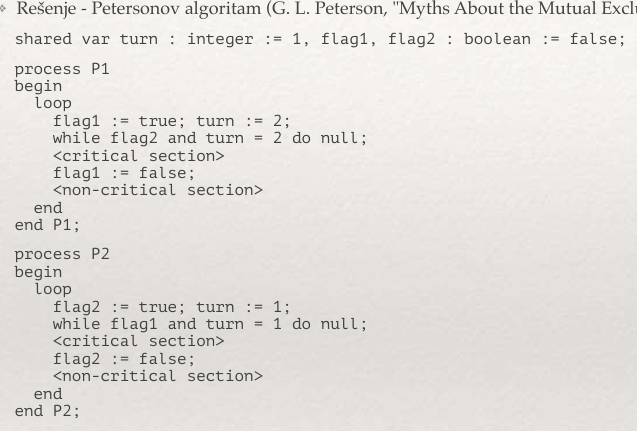
*Problem*: ako obe niti postave svoj fleg i obe uposleno čekaju.   
**Livelock** – procesi zaglavljeni u petljama jer se medjusobno čekaju ili u krug.

Zamene se mesta gde se prvo ispituje fleg suprotne niti a onda postavlja svoj.

*Problem*: nije obezbedjeno medjusobno isključenje.  
**Race condition** – proces se zatrčao, pretekao nekog drugog, krenuo da radi nešto što nije trebalo jer fali sinhronizacija.

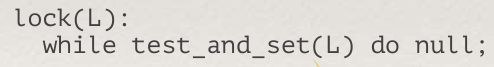
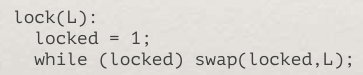
Uvesti *int turn* gde kad je 1 onda prednost ima 1. nit a kad je 2 onda ima 2. nit.

*Problem*: stroga naizmeničnost ulaska u kritičnu sekciju. Ukoliko neki proces ne želi da udje u kritičnu sekciju, ne treba da zadržava druge.

Ako proces želi da udje u kritičnu sekciju, postaviće svoj flag na 1 i čekaće na svoj turn.

*Problem*: procesi moraju da znaju jedan za drugog.  
Uposleno čekanje.

**Spin lock** – mehanizam koji se primenjuje kod LOCK funkcije. Mehanizam vrtenja radi zaključavanja. Uposleno čekanje.



Semafori

Mnogi sistemi podržavaju i posebne, **binarne semafore (binary semaphore)** čija je semantika u osnovi jednostavna:

imaju samo dve vrednosti, 0 i 1

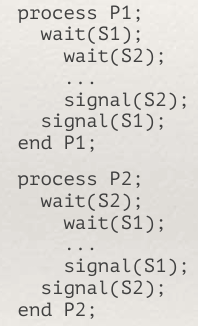
wait: ako je vrednost semafora 1, postavlja se na 0, a proces nastavlja; u suprotnom, proces čeka

dok neki drugi proces ne izvrši signal

signal: ako postoje procesi koji čekaju, jedan se deblokira; u suprotnom, vrednost se postavlja na 1

**Mutex**: binarni semafor namenjen samo za međusobno isključenje kritičnih sekcija; poseduje ograničenje da **samo proces koji je zatvorio semafor operacijom tipa wait može da ga otvori operacijom signal**, u suprotnom se ova operacija smatra greškom; može da poseduje i neku dodatnu podršku za raspoređivanje po prioritetima u RT sistemima-

Događaj (**event**) ili signal: služi za signalizaciju događaja, koji mogu doći i od hardvera, tj. od hardverskih uređaja koji izvršavaju uporedne radnje sa procesorom, pa se operacija signal može vezati i kao reakcija na spoljašnji prekid od hardvera (koji predstavlja apstraktan uporedni proces); mogu nametati i određena ograničenja, npr. to da samo proces koji je kreirao događaj i njegov je “vlasnik” može izvršiti operaciju wait i čekati na događaj, dok ga drugi procesi mogu signalizirati.

**Deadlock** (mrtva ili kružna blokada) – javlja se kad su dva procesa blokirani na semaforima a oba čekaju na signal od drugog procesa.   
To se dešava u ovom primeru kad P1 prodje wait na S1 I P2 prodje wait na S2 ali pošto su sada oba semafora na vrednosti 0, sledeći wait će blokirati te niti. Tako da se obe niti blokiraju na semaforima.

*Razlika u odnosu na živu blokadu je što se procesi sad ne izvršavaju nego su blokirani.*

Optimistički pristup

Jedan od pristupa može biti **optimistički pristup**, gde se kritična sekcija ne zaključava već se prvo pravi kopija deljenog objekta. Pripremi se nova vrednost za objekat. Atomično se uradi: ako je vrednost ostala nepromenjena, onda upiši, u suprotnom, ponovi postupak. **Compare and set** je instrukcija koja omogućuje ovo.

Medjuprocesna komunikacija

Komunikacija se može vršiti preko kernela, gde je on posrednik. Sa send se šalje podatak kao niz char, dužine len, na priključnicu socket. Kernel po pravilu kopira podatak u svoje bafere I kad prijemnik zatraži podatke sa recv, kernel kopira podatke na mesto koje je proces tražio, u bafer.

Procesi mogu razmenjivati podatke preko deljenog fajla, zajednička baza podataka.

Ulazno-izlazni podsistem

*Prema smeru toka podataka*:

**ulazni**: smer toka podataka je samo iz uređaja ka računaru (operativnoj memoriji); na primer: tastatura, miš, mikrofon, CD-ROM itd.

**izlazni**: smer toka samo iz računara na uređaj; npr. ekran, zvučnik, štampač itd.

**ulazno-izlazni**: oba smera; osnovni primer: uređaji za skladištenje podataka koji omogućavaju i čitanje i upis podataka.

*Prema jedinici prenosa:*

**znakovno orijentisani (character-oriented**): vrše prenos malih jedinica podataka, tipično pojedinačnih znakova (ili bajtova); npr. linijski štampač, tastatura i znakovni ekran.

**blokovski orijentisani (block-oriented)**: vrše prenos većih blokova podataka, fiksne ili promenljive dužine; npr. diskovi, mrežni adapteri itd.

*Prema načinu pristupa podacima:*

**sekvencijalni**: pristup podacima je određen prirodnim, fizičkim, prostornim ili vremenskim redosledom kojim ti podaci stižu ili se šalju i ne može im se pristupiti proizvoljnim redosledom; npr. tastatura, zvučnik, mrežni adapter, ali i magnetna traka (podaci se čitaju ili upisuju po redu prolaska trake) itd.

**sa direktnim pristupom**: podacima se može pristupati (na čitanje ili upis, u zavisnosti od vrste uređaja) u prozvoljnom redosledu, nezavisno od toga kako su oni prostorno smešteni ili vremenski pristigli; npr. diskovi, rasterski ekran

*Prema deljivosti, tj. tome da li uporedni procesi mogu slati parcijalne operacije uporedo ili ne:*

**nedeljivi, posvećeni (dedicated)**: jedan proces mora obaviti celu operaciju sa uređajem pre nego što drugi proces obavi svoju veliku operaciju; tipičan primer je štampač: nema nikakvog smisla da jedan proces pošalje nekoliko znakova, a onda neki drugi uradi isto

**deljivi (shared)**: može ih uporedo ili vremenski multipleksirano koristiti više procesa; npr. tastatura, ekran, miš, disk itd.

*Prema vremenskim karakteristikama prenosa podaka:*

**sinhroni (synhronous)**: prenos vrše u pravilnim ili predvidivim vremenskim trenucima ili razmacima

**asinhroni (asynhronous)**: prenos vrše u nepravilnim ili nepredvidivim vremenskim trenucima ili razmacima

*Jedna važna karakteristika sistemskih poziva jeste to da li se kontrola vraća pozivaocu, odnosno da li se pozivajući tok kontrole nastavlja odmah, ili tek kada se obavi cela U/I operacija:*

**sinhroni (synchronous) ili blokirajući (blocking**): pozivajuči proces, odnosno tok kontrole se po potrebi suspenduje i nastavlja tek kada se zahtevana operacija završi

**asinhroni (asynchronous) ili neblokirajući (non-blocking)**: pozivajući proces, odnosno tok kontrole se odmah nastavlja, sistemski poziv mu odmah vraća kontrolu, a zahtevana operacija se obavlja “odloženo”, asinrono

Unix-like sistemi imaju **specijalne fajlove, device file,** koje procesi vide kao virtuelne fajlove u virtuelnom direktorijumu /dev, sa kojima mogu da obavljaju uobičajene operacije sa fajlovima, medjutim, te operacije se preusmeravaju ka odgvorajućim uredjajima.

Ove fajlove instalira administrator komandom mknod .   
int ioctl(int fd, unsigned long request, ...); -fd je deskriptor otvorenog fajla. Direktno upućivanje zahteva ovakvim fajlovima.

Svakom procesu pridružena 3 standardna ulaza:

**stdin** - ulaz

**stdout** - izlaz

**stderr** – izlazni uredjaj za greške, npr. za ispis sistemskim pozivom error

Proces dete nasledjuje od roditelja.

myprogram <input.txt >output.txt 2>error.txt

<input.txt preusmerava stdin na fajl input.txt   
>output.txt preusmerava stdout na fajl output.txt   
2>error.txt preusmerava stderr na fajl error.txt

Sistemski poziv dup2(oldfd, newfd) radi sledeće: kopira deskriptor fajla dat sa oldfd u deskriptor fajla newfd, tako da newfd zapravo ukazuje na isti fajl kao i deskriptor oldfd;   
Za newfd - vrednost 0 uvek označava deskriptor stdin, 1 označava stdout, a 2 stderr.

Štampanje

lpr mydoc.txt

cat mydoc.txt | lpr

lpr šalje mydoc.txt na podrazumevani štampač. **Pipe |** prosledjuje output komande sa leve strane na input komande sa desne strane.

Memorijski preslikani fajlovi

void\* mmap (void \*addr, size\_t length, int prot, int flags, int fd, off\_t offset);

int munmap (void \*addr, size\_t length);

Sadržaj fajla koji je prethodno otvoren i čiji je deskriptor dat u *fd*, i to počev od pomeraja *offset* i dužine *length*, preslikava se u deo memorijskog prostora počev od adrese *addr*; ako je *addr jednak null*, kernel će sam izabrati slobodan deo vrituelnog adresnog prostora procesa; *length* mora biti umnožak veličine stranice koja se dobija pozivom *sysconf(\_SC\_PAGE\_SIZE)*.   
Polje *flags* može biti:

MAP\_SHARED – svi procesi koji preslikavaju isti fajl, dele isti fajl.

MAP\_PRIVATE – proces ima svoju nezavisnu kopiju. Koristi se copy on write.

Realno vreme

Sistemski pozivi time, sleep, timedwait.

Slika

**Framebuffer** – posebni deo operativne memorije koji koristi I video adapter. Deo VAP procesa OS slika u ovaj deo operativne memorije kako bi proces mogao da upisuje u njega jer se on koristi kao **bafer sa bitmapom** jednog kadra slike koji video adapter koristi da bi tu sliku prikazao na ekranu.

Drajveri

Da bi ostatak kernela bio nezavisan od uređaja, ali i da bi kernel bio spreman da prihvati stalno nove uređaje, za komunikaciju sa svakom klasom uređaja postoji poseban deo, modul kernela, koji implementira odgovarajući interfejs prema ostatku kernela za datu klasu uređaja - skup operacija koje kernel poziva za ovu klasu uređaja (npr. prenos jednog znaka, prenos bloka podataka itd.). To je **drajver**, modul koji je zadužen za neposrednu komunikaciju se hardverskim uredjajem, pravi spregu izmedju uredjaja I ulazno izlaznog podsistema OS-a.

**Prilikom priključivanja** novog uredjaja, generički drajver može već postojati, ako ne, može se instalirati.

**Drajver** je zapravo softverski modul koji obezbeđuje implementaciju skupa potprograma (polimorfnih operacija) koje kernel očekuje od date klase uređaja.

Kontroleri uredjaja I DMA

**Control registar** – parametri kontrolera, startovanje kontrolera.  
**Status registar** – ready bit.  
**Data registar** – za upis u uredjaj Ili čitanje.

**Pooling ili prozivanje ili busy waiting** – čitanje statusnog registra radi provere bita spremnosti.  
**Prekid** – ready bit statusnog registra vezan za liniju zahteva za prekid uredjaja. Generiše se prekid svaki put kad je uredjaj spreman.

**DMA kontroler** – kontroler sa direktnim pristupom memoriji koji služi za prenos bloka podataka iz memorije na uredjaj I obrnuto. Razlog njegovog postojanje je da bi procesor mogao da se bavi sofisticiranijim zadacima, a ne da se troši procesorsko vreme na prenos podataka.

**DMA** kontroler generiše **prekid samo jednom**, kad je gotov, I tad ga u prekidnoj rutini treba ugasiti postavljanjem odgovarajućih bita kontrolnog registra.

**BIOS (Basic Input/Output System)** – nalazi se u ROM-u. Softver koji koristi kernel, odnosno drajveri uredjaja radi obavljanja elementarnih ulazno izlaznih operacija.

Upravljanje diskovima

**Hard disk** je blokovski ulazno-izlazni uredjaj sa direktnim pristupom I prezistentnim sadržajem.  
Blokovi u disku su obeleženi rednim brojevima odnosno adresama.   
Osnovne operacije sa diskovima su:

int readBlock I writeBlock (BlockNo block, void\* buffer);

Hard disk se sastoji od više diskova naslaganih a izmedju njih je skoro vakuum u kom se nalazi **glava diska za čitanje I upis**. Zapisi su organizovani koncentričnu u **trake**, svaka traka ima blok odnosno **sektor**.

Zahtevi za pristup disku se stavljaju u red čekanja. Posebna nit kernela upućuje zahteve ka disku. Proces je za to vreme blokiran.

Kada se blok čiji je sadržaj u memoriji izmenjen upisuje na disk. Mogući su različiti pristupi:

odmah pri svakoj izmeni, sinhrono (tzv. **store through**)

odloženo, asinhrono, npr. prilikom izbacivanja (tzv. **write back**) – **flush**, upis na disk.

**Dvostruko baferisanje** – dva bafera, jedan puni producer a iz drugog uzima consumer. Kad se jedan napuni a drugi isprazni vrši se njihova zamena tako da producer opet puni prazan bafer a consumer čita iz punog.

**Spuling**. Ovo je rešenje toga da kad proces zatraži uredjaj, dobije ga odnosno ne blokira se, on predugo drži uredjaj zauzetim. Pogotovo sa štampačem, operacije traju značajno dugačko.  
Svi procesi upisuju u isti fajl u koji će OS preusmeriti sve izlazne operacije. Posebna nit, **spuler**, preusmerava poslove, **job**, na uredjaj. Zapravo se radi o baferisanju.

Fajl sistem

Fajlovi

**Fajl** (file) je logički koncept koji omogućava čuvanje sadržaja na uređajima i pristup tom sadržaju nezavisan od vrste tih uređaja. To je jedini način da korisnički proces smešta podatke. Fajl je osmišljen da bi programe učionio nezavisnim od različitosti u načinu smeštanja sadržaja.

Informacija o tipu fajla može biti u ekstenziji, kao eksplicitan atribut fajla, ili **magic number** – prvih nekoliko bajtova fajla.

**Serijalizacija** – program svoje interne strukture pretvori u niz bajtova za upis u fajl.   
**Deserijalizacija** – iz niza bajtova program gradi strukture.

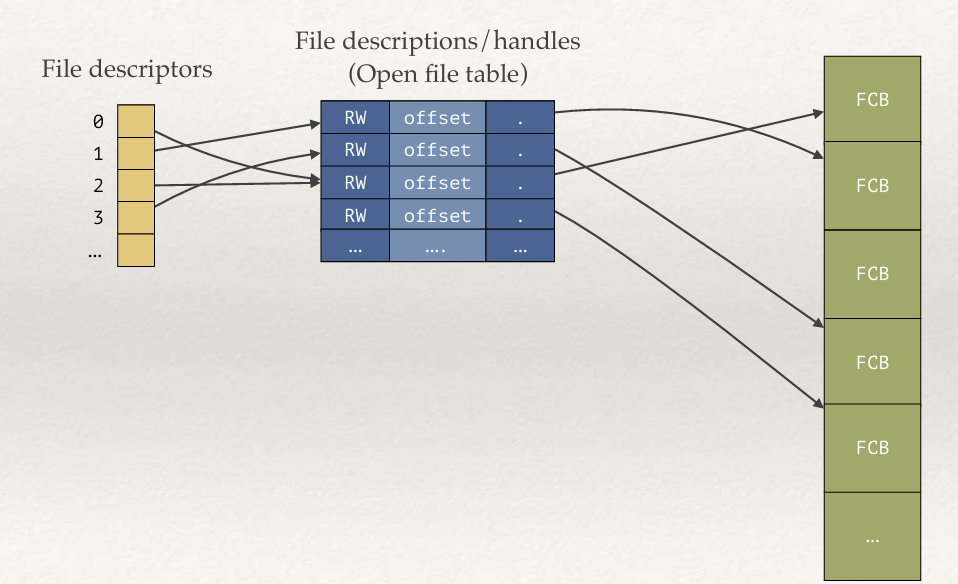
Odgovornost je na programima da rade serijalizaciju ili deserijalizaciju, ne na OS.

int open (const char \*pathname, int flags, mode\_t mode);

**Pathname** – ime fajla.   
**Flags** – modaliteti.

O\_CREAT – ako fajl sa zadatim imenom ne postoji, biće kreiran. Ako ovaj fleg nije uključen, greška. Ako je uključen još I O\_EXCL – vratiće grešku ako fajl postoji.  
O\_RDONLY, O\_WRONLY, O\_RDWR – najava operacija nad fajlom. Mora biti bar 1 uključen.  
O\_APPEND – fajl se otvara za proširenje. Kurzor se postavlja na kraj fajla.  
O\_TRUNC – ako je dozvoljen upis, fajl se otvara a sadržaj se briše.  
O\_DIRECT – mora sinhroni upis I čitanje uz zaobilaženje internih keševa.

Poslednji parametar **mode** **ima značenje samo ako je u argumentu flags postavljen O\_CREAT** (fajl se kreira), u suprotnom ovaj argument može da se izostavi. Tada se ovim argumentom zadaju prava pristupa do fajla, opet postavljanjem simboličkih konstanti.

**File descriptors** – za svaki proces. Čuva se pokazivač na ulaz u tabeli otvorenih fajlova.

**Tabela otvorenih fajlova** – za svaki proces. Sadrži ručku odnosno – najavljena prava pristupa, trenutnu poziciju kurzora, pokazivač na **FCB** datog fajla.

**FCB** – globalna struktura koju mogu da dele I drugi procesi.

Open vraća indeks ulaza u tabeli deskriptora.

Ulazi 0, 1, 2 su stdin, stdout, stderr.



Posix close ne garantuje flush dok C-ov fclose garantuje.

Posix:

ssize\_t read (int fd, void \*buffer, size\_t count);   
ssize\_t write (int fd, const void \*buffer, size\_t count);   
off\_t lseek (int fd, off\_t offset, int whence);

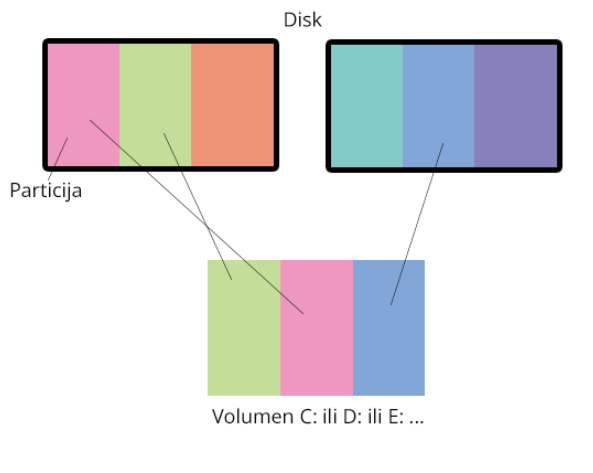
**whence** – u odnosu na šta se postavlja tekuća pozicija - SEEK\_SET - u odnosu na početak sadržaja fajla, SEEK\_CUR - za offset bajtova u odnosu na tekuću poziciju, SEEK\_END - offset bajtova iza kraja sadžaja fajla (može biti i negativan).

C-ovi:

std::size\_t fread (void\* buffer, std::size\_t size, std::size\_t count, std::FILE\* stream);   
std::size\_t fwrite (const void\* buffer, std::size\_t size, std::size\_t count, std::FILE\* stream);   
int fseek (std::FILE\* stream, long offset, int origin);

**Truncate** – brisanje sadržaja fajla.

Direktorijum

Na svakom volumenu se može organizovati fajl sistem **– logička formatizacija diska**. Ovo je **cooked particija**.

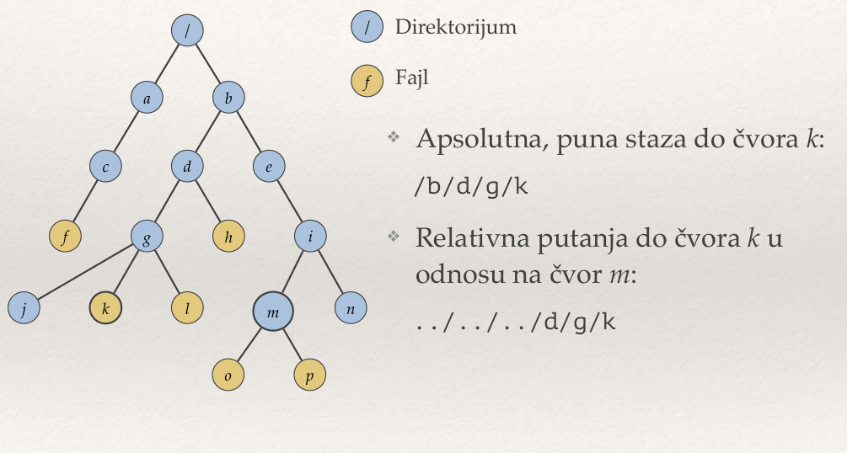
**Raw particija** nema fajl sistem, može se koristiti kao **swap space**.

Osnovni zadatak **direktorijuma** je da logičko ime fajla preslika u njegov FCB, kako bi OS mogao da pronadje tu strukturu I učita je sa uredjaja.

Ako bi fajl sistem bio organizovan kao:

*Jedan direktorijum* – teško za snalaženje, nepraktično jer svi fajlovi moraju da imaju drugačije ime, kako razvrstati fajlove na višekorisničkom sistemu.

*Fajlovi različitih korisnika u različite direktorijume* – još uvek nepregledno, gde smestiti fajlove kojima svi korisnici treba da pristupaju.

**Hijerarhijska struktura** – u obliku stabla.

**Prosto** – **nekvalifikovano ime** – ne mora biti jedinstveno u celom fajl sistemu već samo u roditeljskom direktorijumu.

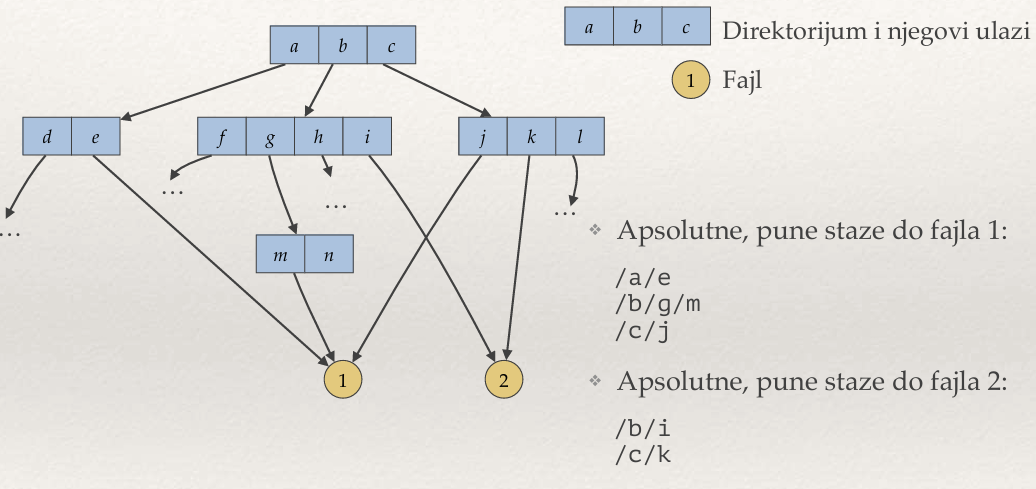
**Puno** – **kvalifikovano ime** – apsolutna putanja od korenog čvora do fajla.

.. – roditeljski direktorijum

. – trenutni direktorijum

Problem – korisnik organizuje I klasifikuje fajlove prema svojim nahodjenjima. Nekad je potrebno isti fajl klasifikovati prema različitim kriterijumima.

**DAG** – **usmeren graf bez petlji**.

**Tvrda veza** (hard link) i **meka veza** (soft link) su dve vrste veza koje se mogu uspostaviti između fajlova u fajl sistemu.

Glavna razlika između ove dve vrste veza je u tome što tvrda veza uspostavlja vezu između dva imena fajla u okviru istog fajl sistema, dok se meka veza uspostavlja između dva fajla na različitim lokacijama u fajl sistemu, pri čemu se jedan fajl koristi kao **simbolička referenca** na drugi fajl.

Kada se **tvrda veza** kreira, dva imena fajla pokazuju na isti blok podataka na disku, tako da ako jedan fajl bude promenjen, drugi fajl će takođe biti promenjen. Tvrde veze ne mogu biti uspostavljene između fajlova na različitim fajl sistemima, niti između direktorijuma.

lnk komanda za pravljenje tvrde veze. Ili sistemski poziv link.

Kada se briše tvrda veza, ne briše se ceo fajl nego samo odgovarajući ulaz u direktorijum. Tek kada se obriše poslednja veza, briše se i ceo fajl. To se postiže **brojanjem referenci**, inkrementiranjem kad se pravi nova veza i dekrementiranjem kad se briše.

Upravo kako bi sprečili formiranje kružnih staza, mnogi sistemi **zabranjuju kreiranje tvrdih veza ka direktorijumima**; kako tvrda veza mora biti u direktorijumu, a mora referencirati samo fajl koji je uvek list, garantovano nema petlji u grafu.

**Meka veza** je poseban fajl koji sadrži putanju do originalnog fajla, a kada se pristupi mekoj vezi, sistem koristi putanju za pronalaženje originalnog fajla. Ako se originalni fajl obriše, meka veza će pokazivati na nepostojeći fajl. Meka veza se može uspostaviti između fajlova na različitim fajl sistemima, kao i između direktorijuma.

**Implementacija meke veze** u fajl sistemu se obično vrši pomoću posebnog fajla koji sadrži putanju do originalnog fajla (**simbolička veza**). Kada se pristupi mekoj vezi, sistem pročita putanju iz fajla i koristi je za pronalaženje originalnog fajla. Fajlovi koji predstavljaju meke veze se obično označavaju nekom posebnom oznakom (npr. znakom link) kako bi se razlikovali od običnih fajlova.

Jedan aspekt implementacije jeste to *gde se smešta putanja na referencirani fajl*; opcije su:

u samom sadržaju fajla, kao tekstualni zapis, dok poseban bit u FCB ukazuje na to da je fajl simbolička veza

kao atribut fajla, u strukturi FCB, što je efikasnije (tzv. **brze veze, fast links**)

ln -s target\_path link\_path-*target\_path* je relativna ili apsolutna putanja do referenciranog fajla a *link\_path* staza do simboličke veze.

mklnk na Windowsu.

ls –l mydoc prikazuje simboličke veze preko ->

Prava pristupa

Korisnik-fajl-operacija je relacija koja se mora napraviti.

Svakom fajlu se može pridružiti lista zapisa parova – **korisnik, operacija** – koji definišu operacije dozvoljene korisnicima za taj fajl. To je **ACL – access control list**.  
*Problemi*: održavanje i pretraga manje efikasni. Za svaki fajl, svaku operaciju i svakog korisnika se mora napraviti zapis.

U praksi – **korisnici, fajlovi, operacije**.

*Unix:*

Svaki korisnik ima svoj **uid**, I grupa kojoj pripada ima svoj **gid**. Svaki proces se izvršava u ime korisnika, odnosno, kontekst procesa uključuje I uid I gid. Kad proces kreira fajl u nečije ime, taj korisnik postaje **owner** – uid I gid su atributi čvora.

Za svaki čvor su definisana prava pristupa za – **owner**, **group** – prava za korisnike koji pripadaju istoj grupi kao vlasnik, **others**.  
*Primer*: komanda: ls -l  
**drwxr-xr-x** 4 JohnDoe staff 128 Sep 27 2013 FolderName  
*d – direktorijum, ako nema ništa onda je fajl  
rwx – dozvoljena čitanja, upisa I izvršavanja – owner  
r-x – samo upis zabranjen – group  
r-x – samo upis zabranjen - others*

chmod u-x, g=rw, o-w mydoc - vlasniku oduzme pravo izvršavanja, grupi dozvoli SAMO čitanje I upis a ostalima oduzme upis, za fajl mydoc.

Uporedan pristup fajlovima

**File locking** – fajl se može zaključavati

*Implicitno* – pri svakoj operaciji sa fajlom

*Eksplicitno* – na zahtev procesa

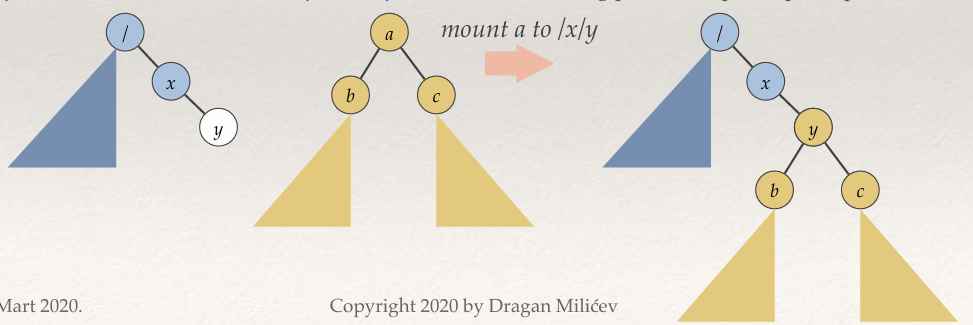
Vrste ključeva:

Jedan ključ koji može imati samo jedan proces I proces može dobiti ključ samo ako ga nijedan drugi ne drži.

**Deljeni (shared)** – svi procesi koji imaju ovaj ključ mogu samo da čitaju, uporedo I to samo ako nijedan proces nema ekskluzivni ključ.

**Ekskluzivni** – potreban za operacije upisa. Proces može dobiti ekskluzivni ključ samo ako nijedan proces nema nikakav ključ nad tim fajlom.

Montiranje fajl sistema

Da bi OS omogućio pristup do fajl sistema na nekom volumenu ili spoljnom uredjaju, mora prvo **montirati** taj fajl sistem.

Samo se stablo priključi na posebna mesta u fajl sistemu kao podstablo.  
**Unmounting** – eject na Windowsu.

Pristup udaljenim fajlovima

Koristi se FTP protokol gde oba računara moraju imati instaliran odgovarajući program I pomoću komanda kao što su get I put, računar klijent može od računara servera da primi fajl ili uploaduje.

Posebna vrsta operativnih sistema ili specijalizovanog softvera (npr. za sisteme u oblaku) implementira distribuirane fajl sisteme (**distributed file system, DFS**). DFS omogućava jedinstven pogled na strukturu direktorijuma i fajlova koji su fizički raspoređeni na različitim računarima u distribuiranom sistemu, uz rešavanje problema particionisanja mreže (nedostupnosti nekih čvorova zbog otkaza računara ili veze) i konzistentnosti promena sadržaja fajlova.

Implementacija fajl sistema

**Boot Control Block**: uvek na određenom mestu, tipično blok broj 0 na volumenu; ukoliko prvi bajt ili bajtovi u ovom bloku imaju neku definisanu vrednost (tzv. magic number), **bootstrap** program u ROM-u računara prepoznaje da ovaj blok sadrži bootstrap program za učitavanje operativnog sistema; pri podizanju računara, ovaj blok se učitava i kod u njemu izvršava za dalje učitavanje sistema; ako particija sadrži ovaj blok, naziva se **bootable**; u suprotnom, ovaj blok se ne koristi.

**Volume Control Block**: na unapred definisanom mestu, odnosno bloku na volumenu, sadrži osnovne informacije o volumenu i globalne parametre fajl sistema na njemu, kao što su npr. ukupan broj blokova, veličinu bloka, eventualno broj slobodnih blokova, pokazivač na prvi slobodan blok u listi i slično.

Kao što je već rečeno, **FCB** (File Control Block) je struktura podataka koja se zapisuje na disku za svaki fajl i sadrži atribute fajla, uključujući i informacije o tome gde je i kako smešten sadržaj tog fajla; na sistemima nalik sistemu Unix, ova struktura naziva se **inode**.

Gde se I kako smeštaju strukture FCB na volumenu:

*Unix* – može se nalaziti **bilo gde** dok u direktorijumu se nalazi informacija gde se nalazi – preslikava simboličko ime u FCB/inode.

*NTFS* – **Master File Table** – u ulazima se nalaze FCB.

Sistemi nalik sistemu **Unix**: **direktorijum se predstavlja** na isti način kao i fajl, ima isto svoj FCB (inode, zato se i zovu jednim imenom, čvor), samo što **jedan bit u FCB ukazuje na to da se radi o direktorijumu**; sam sadržaj ovakvog “fajla” čuva informacije o preslikavanjima simboličkih elemenata čvorova u adrese njihovih struktura FCB.

Neke tipične strukture su sledeće:

**tabela montiranih fajl sistema** koja u svakom ulazu ima strukturu koja opisuje jedan montiran fajl sistem

**tabela deskriptora fajlova** (file descriptor) za svaki proces, kao deo konteksta procesa (slika); svaki ulaz ukazuje na opis fajla (file description), odnosno ulaz u tabeli otvorenih fajlova

**tabela otvorenih fajlova lokalnih za procese (local open file table**, slika), u kojoj svaki ulaz pripada kontekstu nekog procesa; ulaz je opis ili ručka fajla (**file description, file handle, struct file**) i sadrži tekuću poziciju za operacije čitanja i upisa za dati proces, operacije koje je proces najavio prilikom otvaranja, kao i pokazivač na FCB datog fajla

**globalna tabela otvorenih fajlova (global open file table)**, odnosno tabela struktura FCB (inode) za fajlove koje neki od procesa još uvek drži otvoren, i koji je učitan sa diska, a sadrži atribute fajla i informacije o mestu i načinu organizacije njegovog sadržaja

**keširane strukture za ulaze u direktorijumima** kojima se već pristupalo; kako je zadatak direktorijuma da preslika simboličko ime u (adresu strukture) FCB u koje se preslikava to ime, pri čemu se te informacije nalaze zapisane na disku, OS kešira ona preslikavanja (ulaze u direktorijumima) koja su prethodno već bila korišćena, kako ih ne bi svaki put iznova učitavao sa diska

**keš blokova** sa sadržajem fajlova kojima se nedavno pristupalo

Kako se fajlovi mogu smeštati na različite uređaje, pa čak biti i udaljeni, da bi rešio ovu varijabilnost, OS opisane pojedinačne strukture (ulaze u tabelama koji opisuju pojedinačne objekte) zapravo vidi kao objekte koji zadovoljavaju neki interfejs (skup operacija koje se od njih zahtevaju), a koji je definisan u nekoj apstraktnoj osnovnoj klasi. Ceo ovaj skup interfejsa naziva se virtuelni fajl sistem (**Virtual File System, VFS**).

*Na slajdovima se nalazi primer šta se dešava kad se pozove open I read. Bolje pogledati I to.*

Direktorijum može slikati simboličko ime u:

*Direktno u FCB* odnosno da se u direktorijum ugradi FCB struktura. Tad ne bi mogle višestruke tvrde veze.

*Pokazivač na FCB*, broj bloka (UNIX) ili broj ulaza u globalnoj tabeli u kojoj su sve strukture FCB (NTFS).

*Identifikator strukture FCB*, a onda se adresa (broj bloka) gde se nalazi FCB dobija iz neke centralizovane strukture (mape), globalne za ceo volumen, preslikavanjem tog identifikatora.

Za implementaciju direktorijuma se može koristiti **hash tabela**. Slika simboličko ime u FCB. Rešavanje kolizije ulančavanjem.

Metode alokacije fajla

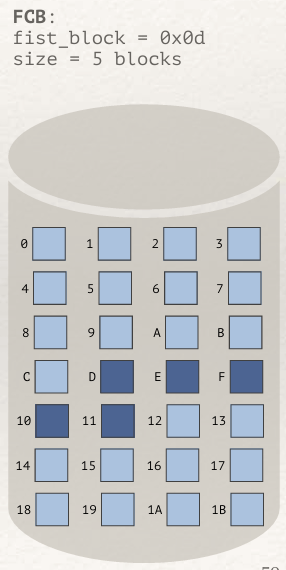
*Vrste pristupa fajlu jesu:*

**sekvencijalan pristup**: karakterističan za znakovne tokove, odnosno za fajlove kojima se tekuća pozicija

ne pomera ekplicitno (operacijom seek), već samo implicitno, tako da se sadržaju fajla pristupa redom

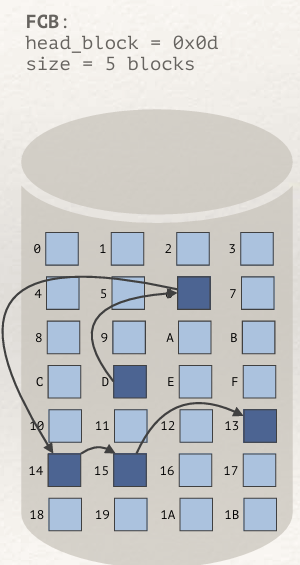
**direktan pristup**: pristupa se sadržaju u proizvoljnom redosledu (eksplicitne operacije seek)

**Kontinualna alokacija** – sadržaj fajla zauzima susdene blokove a FCB ima samo broj početnog bloka I broj blokova.

Jednostavan sekvencijalan I direktan pristup – kao nizu, sabiranjem početnog bloka sa traženim.

*Problemi*:   
Mora se koristiti neki algoritam – first fit, best fit.  
Mora se unapred znati veličina fajla.  
Proširivanje fajla otežano, mora realokacija ako nema slobodnog mesta iza.  
**Eksterna fragmentacija** – može se rešiti **kompakcijom**.

**Ulančana alokacija** – ulančana lista blokova. FCB sadrži glavu bloka I broj blokova. Svaki blok ima pokazivač next.

Jednostavan sekvencijalan pristup, alokacija bilo gde, nema eksterne fragmentacije, ne mora se znati veličina bloka unapred, može se prošitivati fajl.

*Problemi*:  
Neefikasan direktan pristup.  
Odvaja se prostor za smeštanje pokazivača.  
Ako se pokazivač ošteti, lako se **korumpira** fajl. Čak se može I prevezati pokazivač da ukazuje na drugi fajl čime dolazi do **kompromitovanja**.

Potrošnja prostora na pokazivače se može rešiti povećanjem veličine bloka, ili alokacija više susednih blokova kao jedinice alokacije – **klaster**. Time se povećava **interna fragmentacija**.

Problem korupcije se može rešiti upisivanjem redudantnih podataka u blok:

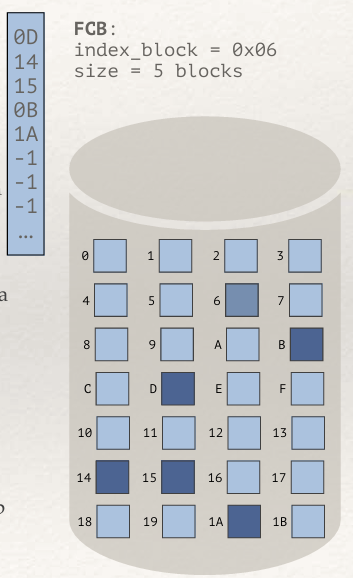
*Identifikator FCB kom blok pripada*

*Dupli pokazivač* – u ispravnom stanu oba pokazivača pokazuju na isti blok.

**File Allocation Table – FAT** – na posebnom unapred definisanom mestu na volumenu se nalazi tabela koja ima broj ulaza koliko I blokova na disku. Jedan na jedan se slikaju fajlovi – indeks 0 odgovara nultom bloku, 1 odgovara prvom bloku …  
Blokovi su ulančani korišćenjem indeksa. Time su pokazivači za ulančavanje odvojeni od blokova I u blokovima se smeštaju samo podaci. FAT se kešira.

*Problemi*: osetljivost na otkaze. Dovoljno je da se jedan pokazivač korumpira I može ceo fajl sistem da otkaže.  
Može se delimično ublažiti pravljenjem duple kopije FAT ali onda se mora dvostruko ažurirati.

**Indeksna alokacija** – svaki fajl ima svoj indeksni blok koji se nalazi na disku. Indeksni blok u sebi ima tabelu sa brojem

blokova koji fajl zauzima.

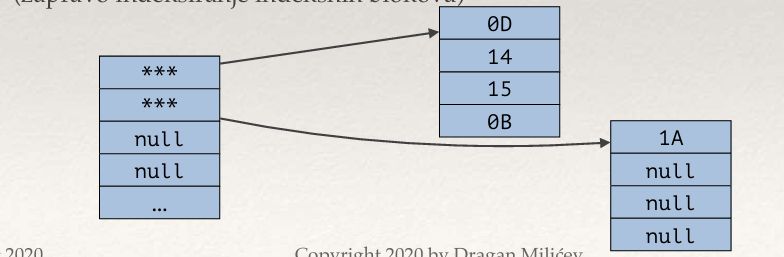
Direktan I sekvencijalan pristup efikasni. Manje podložan otkazima. Indeksni blok može biti bilo gde. Jednostavne operacije sa fajlovima.

*Problemi*:  
Trošenje dodatnog prostora.

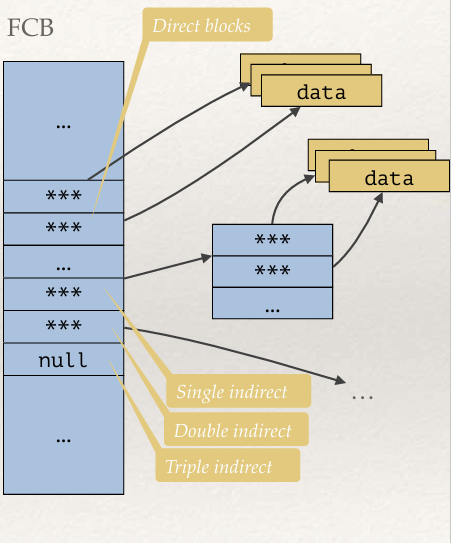
To se može rešiti pravljenjem manjih indeksnih blokova, ali to znači I manje dozvoljene veličine fajlova.

Manje veličine fajlova se mogu rešiti:

sam indeks se može *organizovati kao ulančana lista blokova*, tako da se indeks može dinamički proširivati

indeks se može *organizovati u više nivoa*, kao stablo, poput PMT u više nivoa (zapravo indeksiranje indeksnih blokova)

Sistem nalik Unixu koristi **kombinovanu** varijantu.



*u samom FCB nalazi se indeks nultog nivoa* — određen broj ulaza koji sadrže adrese blokova sa neposrednim sadržajem; dok je sadržaj fajla dovoljno mali, koristi se samo ovaj indeks

kada sadržaj fajla poraste preko kapaciteta nultog indeksa, u FCB postoji ulaz koji sadrži broj bloka sa indeksom prvog nivoa (**single indirect**), u kom je spisak blokova sa sadržajem

i tako dalje, čak do indeksa trećeg nivoa

Tako da sad mali fajlovi zauzimaju mali prostor, a veći fajlovi su podržani pomoću indirekcije. Za male fajlove je pogodna kontinualna alokacija.

Evidencija slobodnih blokova

**Bit vektor** – svaki bit u vektoru odgovara jednom bloku. Alokacija se svodi na pretragu za bitom 1 ili 0. Savremeni procesori imaju instrukcije koje kao rezultat vraćaju broj razreda u bajtu ili reči u kom se nalazi prva jednica odnosno nula sleva ili zdesna. Bit vektor se smešta na volumenu a kešira u kešu.

Nepogodnost je što se mora odvajati poseban prostor - na primer, za prostor od 1 TB i blokom veličine 512 B potrebno je 2 G bita, odnosno 256 MB za bit vektor.

**Ulančavanje slobodnih blokova** – alokacija jednog bloka efikasna, više nije. Dealokacija je efikasna. U fat je već podržano rukovanje slobodnim prostorom jer slobodni blokovi imaju null ulaz u FAT.

Performanse I zaštita

Za sekvencijalni pristup, npr. kada se fajl otvori sa najavom sekvencijalnog pristupa, ili kada OS sam prepozna da proces pristupa fajlu sekvencijalno, učitava blokove unapred, kako bi bili spremni kad budu potrebni (**read-ahead**).

**Vodjenje dnevnika – Journaling**.

OS vodi zapisnik, linearna lista koja se čuva na disku na unapred odredjenom mestu, pri svakom upisu zapisa u zapisnik, blok se snima sinhrono.  
Početak transakcije: **<start>**  
Sadržaj bloka u kešu se menja i kad se završi, blok se prvo zapiše na neko posebno slobodno mesto, dok se original ne menja. Tad se u zapisnik stavi **<oldBlock, newBlock>**, adrese blokova.  
Kad se opercija završi - **<commit>.** Potom se jedan po jedan blok prepisuje iz privremene kopije u original.   
Konačno, zapisnik se briše ako je operacija uspešno završena.

Kada bi do otkaza računara došlo pre nego što se desi commit – u zapisniku će ostati zapis delimično izvršene transakcije. Kad se OS podigne i vidi da nema commit, transakcija se odbacuje a svi blokovi koji su čuvali nove verzije se proglašavaju slobodnim.

Ako otkaz dodje nakon commit, prosto se ponovo krene prepisivanje iz starog u novi blok, ispočetka sve.

Procedura pravljenja rezervne kopije (backup) može biti:

**totalna**: kopiraju se svi fajlovi iz jednog fajl sistema ili njegovog dela (direktorijuma), nezavisno od toga da li su izmenjeni u odnosu na verzije na postojećoj kopiji, pa se takvi neizmenjeni fajlovi bespotrebno ponovo kopiraju

**inkrementalna** (incremental): kopiraju se samo izmenjeni fajlovi; ovo, naravno, može biti značajno brže