Raspoređivanje procesora

1 Uvod

Raspoređivanje CPU-a je u osnovi operativnih sistema koji omogućavaju multi-programiranje. Prebacivanjem CPU-a među procesima, operativni sistem može da učini računar produktivnijim. U ovom poglavlju predstavljamo osnovne koncepte raspoređivanja CPU-a i predstavljamo nekoliko algoritama koji se koriste u tu svrhu. Takođe, razmatramo problem izbora algoritma za određeni sistem. U prethodnim predavanjima smo predstavili niti kao sastavni deo procesa. Za operativne sisteme koji ih podržavaju, operativni sistem raspoređuje zapravo niti, a ne procese. Međutim, izrazi „raspoređivanje procesa" i „raspoređivanje niti" često se koriste naizmenično. U ovom poglavlju koristimo termine raspoređivanje (zakazivanje) procesa kada govorimo o generalnim konceptima raspoređivanja i raspoređivanje (zakazivanje) niti kako bismo govorili o idejama koje su specifične za niti.

2 Osnovni koncepti

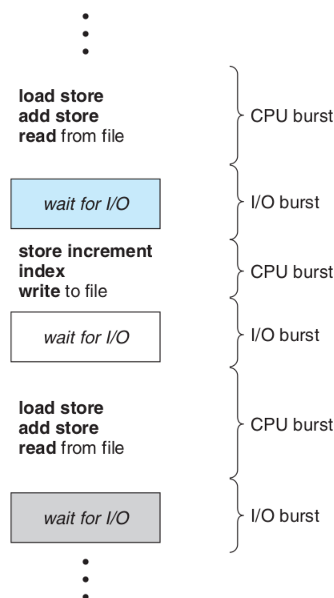
Na sistemima sa jednim procesorskim jezgrom, samo jedan proces se izvršava u datom trenutku, dok ostali čekaju da se CPU oslobodi i prepusti drugom procesu. Cilj multi-programiranja je da obezbedi izvršavanje nekog od procesa u svakom trenutku, kako bi se maksimizirala iskorišćenost procesora. Ideja je relativno jednostavna. Proces se izvršava dok ne dođe u situaciju da mora da sačeka, tipično na završetak zahtevane U/I operacije. U jednostavnom sistemu, CPU je tada besposlen, a svo vreme čekanja je uzalud potrošeno u takvom scenariju, jer se nikakav koristan rad nije obavio. Dodatkom multi-programiranja, pokušavamo da iskoristimo ove periode produktivno. Nekolicina procesa se nalaze u memoriji u datom trenutku, kada jedan od njih mora da čeka, OS će oduzeti CPU tom procesu i dodeliti ga drugom, ponavljajući takvo ponašanje ciklično. Svaki period kada jedan proces mora da čeka, može biti iskorišćen kako bi se drugi proces izvršavao.

Raspoređivanje ove vrste je fundamentalna funkcija operativnog sistema. Gotovo svi računarski resursi se raspoređuju pre korišćenja, pri čemu je CPU, svakako, jedan od primarnih računarskih resursa. Stoga, raspoređivanje CPU-a je centralni problem u dizajnu operativnog sistema.

3 CPU - U/I ciklusi

Uspešnost raspoređivanja CPU-a zavisi od ponašanja procesa: izvršavanje procesa se sastoji od ciklusa u kojima se izvršavaju CPU instrukcije i čeka na U/I događaj. Proces neprekidno menja ova dva stanja tokom svog izvršavanja. Izvršavanje procesa počinje sa CPU ciklusom, kojeg prati U/I ciklus, za kojim ide CPU ciklus, itd. Na kraju, završni CPU ciklus prethodi zahtevu da se terminira izvršavanje procesa (slika 1).

Trajanje CPU ciklusa je od izuzetnog značaja. Iako uveliko varira od procesa do procesa i od računara do računara, CPU ciklusi uglavnom imaju histogram



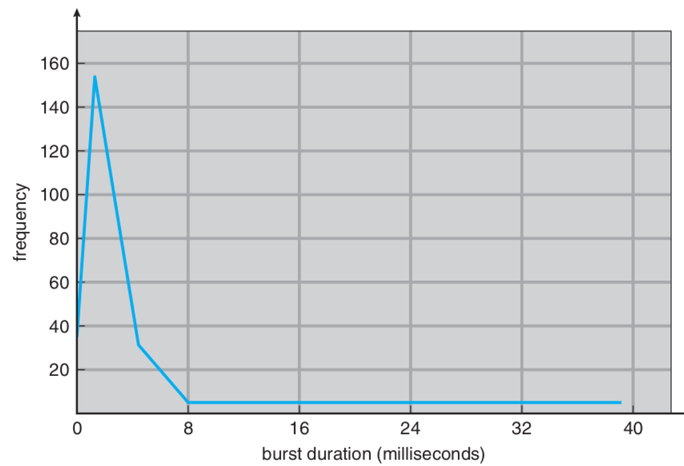
Slika 1: Sekvenca CPU i U/I ciklusa koji se smenjuju

sličan onom prikazanom na slici 2. Histogram CPU ciklusa prikazuje učestalost CPU ciklusa u zavisnosti od dužine njihovog trajanja. Nekada se prikazuje koristeći logaritamsku ili eksponencijalnu razmeru, ali sa krive se može videti da se često dešavaju kratki CPU ciklusi i retko dugački. U/I kontrolisani proces tipično ima mnogo kratkih CPU ciklusa, dok CPU kontrolisani program može imati nekoliko veoma dugačkih CPU ciklusa. Poznavanje ove raspodela je veoma značajno prilikom selekcije odgovarajućeg algoritma koji će se koristiti za CPU raspoređivanje.

4 CPU planer

Uvek kada CPU postane besposlen, OS mora odabrati jedan od procesa iz reda spremnih procesa za izvršavanje. Ovaj odabir se vrši od strane kratkoročnog planera ili CPU planera. Planer odabira proces iz skupa procesa koji se nalaze u memoriji i koji su spremni za izvršavanje i dodeljuje CPU tom procesu.

Red spremnih procesa ne mora nužno biti FIFO red. Kao što ćemo videti kasnije kada budemo analizirali različite algoritme za raspoređivanje, red spremnih procesa može biti implementiran kao FIFO red, red sa prioritetima (prioritetni red, eng. *priority queue*), stablo ili jednostavno ne-sortirana povezana lista. Konceptualno, svi procesi u redu spremnih procesa čekaju na svoju šansu da se izvršavaju na datom CPU-u. Zapisi u redu spremnih procesa su uglavnom kontrolni blokovi procesa (PCB).



Slika 2: Histogram CPU ciklusa

5 Prisvojivo raspoređivanje (preemptive scheduling)

Odluke u vezi sa raspoređivanjem procesora donose se u sledeće 4 situacije:

1. Kada proces pređe is stanja izvršavanja u stanje čekanja (npr. kao rezultat U/I zahteva ili čekanje na terminiranje dete procesa)
2. Kada proces pređe iz stanja izvršavanja u stanje spreman (npr. kada se desi prekid)
3. Kada proces pređe iz stanja čekanja u stanje spreman (npr. nakon završetka U/I operacije koja se čekala)
4. Kada se proces terminira

Za situacije 1 i 4 od gore, zapravo ne donosi se nikakva odluka sa stanovišta raspoređivanja - novi proces (ako takav postoji u redu čekanja spremnih procesa) mora biti odabran za izvršavanje. Sa druge strane, izbor postoji u slučajevima 2 i 3.

Kada se raspoređivanje vrši samo u slučajevima 1 i 4, kažemo da je raspoređivanje neprisvojivo (eng. *nonpreemptive*) ili kooperativno (eng. *cooperative*), u suprotnom kažemo da je prisvojivo (eng. *preemptive*). Pod kooperativnim raspoređivanjem, kada se CPU dodeli procesu, proces ga zadržava sve dok se ne terminira ili pređe u stanje čekanja. Ovakva politika raspoređivanja je korišćena u Windows 3.x operativnom sistemu, dok je u Windows 95 operativnom sistemu uvedeno prisvojivo raspoređivanje, koje se koristi u svim narednim verzijama Windows operativnog sistema. Na nekim platformama jedino kooperativno raspoređivanje dolazi u obzir jer ono ne zahteva dodatni hardver (na

primer tajmer) neophodan za prisvojivo raspoređivanje. Ipak, generalno, svi savremeni operativni sistemi koriste prisvojivo raspoređivanje.

Nažalost, prisvojivo raspoređivanje rezultuje stanjem utrkivanja (*race condition*) o kojem smo detaljno pričali na prethodnom predavanju. Ono se dešava kada se podaci dele između više procesa. Osim toga, prisvajanje takođe ima uticaj na dizajn kernela operativnog sistema. Tokom obrade sistemskog poziva, kernel može biti zauzet aktivnostima koje obavlja u ime procesa. Ovakve aktivnosti mogu da uključe modifikacije kernel struktura (na primer U/I redova čekanja). Šta se dešava ako se proces prekine tokom ovih izmena, pa kernel (ili upravljački program-drajver uređaja) čita ili modifikuje iste te strukture? Određeni operativni sistemi, uključujući i većinu onih baziranih na Unix-u, rešavaju ovaj problem tako što ili čekaju da se izvrši do kraja sistemski poziv ili čekaju da se blokiranje usled U/I operacije desi, pre nego što se krene sa zamenom konteksta. Ovakva šema osigurava jednostavne kernel strukture, obzirom na to da kernel neće prekinuti proces dok su kernel strukture podataka u prelaznom stanju. Nažalost, ovakav model nije prikladan u cilju podrške obrađivanja podataka u realnom vremenu, gde određeni zadatak mora biti izvršen u datom vremenskom roku. Pričaćemo posebno o tome kasnije tokom ovog predavanja.

6 Dispečer

Dispečer je još jedna komponenta uključena u raspoređivanje procesora. Dispečer je modul koji daje kontrolu nad CPU-om procesu koji je odabran od strane kratkoročnog planera. On je zadužen za:

1. Zamenu konteksta
2. Prelazak u korisnički mod izvršavanja
3. Prelazak na odgovarajuću lokaciju u korisničkom programu kako bi se nastavilo izvršavanje na adekvatan način

Dispečer treba da bude brz koliko god je to moguće, pošto se poziva prilikom svake zamene procesa. Vreme koje je potrebno kako bi dispečer prekinuo jedan proces i pokrenuo drugi zove se *latentnost dispečera*.

7 Kriterijumi za raspoređivanje

Različiti algoritmi za raspoređivanje CPU-a imaju različite karakteristike, a izbor odgovarajućeg algoritma može favorizovati jednu klasu procesa u odnosu na procese iz druge klase. Prilikom odabira algoritma koji će se koristiti u datoj situaciji, moramo uzeti u obzir karakteristike ovih algoritama.

Postoji mnoštvo kriterijuma koji se koriste prilikom poređenja algoritama za raspoređivanje procesora:

1. Iskorišćenost procesora - želimo da držimo CPU zauzetim što je moguće duže. Konceptualno, iskorišćenost procesora može da bude u opsegu od

0-100 procenata. U realnom sistemu, trebala bi da bude između 40 (za slabo opterećen sistem) do 90 (za veoma opterećen sistem).

2. Propusnost - predstavlja broj procesa koji se kompletiraju u jedinici vremena. Za dugotrajne procese, propustnost može biti jedan proces na sat, za kratkotrajne može biti desetak procesa u sekundi.

3. Ukupno vreme izvršavanja (*Turnaround time*) - predstavlja proteklo vreme od prijave procesa (zahteva da se proces izvršava) u sistemu do trenutka terminiranja procesa. Ovo vreme se sastoji od zbira perioda čekanja da proces dospe u memoriju, čekanja u redu spremnih procesa, izvršavanja na CPU-u i čekanja na U/I operacije.

4. Vreme čekanja - algoritam raspoređivanja ne utiče na period vremena tokom kojeg se proces izvršava ili vrši U/I operacije. On samo utiče na period vremena koji proces provodi čekajući u redu čekanja spremnih procesa. Vreme čekanja predstavlja sumu perioda koje proces provede čekajući u redu čekanja spremnih procesa.

5. Vreme odziva - kod interaktivnih sistema, ukupno vreme izvršavanja uglavnom nije dobar kriterijum. Često proces generiše izlaz brzo a nakon toga nastavi izračunavanje novih rezultata dok se prethodni prikazuju korisniku. U svetlu ovoga, vreme odziva se odnosi na vreme proteklo od prijave procesa do trenutka kada se generiše prvi odgovor.

Cilj je da se poveća iskorišćenost procesora, kao i propusnost, a da se minimizira ukupno vreme izvršavanja, vreme čekanja i vreme odziva. U većini slučajeva, pokušavamo da optimizujemo prosečnu meru. Ipak, u nekim situacijama, preferiramo optimizaciju minimalne ili maksimalne vrednosti, pre nego prosečne. Na primer, kako bismo garantovali da će svi korisnici biti prihvatljivo opsluženi, želimo da minimizujemo maksimalno vreme odziva.

Istraživanja su dovela do zaključka da je kod interaktivnih sistema (kao na primer desktop računara), važnije minimizovati varijansu vremena odziva, nego minimizovati prosečno vreme odziva. Sistem kod koga je vreme odziva prihvatljivo i predvidivo, više se ceni od sistema koji se brži u proseku, ali veoma nepredvidiv u terminima odziva. Ipak, gotovo nijedan algoritam za raspoređivanje procesora ne uzima u obzir ovu varijansu kao parametar.

U nastavku ćemo analizirati različite algoritme za raspoređivanje procesora. Precizna ilustracija njihovog rada podrazumevala bi mnogo procesa, od kojih je svaki sekvenca nekoliko stotina CPU ciklusa i U/I ciklusa. Ipak, zbog jednostavnosti, posmatraćemo samo jedan CPU ciklus (u milisekundama) po svakom procesu, a metrika koju koristimo za poređenje algoritama odnosi se na srednje vreme čekanja.

8 Algoritmi za raspoređivanje procesora

Ovi algoritmi rešavaju problem odlučivanja koji od procesa u redu spremnih procesa će biti odabran za izvršavanje. Postoji mnogo algoritama koji se koriste,

Slika 3: Gantov dijagram (*Gantt chart*) izvršavanja tri procesa u FCFS maniru

a ovde ćemo prikazati samo neke od njih.

8.1 First-Come, First Served (FCFS) algoritam

Najjednostavniji algoritam za raspoređivanje procesora je FCFS algoritam. On će za izvršavanje odabrati proces koji je prvi podneo zahtev, odnosno koji je prvi stigao u red spremnih procesa, tako da se njegova implementacija svodi na korišćenje FIFO reda spremnih procesa. Kada proces dospe u red spremnih procesa, njegov PCB je povezan na kraj (eng. *tail*) reda procesa. Proces koji će sledeći da se izvršava uzima se sa početka reda, tako da je ovaj algoritam jednostavan za razumevanje i implementaciju.

Negativna strana ovog algoritma ogleda se u činjenici da je srednje vreme čekanja kod FCFS algoritma prilično dugačko. Razmotrimo sledeći skup procesa koji se pojavljuju u trenutku t=0, svaki dat sa CPU ciklusom prikazanim u milisekundama.

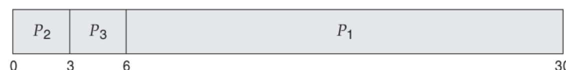
Proces	Period izvršavanja
P_1	24
P_2	3
P_3	3

Ako procesi pristižu u red spremnih procesa u redosledu P_1,P_2, P_3 dobijamo raspoređivanje procesa prikazano Gantovim dijagramom 3 na kome se vide trenuci početka i završetka izvršavanja svakog procesa.

Vreme čekanja je 0 ms za proces P_1, 24 ms za proces P_2 i 27ms za proces P_3. Stoga, srednje vreme čekanja je (0ms + 24ms + 27ms)/3 = 17 ms. Ako bi procesi stizali u drugom redosledu P_2, P_3, P_1, rezultujući Gantov dijagram bi bio:

Prosečno vreme čekanja bi sada bilo (6ms + 3ms + 0ms) / 3 = 3ms. Ovo smanjenje srednjeg vremena izvršavanja je značajno i možemo da zaključimo da korišćenjem FCFS algoritma dobijamo srednje vreme čekanja koje nije minimalno i značajno varira ako vremena izvršavanja procesa variraju.

Dodatno, razmotrimo ponašanje FCFS algoritma u dinamičkom okruženju. Pretpostavimo da imamo jedan CPU-kontrolisani proces i više U/I-kontrolisanih



Slika 4: Gantov dijagram (*Gantt chart*) izvršavanja tri procesa u FCFS maniru

procesa. Kako procesi kruže sistemom, moguće je da se desi sledeći scenario u kome CPU-kontrolisani proces dobije CPU na korišćenje i drži ga zauzetim. Za to vreme, svi ostali procesi završe sa svojim U/I operacijama i premeste se u red čekanja spremnih procesa, tako da će U/I uređaji biti besposleni. Vremenom, CPU-kontrolisani proces završi svoj CPU ciklus i zahteva pristup U/I uređaju. Svi U/I-kontrolisani procesi, koji imaju kratke CPU cikluse, izvrše se brzo i vraćaju se u U/I redove čekanja. U ovom periodu CPU je besposlen. Nakon izvesnog vremena, CPU-kontrolisani proces se premešta u red spremnih procesa i biće mu dodeljen procesor. Ponovo, svi U/I-kontrolisani procesi će završiti u redu čekanja spremnih procesa dok CPU-kontrolisani proces ne završi svoj CPU ciklus. Ovo se naziva konvoj efekat (*convoy effect*) jer svi procesi čekaju jedan „veliki“ proces da završi sa korišćenjem CPU-a. Ovakav efekat rezultuje znatno neefikasnijim korišćenjem CPU-a i U/I uređaja nego što bi bio slučaj kada bi se „kratki“ procesi prvi izvršavali.

Takođe, treba primetiti da je FCFS algoritam neprisvojiv. Kada se jednom dodeli CPU datom procesu, taj proces koristi CPU sve dok sam ne odluči da ga oslobodi, bilo zbog toga što treba da se terminira, bilo zbog toga što se blokira nakon U/I zahteva. Stoga je FCFS izuzetno nezgodan i nepraktičan za sisteme sa deljenjem vremena (multitasking), gde je izuzetno važno da svaki korisnik dobije deo CPU vremena u regularnim intervalima vremena. Posledice bi bile pogubne ako bi se jednom procesu dozvolilo da koristi CPU duže nego što je planirano.

8.2 Shortest Job First (SJF) algoritam

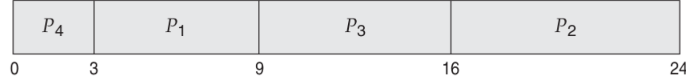
Drugačiji pristup CPU raspoređivanju donosi SJF algoritam. Ovaj algoritam pridružuje svakom procesu dužinu trajanja njegovog narednog CPU ciklusa. Kada CPU bude dostupan, dodeliće se procesu koji ima najkraći naredni CPU ciklus. Ako dva procesa imaju naredne CPU cikluse jednakog trajanja, FCFS se koristi da odluči koji od njih će se prvi izvršavati. Samim tim, prikladniji naziv algoritma bi svakako bio *Shortest next CPU burst first* algoritam, obzirom na to da raspoređivanje zavisi od dužine narednog CPU ciklusa datog procesa, pre nego ukupnog preostalog trajanja procesa. Ipak, koristićemo termin SJF pošto se u literaturi uglavnom uvek koristi ovaj naziv.

Kao primer, posmatramo skup procesa sa CPU ciklusima prikazanim u milisekundama:

Proces	CPU ciklus
P_1	6
P_2	8
P_3	7
P_4	3

Korišćenjem SJF raspoređivanja, dobićemo Gantov dijagram prikazan na slici 5.

Vreme čekanja je 3ms za P_1, 16ms za P_2, 9ms za P_3i 0ms za P_4. Dakle, srednje vreme čekanja je (3ms + 16ms + 9ms + 0ms) / 4 = 7ms. Poređenja



Slika 5: Gantov dijagram (*Gantt chart*) izvršavanja četiri procesa u SJF maniru

radi, da smo koristili FCFS algoritam, prosečno vreme čekanja u ovom primeru bi bilo 10.25ms.

SJF algoritam je verovatno optimalan u smislu srednjeg vremena čekanja za dati skup procesa. Prioritizovanjem kratkih procesa pre dugačkih smanjuje vreme čekanja kratkih procesa više nego što produžava vreme čekanja dugačkih procesa. Kao posledica, smanjuje se prosečno vreme čekanja.

Međutim, problem kod SJF algoritma ogleda se u činjenici da je potrebno poznavanje vremena izvršavanja narednog CPU ciklusa za dati proces. Iako je optimalan, SJF se ne može implementirati kod kratkoročnog raspoređivanja, gde je nemoguće znati koliko vremena će nositi naredni CPU ciklus za neki od procesa. Jedno rešenje problema je da se pokuša aproksimirati SJF raspoređivanje: možda ne znamo trajanje narednog CPU ciklusa, ali možemo da predvidimo koliko će ono biti. Računanjem aproksimativne dužine izvršavanja narednog CPU ciklusa, možemo da odaberemo proces sa najkraćim očekivanim CPU ciklusom.

Naredni CPU ciklus se najčešće predviđa kao eksponencijalno usrednjavanje (eng. *exponential average*) merene dužine prethodnih CPU ciklusa. Ako je t_n dužina izvršavanja n-tog CPU ciklusa, predviđeno trajanje CPU ciklusa n+1 računa se kao:

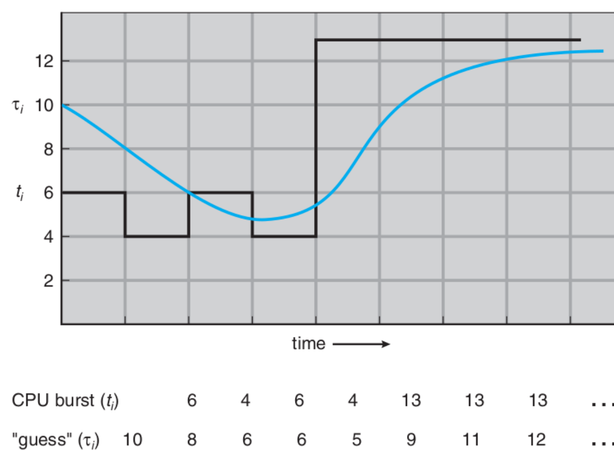
$$\tau_{n+1} = \alpha t_n + (1-\alpha)\,\tau_n$$

gde je $0 \leq \alpha \leq 1$, vrednost t_n sadrži najskoriji CPU ciklus, dok τ_n čuva istoriju prethodnih vrednosti. Parametar α kontroliše relativnu težinu najskorijeg rezultata i prethodne istorije. Ako je vrednost $\alpha = 0$ tada je $\tau_{n+1} = \tau_n$ i predtodna istorija nema nikakvog uticaja na predviđanje narednog CPU ciklusa. Sa druge strane, ako je $\alpha = 1$ tada je $\tau_{n+1} = t_n$ i samo najskoriji CPU ciklus se uzima u obzir prilikom predviđanja narednog, a prethodni istorijat nema uticaja na njega. Češće se koristi ipak $\alpha = \frac{1}{2}$, što za posledicu daje jednak uticaj poslednje merene vrednosti CPU ciklusa i istorijata prethodnih vrednosti. Inicijalni τ_0 može da se odabere kao konstanta, ili kao srednja vrednost CPU ciklusa na nivou sistema. Slika 6 prikazuje računanje eksponencijalnog usrednjavanja dobijenog sa $\alpha = \frac{1}{2}$ i $\tau_0 = 10$.

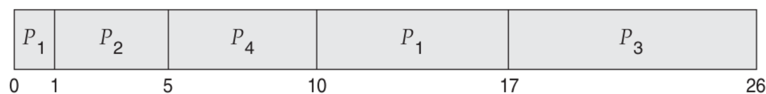
Da bi se razumelo ponašanje sistema sa eksponencijalnim usrednjavanjem, proširićemo formulu:

$$\tau_{n+1} = \alpha t_n + (1-\alpha)\,\alpha t_{n-1} + \cdots + (1-\alpha)^j\,\alpha t_{n-j} + \cdots + (1-\alpha)^n\,\tau_0$$

Tipično, α je manje od 1, pa kao rezultat i $(1-\alpha)$ je takođe manje od 1 i svaki naredni član ima manju težinu u odnosu na prethodni.



Slika 6: Predviđanje narednog CPU ciklusa



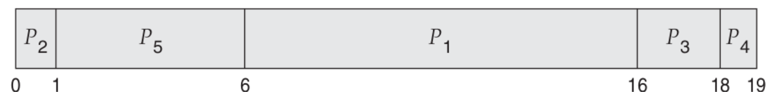
Slika 7: Gantov dijagram na primeru sa prisvojivim SJF

SJF algoritam može da bude kako *prisvojiv* tako i *neprisvojiv*. Izbor se pojavljuje u trenutku kada novi proces dospeva u red spremnih procesadok se prethodni proces i dalje izvršava. Naredni CPU ciklus novopristiglog procesa u redu čekanja spremnih procesa može biti kraći od preostalog ciklusa procesa koji se trenutno izvršava. Prisvojivi SJF algoritam tada prekida trenutni proces i umesto njega raspoređuje novopristigli proces u redu čekanja. Suprotno tome, kod neprisvojivog SJF algoritma, proces koji se trenutno izvršava će završiti svoje izvršavanje pre nego što se novopristiglom procesu dodeli procesor. Shodno tome, prisvojivi SJF algoritam se često naziva i *Shortest Remaining Time First* (SRTF) raspoređivanje.

Kao primer, razmatramo četiri procesa od kojih je svaki dat sa svojim CPU ciklusom ali i trenutkom pristizanja u red spremnih procesa:

Proces	Pristigao	CPU ciklus
P_1	0	8
P_2	1	4
P_3	2	9
P_4	3	5

Ako procesi pristižu u red spremnih procesa u skladu sa drugom kolonom iz gornje tabele, prisvojivi SJF će rezultirati sa Gantovim dijagramom prikazanim na slici 7.



Slika 8: Gantov dijagram na primeru sa prioritetnim raspoređivanjem

Proces P_1počinje da se izvršava u trenutku 0, jer je on jedini proces u redu čekanja u datom trenutku. Proces P_2pristiže u trenutku 1. Preostalo vreme za proces P_1(7ms) je kraće od vremena potrebnog za proces P_2 (4ms), tako da je proces P_1 prekinut, a proces P_2 kreće da se izvršava. Srednje vreme čekanja za ovaj primer je $[(10ms - 1ms) + (1ms - 1ms) + (17ms - 2ms) + (5ms - 3ms)]/4 = 6,5ms$. Da je korišćen neprisvojivi SJF algoritam srednje vreme čekanja bi bilo 7,75ms.

8.3 Prioritetno raspoređivanje

Prethodno prikazani SJF algoritam je specijalan slučaj generalizovane klase algoritama sa *prioritetnim raspoređivanjem.* Kod njih je prioritet dodeljen svakom procesu a CPU se dodeljuje procesu sa najvišim prioritetom. Procesi istog prioriteta se raspoređuju u FCFS maniru, baš kao i u slučaju SJF algoritma. SJF algoritam je zapravo algoritam sa prioritetnim raspoređivanjem kod kojeg se prioritet (p) određuje kao inverzna vrednost (predviđenog) narednog CPU ciklusa. Što je duže predviđeno naredno vreme izvršavanja, to je niži prioritet, i obrnuto.

Iako o prioritetima govorimo u smislu visokog ili niskog prioriteta, generalno, prioriteti se postavljaju korišćenjem brojeva iz nekog fiksnog opsega, npr. 0 do 7 ili 0 do 4095. Ipak, ne postoji usvojen dogovor u vezi sa tim da li je 0 najviši ili najniži prioritet - neki sistemi koriste male brojeve da predstave nizak prioritet, dok drugi koriste male brojeve da predstave visok prioritet.

Kao primer, posmatramo skup procesa koji pristižu u red čekanja spremnih procesa u trenutku 0, sa trajanjem CPU ciklusa datim u tabeli:

Proces	Prioritet	CPU ciklus
P_1	3	10
P_2	1	1
P_3	4	2
P_4	5	1
P_5	2	5

Korišćenjem prioritetnog raspoređivanja, dobija se Gantov dijagram dat na slici 8.

Srednje vreme čekanja u ovom primeru je 8,2ms.

Prioriteti mogu biti definisani kako interno tako i eksterno. Interno definisani prioriteti postavljeni su na osnovu neke merljive metrike (ili više njih) kako bi se izračunao prioritet procesa. Na primer, metrika bi mogla biti vezana za vremensko ograničenje, zahtev za memorijom, broj otvorenih datoteka, kao i

odnos prosečnog U/I ciklusa i CPU ciklusa. Eksterni prioriteti su postavljeni van operativnog sistema, u skladu sa značajem procesa.

Prioritetno raspoređivanje, takođe, može biti prisvojivo i neprisvojivo. Kada proces dospe u red spremnih procesa, njegov prioritet se poredi sa prioritetom procesa koji se trenutno izvršava. U slučaju prisvojivog prioritetnog raspoređivanja, proces koji se trenutno izvršava će biti prekinut ukoliko je prioritet novopristiglog procesa viši. Kod neprisvojivog prioritetnog raspoređivanja, novopristigli proces će se samo smestiti na početak reda spremnih procesa, kako bi on bio prvi koji se naredni izvršava.

Glavni problem kod algoritma prioritetnog raspoređivanja je *neograničeno blokiranje* ili *izgladnjivanje*. Proces koji je spreman da se izvršava ali čeka na CPU se, u ovom smislu, smatra blokiranim. Prioritetno raspoređivanje može da ostavi neke procese niskog prioriteta u blokiranom stanju neograničeno dugo vremena: ukoliko je procesor preopterećen, sekvenca procesa visokog prioriteta može sprečiti procese nižeg prioriteta da se ikada domognu CPU-a. Dve su moguća ishoda u ovom slučaju: ili će se proces eventualno izvršiti (kada sistem bude manje opterećen tj. kada procesi visokog prioriteta završe svoja izvršavanja), ili će se sistem srušiti pri čemu će se izgubiti svi procesi niskog prioriteta koji su čekali na procesorsko vreme.

Rešenje gorepomenutog problema neograničenog blokiranja procesa niskog prioriteta je *zastarevanje* (eng. *Aging*). Zastarevanje podrazumeva postepeno uvećavanje prioriteta procesa koji čekaju u sistemu dugo vremena. Na primer, ako su prioriteti u opsegu od 127 (nizak) do 0 (visok), možemo uvećavati prioritet proces koji čeka za 1 svakih N minuta (ili N ms u skladu sa željenim performansama). Vremenom, čak i proces koji je imao najniži prioritet imao bi najviši prioritet u sistemu i izvršavao bi se.

8.4 Round-Robin (RR) raspoređivanje

RR algoritam za raspoređivanje je dizajniran za sisteme sa deljenim vremenom (multitasking sisteme). Sličan je FCFS raspoređivanju, ali prisvajanje je dodato kako bi se dozvolilo sistemu da zamenjuje procese koji se izvršavaju. Definisan je kratak interval vremena, koji se naziva *vremenski kvant* (eng. *time quantum*, *time slice*). Trajanje vremenskog kvanta je obično od 10 do 100ms. Red spremnih procesa se posmatra kao cirkularni red. CPU planer prolazi kroz red čekanja spremnih procesa i alocira CPU svakom od njih u periodu od maksimalno jednog vremenskog kvanta.

Implementacija RR algoritma, dakle zahteva korišćenje reda spremnih procesa kao FIFO reda. Novi procesi se dodaju na kraj reda, a CPU planer uzima procese sa početka reda, postavlja tajmer da generiše prekid nakon 1 vremenskog kvanta i prosleđuje proces dispečeru.

Nakon toga, dešava se jedan od dva moguća scenarija:

1. Proces će imati CPU ciklus trajanja kraćeg od vremenskog kvanta. U ovom slučaju proces će sam osloboditi CPU, a planer će nastaviti sa sledećim procesom u redu spremnih procesa;

P_1	P_2	P_3	P_1	P_1	P_1	P_1	P_1

0 4 7 10 14 18 22 26 30

Slika 9: Gantov dijagram na primeru sa RR raspoređivanjem

2. Proces ima CPU ciklus duži od vremenskog kvanta. U ovom slučaju tajmer će signalizirati da je vreme za izvršavanje isteklo generisanjem prekida, kao rezultat doći će do zamene konteksta, a prethodno izvršavani proces biće postavljen na kraj reda čekanja spremnih procesa. Nakon toga, CPU planer odabira sledeći proces iz reda čekanja.

Srednje vreme čekanja kada se koristi RR raspoređivanje je obično veoma veliko. Razmotrimo sledeći skup procesa koji svi dostižu u red čekanja u trenutku 0, a vremenski kvant je 4ms.

Proces	CPU ciklus
P_1	24
P_2	3
P_3	3

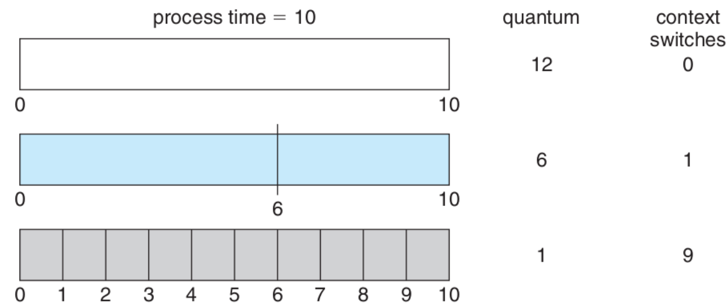
Proces P_1 dobija prve 4ms. Pošto on zahteva još 20ms, biće prekinut nakon isteka prvog vremenskog kvanta, a CPU će se dodeliti drugom procesu u redu čekanja, procesu P_2. Proces P_2 ne treba 4ms za izvršavanje i biće gotov pre nego što istekne vremenski kvant. CPU je tada dodeljen sledećem procesu P_3 i tek nakon što su svi procesi izvršavani (maksimalno vremenski kvant vremena), kontrola se ponovo prepušta procesu P_1, u trajanju od još jednog vremenskog kvanta. Rezultujući Gantov dijagram prikazan je na slici 9.

Računica srednjeg vremena čekanja je: P_1 čeka 6ms (10ms - 4ms), P_2 čeka 4ms a P_3 čeka 7ms, što daje srednje vreme čekanja 17ms/3=5,66ms.

U RR raspoređivanju, nijedan proces nije dobio CPU na korišćenje duže od 1 vremenskog kvanta u nizu (osim ukoliko je on trenutno jedini proces koji je spreman za izvršavanje). Ako dati proces ima CPU ciklus duži od vremenskog kvanta, on će biti prekinut i vraćen nazad u red čekanja spremnih procesa, što čini RR algoritam raspoređivanja prisvojivim.

Ako imamo n procesa u redu spremnih procesa i ako je vremenski kvant q, svaki proces dobija $1/n$ CPU vremena u delovima maksimalnog trajanja q. Sveki proces neće čekati duže od $(n-1)\cdot q$ do svog sledećeg vremenskog kvanta. Na primer, sa 5 procesa i vremenskim kvantom 20ms, svaki proces dobija do 20ms svakih 100ms (najviše).

Performanse RR algoritma umnogome zavise od veličine vremenskog kvanta. U jednom ekstremnom slučaju, ako je vremenski kvant prevelik, RR raspoređivanje će biti isto kao FCFS raspoređivanje. Sa druge strane, ako je vremenski kvant ekstremno mali (npr. 1ms) RR algoritam će rezultovati u velikom broju zamena konteksta. Zamislimo, na primer, da imamo samo jedan proces trajanja 10 vremenskih jedinica. Ako je vremenski kvant 12 vremenskih jedinica, proces će se završiti pre isteka vremenskog kvanta. Međutim, ako je vremenski



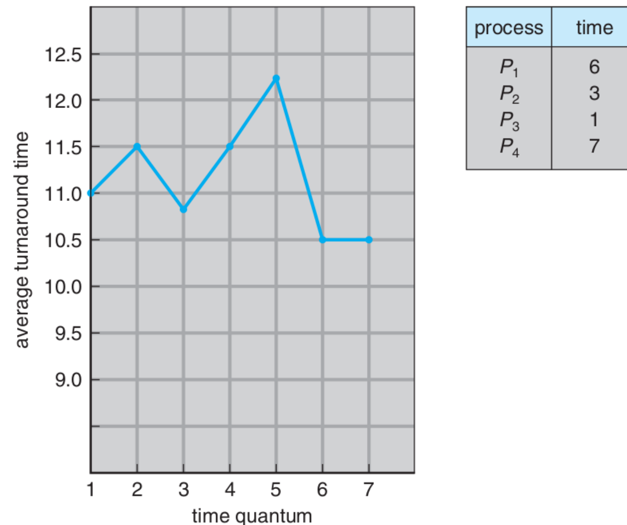
Slika 10: Uticaj vremenskog kvanta na performanse sistema

kvant 6 vremenskih jedinica, proces će zahtevati dva kvanta da bi se izvršio, što će rezultovati zamenom konteksta. Ukoliko bi vremenski kvant bio 1 vremensku jedinicu dugačak, 9 zamena konteksta bi se desilo, značajno usporavajući izvršavanje procesa, kao što je prikazano na slici 10.

Kao rezultat gornje analize, potrebno je da vremenski kvant bude velik u poređenju sa vremenom potrebnim za zamenu konteksta. Ako je vreme potrebno za zamenu konteksta približno 10% vremenskog kvanta, onda se otprilike 10% CPU vremena provodi u zameni konteksta. U praksi, većina savremenih sistema imaju vremenski kvant koji je u opsegu 10ms do 100ms, dok je vreme zamene konteksta tipično kraće od $10\mu s$, što je veoma mali deo vremenskog kvantuma.

Ukupno vreme izvršavanja takođe zavisi od veličine vremenskog kvanta. Kao što se može videti sa slike 11, prosečno ukupno vreme izvršavanja skupa procesa ne mora nužno da se poboljšava kako se povećava vremenski kvant. Generalno, prosečno ukupno vreme izvršavanja može biti poboljšano ako većina procesa završi svoj naredni CPU ciklus tokom narednog dobijenog vremenskog kvanta. Na primer, ako imamo tri procesa sa vremenom izvršavanja 10 vremenskih jedinica svaki i vremenski kvantum od 1 vremenske jedinice, prosečno ukupno vreme izvršavanja je 29. Ako bi vremenski kvant bio 10, međutim, prosečno ukupno vreme izvršavanja je 20. Ako se uračunava i vreme potrebno za zamenu konteksta, prosečno ukupno vreme izvršavanja se povećava dodatno za mali vremenski kvant, obzirom na to da je više zamena konteksta potrebno.

Iako vremenski kvant treba da bude dovoljno velik u poređenju sa vremenom potrebnim za zamenu konteksta, on ipak ne treba da bude prevelik. Kao što smo rekli ranije, povećavanjem vremenskog kvanta, RR raspoređivanje se deformiše u FCFS raspoređivanje. Grubo pravilo kojeg se treba držati u ovom smislu jeste da se odabere vremenski kvant kod kojeg je 80% CPU ciklusa kraće od vremenskog kvanta.



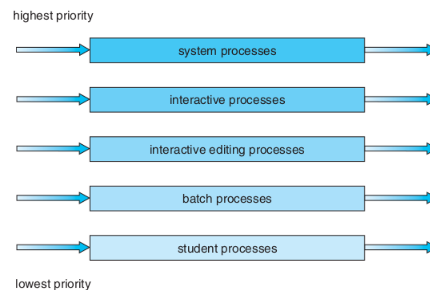
Slika 11: Uticaj vremenskog kvanta na prosečno ukupno vreme izvršavanja

8.5 Raspoređivanje u višestrukim redovima (Multilevel Queue Scheduling)

I ova klasa algoritama za raspoređivanje je osmišljena u cilju korišćenja u situacijama kada se procesi mogu jednostavno klasifikovati i različite grupe. Na primer, uobičajena je podela na interaktivne procese koji se izvršavaju u prvom planu (eng. *foreground* - interakcija sa korisnikom) i pozadinske procese (eng. *background*). Ova dva tipa procesa imaju različite zahteve u vezi sa vremenom odziva, tako da mogu imati i značajno različite zahteve u pogledu raspoređivanja. Dodatno, interaktivni procesi mogu imati i viši prioritet (eksterno definisan) u odnosu na pozadinske procese.

Algoritmi za raspoređivanje u višestrukim redovima particionišu red spremnih procesa u više odvojenih redova (slika 12). Proces je permanentno dodeljen jednom od redova, uglavnom bazirano na nekim osobinama procesa, kao što je veličina memorije, prioritet procesa ili tip procesa. Svaki red ima svoj sopstveni algoritam za raspoređivanje. Na primer, odvojeni redovi mogu da se koriste za interaktivne i pozadinske procese, pri čemu interaktivni mogu da se raspoređuju u skladu sa RR algoritmom, dok se pozadinski raspoređuju korišćenjem FCFS algoritma.

Dodatno, u ovom scenariju mora postojati i raspoređivanje između redova, koje je uglavnom implementirano kao raspoređivanje u odnosu na fiksne prioritete. Na primer, u primeru od gore, red interaktivnih procesa ima apsolutni prioritet u odnosu na red pozadinskih procesa, što znači da se neki proces iz reda pozadinskih procesa može izvršavati samo ukoliko je red interaktivnih procesa prazan.



Slika 12: Primer raspoređivanja u višestrukim redovima

Pogledajmo primer raspoređivanja u višestrukom redu sa pet redova, izlistanih po prioritetima od najvišeg do najnižeg, sa slike 12.

U ovom primeru, svaki red ima apsolutni prioritet u odnosu na redove sa nižim prioritetom. Ni jedan proces iz 4. reda, na primer, ne može da se izvršava ukoliko sva tri prva reda nisu prazni. Ukoliko je proces pristigao u neki od ovih redova višeg prioriteta dok se izvršava proces iz 4. reda, on će biti prekinut i procesor će biti dodeljen procesu višeg prioriteta.

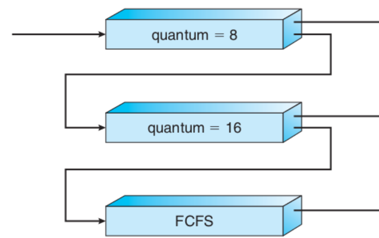
Još jedna mogućnost je da se dodatno deli vreme između redova. U ovom pristupu svaki red dobija određeni procenat CPU vremena, koje onda može da koristi za raspoređivanje svojih procesa. Na primer, ako govorimo o podeli na interaktivan red i pozadinski red, interaktivni može dobiti 80% CPU vremena za RR raspoređivanje, dok će pozadinski red koristiti preostalih 20% CPU vremena za raspoređivanje procesa u FCFS maniru. Na ovaj način se obezbeđuje da će se i procesi sa nižim prioritetom izvršavati.

8.6 Dinamičko raspoređivanje u višestrukim redovima (Multilevel Feedback Queue Scheduling)

Uobičajeno, ako se koristi raspoređivanje u višestrukim redovima, procesi se permanentno smeštaju u jedan od redova kada pristižu u sistem. Ako postoje, na primer, redovi za interaktivne i pozadinske procese, procesi se neće premeštati iz reda u red, pošto procesi ne menjaju svoju prirodu, u tom smislu. Ovakav pristup zahteva jednostavno raspoređivanje, ali je veoma nefleksibilan.

Algoritmi za dinamičko raspoređivanje u višestrukim redovima, nasuprot, dozvoljava svakom procesu da se premešta iz reda u red, tokom vremena. Ideja je da se odvoje procesi u skladu sa karakteristikama u pogledu CPU ciklusa. Ako proces koristi mnogo CPU vremena, on će biti premešten u red nižeg prioriteta. Ovakva shema ostavlja U/I-kontrolisane procese i interaktivne procese u redovima visokog prioriteta. Dodatno, ako proces čeka predugo u redu niskog prioriteta, on može biti premešten u red visokog prioriteta kako bi se sprečilo *izgladnjivanje* (ovo omogućava jednu implementaciju *zastarevanja*).

Kao primer, razmatramo planer koji dinamički raspoređuje procese u tro-



Slika 13: Primer dinamičkog raspoređivanja u višestrukim redovima

strukom redu, pri čemu su redovi označeni sa 0 do (slika 13). Planer najpre izvršava sve procese u redu 0. Samo ako je red 0 prazan, procesi iz reda 1 će se izvršavati. Slično, procesi iz reda 2 će se izvršavati samo ako su prazni redovi 0 i 1. Proces koji pristigne u red 1 će prekinuti izvršavanje procesa u redu 2, a proces iz reda 1 će biti prekinut ukoliko se pojavi proces pristigao u red 0.

Proces koji pristigne u red spremnih procesa se smešta u red 0, gde mu je dodeljen vremenski kvant od 8ms. Ako se proces ne završi u predviđenom vremenu, on se premešta na kraj reda 1. Ukoliko je red 0 prazan, proces sa početka reda 1 će dobiti vremenski kvant 16ms za izvršavanje. Ukoliko se ne završi u tom predviđenom intervalu vremena, biće prekinut i premešten u red 2. Procesi u redu 2 se izvršavaju u FCFS maniru, ali samo ukoliko su red 0 i red 1 prazni.

Ovakav algoritam raspoređivanja daje najviši prioritet bilo kom procesu koji ima CPU ciklus manji ili jednak sa 8ms. Ovakav proces će brzo dobiti CPU, završiti sa svojim CPU ciklusom i preći na U/I ciklus. Procesi koji zahtevaju više od 8ms, ali manje od 24ms su takođe usluženi brzo, ali sa nižim prioritetom od kratkih procesa. Dugački procesi automatski prelaze u red 2 gde se izvršavaju u FCFS maniru tokom bilo kojeg preostalog CPU vremena neiskorišćenog od strane procesa iz redova 0 i 1.

Generalno, planer koji koristi dinamičko raspoređivanje u višestrukim redovima definisan je pomoću sledećih parametara:

1. Broj redova
2. Algoritam raspoređivanja korišćen u svakom redu
3. Metod na osnovu kojeg se određuje kada proces prelazi u red višeg prioriteta
4. Metod na osnovu kojeg se određuje kada proces prelazi u red nižeg prioriteta
5. Metod na osnovu kojeg se određuje u koji red se proces smešta kada ga treba servisirati

Ovakva definicija obezbeđuje najgeneralniji algoritam za CPU raspoređivanje, koji se može prilagoditi specifičnom dizajnu sistema. Nažalost, ovakav pristup je

takođe i veoma kompleksan, jer pronalaženje optimalnog planera zahteva odabir odgovarajućih parametara navedenih iznad.

9 Raspoređivanje niti

Na jednom od prethodnih predavanja smo detaljno pričali o nitima, a predstavili smo model u kojem smo razlikovali korisničke niti izvršavanja od kernel niti izvršavanja. Na operativnim sistemima koji podržavaju niti, kernel niti, ne procesi, se raspoređuju od strane operativnog sistema. Korisničke niti se kontrolišu od strane biblioteke za rad sa nitima, a sam kernel uopšte nije ni svestan njihovog postojanja. Da bi se izvršavala na CPU, korisnička nit mora biti mapirana na dodeljenu kernelsku nit, iako ovo mapiranje može biti i indirektno i pritom koristiti lagane procese (LWP) koje smo takođe definisali ranije. Ovde ćemo ukratko pričati o problemima raspoređivanja koje uključuje korisničke i kernel niti, a daćemo primer sa bibliotekom Pthreads.

9.1 Opseg raspoređivanja (eng. *Contention scope*)

Jedna od razlika korisničkih i kernel niti leži u načinu na koji se one raspoređuju. Na sistemima koji koriste „više na jedan“ ili „više na više“ model, biblioteka za rad sa nitima raspoređuje korisničke niti da se izvršavaju na dostupnom LWP (laganom procesu). Ovaj pristup je poznat kao *opseg raspoređivanja procesa* (eng. *Process Contention Scope* - PCS), obzirom na to da se borba za CPU vrši među nitima koji pripadaju istom procesu. Pri tome, kada kažemo da biblioteka raspoređuje korisničke niti na dostupne LWP, to ne znači da se te niti izvršavaju na fizičkom procesoru. Da bi došlo do toga, neophodno je da kernel rasporedi odgovarajuću kernel nit na fizički procesor. Da bi odredio koju kernel nit da rasporedi na CPU, kernel koristi *opseg raspoređivanja sistema* (eng. System Contention Scope - SCS). Borba za CPU vreme korišćenjem SCS raspoređivanja dešava se između svih kenel niti na nivou sistema (za razliku od PCS gde se raspoređuju niti datog procesa). Sistemi koji koriste model „jedan na jedan“, kao na primer, Windows, Linux i Solaris, raspoređuju niti isključivo korišćenjem SCS.

Tipično PCS se vrši u skladu sa prioritetima - planer odabira nit koja će se izvršavati tako što traži nit sa najvišim prioritetom. Prioriteti korisničkih niti su postavljeni od strane programera i bez modifikacije se koriste od strane biblioteke za rad sa nitima, iako neke biblioteke omogućavaju programeru da menja prioritete dodeljene korisničkim nitima. Važno je primetiti da će PCS prekinuti nit koja se izvršava kako bi omogućio izvršavanje niti sa višim prioritetom, ali ne postoji garancija da će niti istog prioriteta dobijati ravnomerno CPU vreme za izvršavanje.

9.2 Raspoređivanje u Pthread biblioteci

Ovde ćemo prikazati POSIX Pthread API koji omogućava specificiranje PCS ili SCS prilikom kreiranja niti. Pthread biblioteka prepoznaje sledeće vrednosti:

- PTHREAD_SCOPE_PROCESS koji definiše da raspoređivanje koristi PCS
- PTHREAD_SCOPE_SYSTEM koji definiše da raspoređivanje koristi SCS

Na sistemima koji implementiraju model „više na više" PTHREAD_SCOPE_PROCESS vodi ka tome da se korisničke niti raspoređuju na dostupne LWP. Određeni broj LWP virtualnih procesora je održavan od strane Pthread biblioteke (važi i za druge, ali ovde razmatramo Pthread) na primer korišćenjem *aktivacije planera*. Sa druge strane, PTHREAD_SCOPE_SYSTEM će kreirati LWP i povezati ga sa svakom pojedinačnom korisničkom niti. Na sistemima koji koriste model „jedan na jedan" ovo je jedini dozvoljeni način raspoređivanja korisničkih niti korišćenjem Pthread biblioteke (npr. Linux), dok na sistemima koji koriste model „više na više", ovakvo raspoređivanje daje efekat mapiranja niti po modelu „jedan na jedan". Pthread biblioteka obezbeđuje dve funkcije za čitanje i postavljanje opsega kolizije:

- pthread_attr_setscope(pthread_attr_t *attr, int scope)
- pthread_attr_getscope(pthread_attr_t *attr, int *scope)

Prvi parametar u obe funkcije je pokazivač na atribute koji se koriste prilikom kreiranja niti. Drugi parametar u prvoj funkciji je ili PTHREAD_SCOPE_PROCESS ili PTHREAD_SCOPE_SYSTEM i određuje na koji način će se postaviti opseg kolizije. U drugoj funkciji, drugi parametar će biti postavljen na trenutno podešeni opseg kolizije od strane funkcije. Ako dođe do greške, obe funkcije vraćaju ne-nultu vrednost.

10 Raspoređivanje na multi-procesorskim sistemima

Dosadašnja diskusija se odnosila na problem raspoređivanja procesora u sistemu sa jednim procesorom. Ako imamo više dostupnih procesora *raspodela opterećenja* (eng. *load sharing*) postaje moguća, ali samim tim i samo raspoređivanje postaje znatno složenije. Brojne mogućnosti postoje, ali, kao što smo videli i u slučaju raspoređivanja sa jednim procesorom, ne postoji najbolje rešenje.

U ovom odeljku ćemo diskutovati neke od problema koji postoje kod raspoređivanja u multi-procesorskom okruženju. Pri tome se fokusiramo na *homogene sisteme* - sisteme kod kojih su svi procesori identični u pogledu njihove funkcionalnosti. U tom smislu, možemo koristiti bilo koji dostupan procesor kako bismo izvršavali bilo koji proces iz reda čekanja. Ipak, treba imati u vidu da, čak i sa

homogenim multi-procesorskim sistemima, postoje uvek određena ograničenja u vezi sa raspoređivanjem (npr. U/I uređaj povezan na privatnu magistralu jednog od procesora zahtevaju da procesi koji koriste taj uređaj budu raspoređeni na tom procesoru).

10.1 Pristupi kod raspoređivanja u multi-procesorskim sistemima

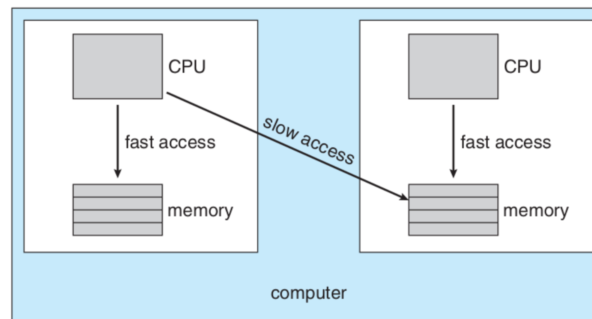
Jedan pristup kod ovakvog raspoređivanja jeste da se sve aktivnosti u vezi sa raspoređivanjem, procesiranje U/I zahteva, kao i sve ostale sistemske aktivnosti vrše na jednom procesoru koji se naziva *master server*. Ostali procesori tada isključivo služe da se na njima pokreće korisnički kod, a ovakav pristup se naziva *asimetrično multiprocesiranje*. Asimetrično multiprocesiranje je jednostavno jer samo jedan procesor pristupa sistemskim strukturama podataka, što eliminiše potrebu za deljenjem podataka.

Drugi pristup koristi *simetrično multiprocesiranje* (SMP), gde se na svakom procesoru vrši zasebno raspoređivanje. Pri tome, svi procesi mogu biti u zajedničkom redu čekanja spremnih procesa, ili svaki procesor može imati svoj privatan red čekanja spremnih procesa. Nezavisno od toga, raspoređivanje se vrši tako što planer, za svaki procesor, proverava red spremnih procesa i odabira proces koji će se izvršavati. Kao što smo videli ranije, ukoliko više procesora pristupa i ažurira zajedničke strukture podataka, planer mora biti programiran veoma pažljivo. Moramo obezbediti da dva nezavisna procesora nikako ne odaberu isti proces za izvršavanje, kao i da se procesi ne izgube iz reda čekanja. Svi moderni operativni sistemi podržavaju SMP, uključujući Windows, Linux i Mac OS X, tako da ćemo u nastavku ovog poglavlja analizirati samo SMP pristup.

10.2 Afinitet procesora

Razmotrimo šta se dešava sa keš memorijom (o keš memoriji ćemo pričati mnogo detaljnije na narednim predavanjima) kada imamo proces koji se izvršavao na određenom procesoru. Podaci kojima je proces najskorije pristupao su smešteni u keš memoriji tog procesora. Kao rezultat, sukcesivni pristupi memoriji od strane tog procesa uglavnom uvek rezultuju pristupom keš memoriji. Sada razmotrimo šta se dešava kada taj proces bude premešten na drugi procesor. Tada, sadržaj keš memorije mora biti označen nevalidnim za prvi procesor, a keš memorija drugog procesora mora biti ažurirana. Obzirom na to da su ove dve operacije izuzetno „skupe", većina SMP sistema se trudi da izbegne migraciju procesa sa jednog procesora na drugi, već se trude da nastave da izvršavaju proces na istom procesoru. Ova osobina je poznata kao *afinitet procesora* - jer proces ima afinitet prema procesoru na kojem se izvršavao do tada.

Afinitet procesora ima nekoliko oblika. Kada operativni sistem pokušava da zadrži proces da se izvršava na istom procesoru, ali bez garantovanja da će se to desiti, imamo situaciju poznatu kao *uslovni afinitet* (eng. soft affinity). Kod njega, operativni sistem će pokušati da drži proces na jednom procesoru, ali moguće je da će proces ipak biti premešten na drugi procesor. Suprotno tome,



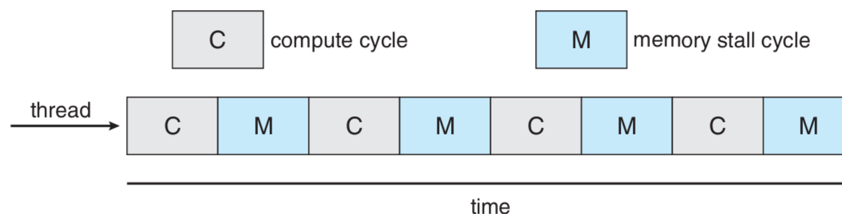
Slika 14: Ne-uniformni pristup memoriji

neki sistemi obezbeđuju sistemski poziv koji poržava *bezuslovni afinitet* (eng. hard affinity), pri čemu se procesu dozvoljava da specificira podskup procesora na kojima može da se izvršava. Većina sistema podržava i jedan i drugi pristup. Na primer, Linux implementira uslovni afinitet, ali takođe pruža mogućnost korišćenja sched_setaffinity() sistemskog poziva, koji se koristi za bezuslovni afinitet.

Arhitektura osnovne memorije u sistemu može da utiče na afinitet procesora, takođe. Na slici 14 prikazana je arhitektura ne-uniformnog pristupa memoriji (*Non-Uniform Memory Access*, NUMA), u kojoj CPU brže pristupa određenim delovima memorije nego drugim. Tipično, ovo se dešava u sistemima koji sadrže kombinovane CPU i memorijske ploče. Procesori na datoj ploči će pristupati memoriji na istoj ploči brže nego što mogu pristupati memoriji na drugim pločama u sistemu. Ukoliko CPU planer operativnog sistema i sistem za upravljanje memorijom rade zajedno, procesu koji koristi određeni CPU će biti alocirana memorija koja se nalazi na istoj ploči.

10.3 Balansiranje opterećenja (*Load Balancing*)

Na SMP sistemima, veoma je važno da se opterećenje ravnomerno raspodeljuje između svih procesora, kako bi se u potpunosti iskoristile prednosti postojanja više procesora u sistemu. U suprotnom, jedan ili više procesora će biti besposleni, dok su drugi preopterećeni. Balansiranjem opterećenja ravnomerno se raspodeljuje posao koji treba da se obavi, između svih procesora koji postoje u SMP sistemu. Važno je napomenuti da je balansiranje opterećenja tipično neophodno samo na sistemima gde svaki procesor ima svoj privatni red čekanja spremnih procesa. Na sistemima gde postoji zajednički red čekanja spremnih procesa, balansiranje opterećenja je nepotrebno, pošto čim procesor završi sa izvršavanjem prethodnog procesa, odmah će preuzeti sledeći spreman proces iz zajedničkog reda. Ipak, jednako je važno pomenuti da kod većine savremenih operativnih sistema koji podržavaju SMP, svaki procesor dobija svoj privatan red čekanja spremnih procesa.



Slika 15: Memorijski zastoj

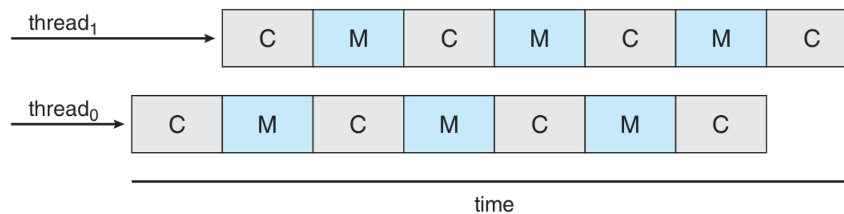
Postoje dva pristupa kod balansiranja opterećenja: migracija predajom (eng. *push migration*) i migracija preuzimanjem (eng. *pull migration*). Kod migracije predajom, specifičan proces periodično proverava opterećenje na svakom od procesora i, ako nađe odgovarajući disbalans, ravnomerno raspoređuje opterećenje tako što premešta procese sa preopterećenog procesora ka manje opterećenim procesorima. Migracija preuzimanjem se javlja kada neiskorišćen procesor prezima procese od zauzetog procesora. Ova dva pristupa ne moraju nužno biti uzajamno isključiva - zapravo su uglavnom implementirani u paraleli na sistemima kod kojih se vrši balansiranje opterećenja. Na primer, Linux-ov planer (o kome ćemo uskoro više pričati) implementira obe tehnike.

Interesantno je primetiti da balansiranje opterećenja često poništava benefite dobijene korišćenjem afiniteta procesora. Cilj zadržavanja procesa na datom procesoru jeste da se iskoristi činjenica da se podaci koje proces koristi nalaze u keš memoriji procesora, a bilo kakvo balansiranje opterećenja eliminiše ovaj benefit. U skladu sa tim, na nekim sistemima neiskorišćen procesor će uvek preuzeti zadatke od preopterećenog, dok se na drugim sistemima to dešava samo ako je disbalans prešao određeni prag tolerancije.

10.4 Višejezgarni procesori

Tradicionalno, SMP sistemi dozvoljavaju paralelno izvršavanje više niti obezbeđujući više fizičkih procesora. Ipak, savremena tehnologija omogućava integraciju višestrukih procesorskih jezgara na jednom fizičkom čipu, čime nastaju višejezgarni procesori. Svako jezgro ima svoju arhitekturu i stoga se od strane operativnog sistema prepoznaje kao zaseban procesor. SMP sistemi koji koriste višejezgarne procesore su brži i troše manje energije u poređenju sa sistemima kod kojih je svaki procesor na zasebnom čipu.

Sa druge strane, višejezgarni procesori mogu zakomplikovati raspoređivanje. Istraživanja su ustanovila da kada procesor pristupa memoriji, značajan deo vremena se troši na čekanje na podatke da postanu dostupni. Ova situacija je poznata kao *memorijski zastoj* (eng. *memory stall*). Memorijski zastoj može da se desi usled više uzroka, na primer kao posledica pristupa podacima koji se ne nalaze u keš memoriji (*cash miss*). Slika 15 prikazuje memorijski zastoj. U ovom scenariju, procesor će potrošiti do 50% svog vremena čekajući na podatke iz memorije da budu dostupni. Da bi se rešio ovaj problem, većina savremenih

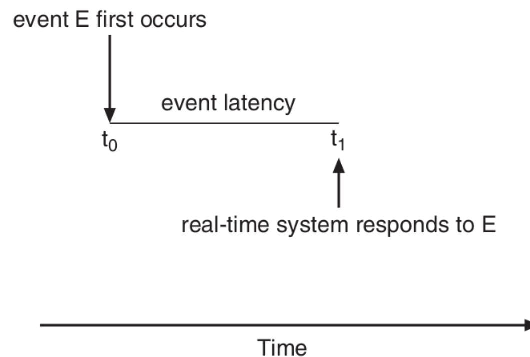


Slika 16: Dve hardverske niti i memorijski zastoj

dizajna hardvera implementira procesore sa više hardverskih niti - tzv *višenitne* procesore (*multithreaded processors*). Kod njih, dve (ili više) hardverskih niti postoje u okviru svakog jezgra. Takva struktura omogućava da jezgro koristi jednu hardversku nit, dok druga nit čeka na memoriju. Slika 16 prikazuje procesorsko jezgro sa dve hardverske niti (*dual-threaded procesor core*) na kome se kombinuju i prepliću izvršavanja na niti 0 i niti 1. Sa stanovišta operativnog sistema, svaka hardverska nit je prepoznata kao zaseban logički procesor koji se može koristiti za izvršavanje softverskih niti. Dakle, na dvojezgarnom sistemu sa dvostrukim hardverskim nitima (*dual-threaded dual-core system*) četiri logička procesora se pojavljuju u operativnom sistemu.

Generalno, postoje dva tipa višenitnih procesorskih jezgara. To su grubo struktuirana (eng. *coarse-grained*) i fino struktuirana (eng. *fine-grained*) višenitna jezgra. Kod grubo struktuiranih jezgara, nit se izvršava na procesoru sve dok ne dođe do događaja koji zahteva čekanje (npr. memorijskog zastoja). Kada dođe do toga, procesor prenosi izvršavanje na drugu hardversku nit kako bi nastavio izvršavanje. Međutim, cena zamene hardverskih niti je visoka, obzirom na činjenicu da blok za protočnu obradu instrukcija (eng. *instruction pipeline*) mora biti ispražnjen pre nego što druga nit krene sa svojim izvršavanjem i ponovo popunjen kako nova nit krene da se izvršava. Fino struktuirana jezgra (često se zovu i isprepletena - eng. *interleaved*) vrše zamenu sa mnogo finijom granularnošću, tipično na nivou instrukcijskih ciklusa. Da bi ovo bilo moguće, arhitektura fino struktuiranih višenitnih jezgara uključuje i logiku za zamenu niti, što za posledicu daje izuzetno „malu cenu“ zamene niti u terminima vremena i resursa.

Neophodno je primetiti da kod višenitnih višejezgarnih procesora imamo zapravo dva nivoa raspoređivanja. Na prvom nivou donose se odluke od strane operativnog sistema koji odabira koja softverska nit će se izvršavati na kojoj hardverskoj niti odnosno na kojem logičkom procesoru. U ovu svrhu, bilo koji algoritam raspoređivanja opisan ranije može da se koristi. Na drugom nivou, raspoređivanje specificira na koji način svako od jezgara odlučuju koju hardversku nit će da koristi. Postoji nekoliko strategija koje se koriste. UltraSPARC T3, koji ima 16 jezgara na čipu i 8 niti po jezgru, koristi RR algoritam kako bi raspoređivao svojih osam hardverskih niti. Sa druge strane, Intel-ov Itanium, koji je procesor sa dva jezgra sa dve niti po jezgru, svakoj hardverskoj niti dodeljuje



Slika 17: Latentnost događaja

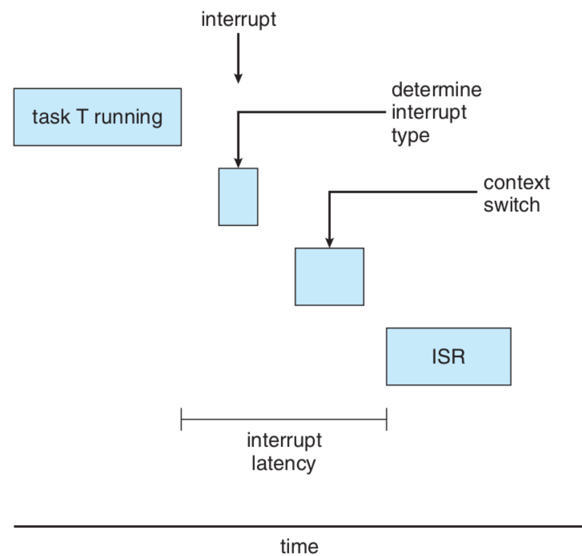
dinamičku *urgentnost* (eng. *urgency*) u opsegu 0-7 gde 0 predstavlja najnižu, a 7 najvišu. Itanium definiše pet različitih događaja koji mogu da izazovu zamenu niti, a kada se neki od njih desi, logika zadužena za zamenu niti proverava urgentnost niti, odabira onu nit koja ima najvišu urgentnost i nju izvršava na procesorskom jezgru.

11 CPU raspoređivanje u realnom vremenu (*Real-time scheduling*)

Raspoređivanje CPU-a za operativne sisteme u realnom vremenu donosi specifične probleme. Generalno, možemo razlikovati tolerantne sisteme u realnom vremenu (*soft real-time systems*) i netolerantne sisteme u realnom vremenu (*hard real-time systems*). Tolerantni sistem u realnom vremenu ne daje garanciju kada će se izvršavati kritični proces. Oni garantuju samo da će proces imati prednost nad nekritičnim procesima. Netolerantni sistemi u realnom vremenu imaju znatno strožije zahteve. Zadatak se mora obraditi pre krajnjeg roka; završetak obrade nakon isteka roka isti je kao i neizvršeni zadatak. U ovom odeljku diskutujemo nekoliko problema koji se odnose na raspoređivanje procesa u tolerantnim i netolerantnim operativnim sistemima u realnom vremenu.

11.1 Minimizacija kašnjenja

Razmatraćemo prirodu sistema u realnom vremenu koji je upravljan događajima (*event driven*). Sistem obično čeka da se dogodi događaj u realnom vremenu. Događaji se mogu pojaviti ili u softveru (istekao tajmer), ili u hardveru (kada vozilo sa daljinskim upravljanjem otkrije da se približava prepreci). Kada se događaj dogodi, sistem mora reagovati i servisirati ga što je brže moguće. Terminom *latentnost događaja* nazivamo vreme koji protekne od trenutka pojave događaja do trenutka kada je on servisiran (slika 17).



Slika 18: Latentnost prekida

Obično različiti događaji imaju različite zahteve u vezi kašnjenja. Na primer, zahtev za kašnjenje za ABS (*Antilock Braking System*) može biti 3 do 5 milisekundi. Odnosno, od trenutka kada točak detektuje da je blokiran i da klizi, sistem koji upravlja kočnicom ima 3 do 5 milisekundi da reaguje i kontroliše situaciju kroz brze kratkotrajne sekvence kočenja i otpuštanja kočnice. Bilo koja reakcija koji traje duže može dovesti do toga da automobil ostane van kontrole. Suprotno tome, embeded sistem za kontrolu radara u avionu može da toleriše vremenski period od nekoliko sekundi.

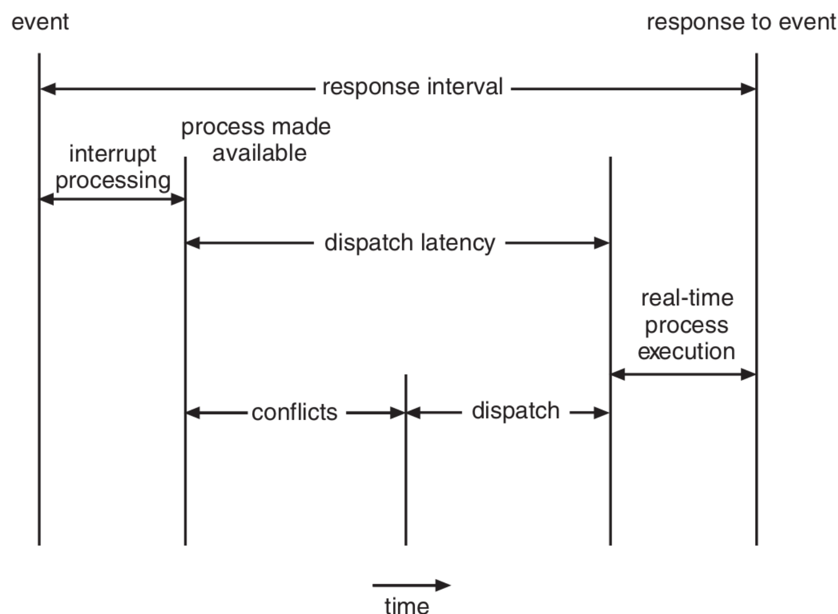
Postoje dva tipa kašnjenja kojie utiču na performanse sistema u realnom vremenu:

1. Latentnost prekida (*interrupt latency*)

2. Latentnost dispečera (*dispatcher latency*)

Latentnost prekida odnosi se na vremenski period od pristizanja zahteva za prekidom CPU-u do početka rutine za obradu prekida. Kada dođe do prekida, operativni sistem prvo mora dovršiti instrukciju koju izvršava i odrediti vrstu prekida koji se dogodio. Zatim mora da sačuva stanje trenutnog procesa pre servisiranja prekida korišćenjem odgovarajuće prekidne rutine (ISR). Ukupno vreme potrebno za obavljanje ovih zadataka predstavlja latentnost prekida (slika 18).

Očigledno je da je za operativne sisteme u realnom vremenu ključno da smanje latentnost prekida kako bi se osiguralo da zadaci u realnom vremenu dobiju momentalnu pažnju. Dodatno, kod netolerantnih sistema u realnom



Slika 19: Latentnost dispečera

vremenu, nije dovoljno svesti latentnost prekida na minimum, već ona mora biti vremenski ograničena kako bi bili ispunjeni strogi zahtevi u pogledu rokova kod ovih sistema.

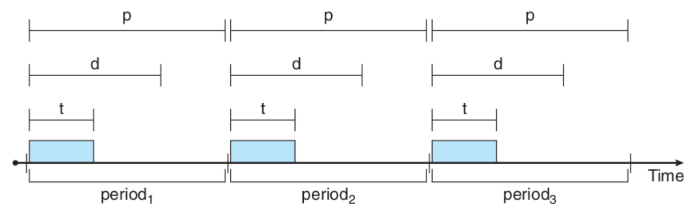
Jedan veoma važan faktor koji doprinosi latentnosti prekida je količina vremena tokom kojeg prekidi koji mogu biti onemogućeni kada se ažuriraju strukture podataka kernela. Operativni sistemi u realnom vremenu zahtevaju da se prekidi, u ovom slučaju, onemoguće samo tokom vrlo kratkog vremena.

Sa druge strane, vreme potrebno dispečeru da zaustavi jedan proces i započne izvršavanje drugog poznato je kao *latentnost dispečera*. Omogućavanje neposrednog pristupa CPU-u zadacima u realnom vremenu nalaže da operativni sistemi u realnom vremenu minimiziraju i ovu vrstu kašnjenja. Najefikasnija tehnika za smanjivanje latentnosti dispečera jeste korišćenje prisvojivih kernela.

Na slici 19 prikazani su uzroci latentnosti dispečera. *Faza konflikta* kod latentnosti dispečera ima dve komponente:

1. Prekidanje procesa koji se izvršava
2. Oslobađanje resursa potrebnih procesima visokog prioriteta od strane procesa niskog prioriteta

Kao primer, u operativnom sistemu Solaris latentnost dispečera sa zabranjenim prisvajanjem je preko sto milisekundi. Ako je omogućeno prisvajanje, ona se smanjuje na manje od milisekunde.



Slika 20: Periodični proces sa svojim karakteristikama

11.2 Raspoređivanje po prioritetima

Najvažnija karakteristika operativnog sistema u realnom vremenu je da odmah reaguje na proces, čim taj proces zahteva CPU. Kao rezultat toga, planer operativnog sistema u realnom vremenu mora da koristi algoritam zasnovan na prioritetima. Podsetimo se da algoritmi za raspoređivanje bazirani na prioritetima svakom procesu dodeljuju prioritet na osnovu njegove važnosti: važnijim zadacima su dodeljeni viši prioriteti od onih koji se smatraju manje važnim. Ako planer takođe koristi prisvajanje, proces koji trenutno radi na CPU-u će biti prekinut ako proces s višim prioritetom postane dostupan za izvršavanje.

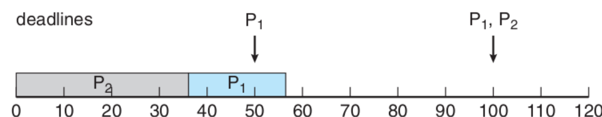
Prisvojivi algoritmi za raspoređivanje zasnovani na prioritetima detaljno su opisani prethodno. Što se tiče prioriteta u najvažnijim operativnim sistemima, Windovs ima 32 različita nivoa prioriteta. Najviši nivoi, vrednosti prioriteta 16 do 31, rezervisani su za procese u realnom vremenu. Solaris i Linux imaju slične šeme prioriteta.

Imajte na umu da prisvajanje planirano na osnovu prioriteta garantuje samo tolerantnu funkcionalnost u realnom vremenu. Netolerantni sistemi u realnom vremenu moraju dodatno garantovati da će se zadaci u realnom vremenu servisirati u skladu sa njihovim krajnjim rokovima, a takve garancija zahtevaju dodatne osobine raspoređivanja. U ostatku ovog odeljka pričaćemo o algoritmima planiranja koji su pogodni za netolerantne sisteme u realnom vremenu.

Pre nego što nastavimo sa detaljima vezanih za pojedine planere, međutim, moramo definisati određene karakteristike procesa koji treba rasporediti. Prvo, procesi se u netolerantnim operativnim sistemima za rad u realnom vremenu smatraju periodičnim. Drugim rečima, oni zahtevaju CPU u konstantnim intervalima vremena (periodima). Jednom kada periodični proces dobije CPU, on ima određeno vreme obrade t, rok d do kada najkasnije CPU mora da servisira proces i periodu izvršavanja p. Odnos vremena obrade, krajnjeg roka i periode može se izraziti kao $0 \leq t \leq d \leq p$. Učestanost periodičnog procesa je $1 \;/\; p$.

Slika 20 ilustruje izvršavanje periodičnog procesa tokom vremena. Planeri mogu iskoristiti ove karakteristike i dodeliti prioritete u skladu sa krajnjim rokom ili zahtevima u vezi sa učestanosti procesa.

Ono što je neobično u ovom obliku raspoređivanja je svakako to da proces mora da informiše planer u vezi sa svojim krajnjim rokovima. Zatim, koristeći tehniku poznatu kao *algoritam kontrole prijema* (eng. *admission-control*



Slika 21: Raspoređivanje sa monotonom stopom gde je procesu P_2dat viši prioritet

algorithm), planer izvršava jedan od dva moguća scenarija. Ili prihvata proces, garantujući da će se proces završiti na vreme, ili odbacuje zahtev kao nemoguć ako ne može garantovati da će proces biti servisiran do isteka njegovog krajnjeg roka.

11.3 Raspoređivanje sa monotonom stopom (*Rate-Monotonic Scheduling*)

Ovaj algoritam raspoređuje periodične zadatke koristeći statičke prioritete uz prisvajanje. Ako je pokrenut proces sa nižim prioritetom i proces sa višim prioritetom postane spreman za pokretanje, on će se izvršavati umesto procesa nižeg prioriteta. Prilikom ulaska u sistem, svakom periodičnom zadatku se dodeljuje prioritet, obrnuto srazmeran njegovoj periodi izvršavanja. Što je kraća perioda, veći je prioritet, a što je ona duža, to je prioritet niži. Razlog ove politike je dodeljivanje većeg prioriteta zadacima koji češće zahtevaju CPU. Pored toga, raspoređivanje sa monotonom stopom podrazumeva da je vreme obrade periodičnog procesa isto za svaki CPU ciklus. Drugim rečima, svaki put kada proces dobije CPU na korišćenje, trajanje njegovog CPU ciklusa je isto.

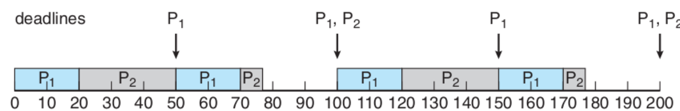
Razmotrimo primer u kojem imamo dva procesa, P_1 i P_2. Periode za P_1 i P_2 su 50 i 100, odnosno, $p_1 = 50$ i $p_2 = 100$. Vremena obrade su $t_1 = 20$ za P_1 i $t_2 = 35$ za P_2. Krajnji rok za svaki proces zahteva da se završi trenutni CPU ciklus do početka sledećeg perioda.

Prvo se mora ustanoviti da li je uopšte moguće rasporediti ove zadatke tako da svaki ispuni svoje krajnje rokove. Ako izmerimo iskorišćenost procesora za proces P_i kao odnos njegovog ciklusa i njegove periode t_i / p_i - uskorišćenost CPU od strane P_1 je $20/50 = 0{,}40$, a od strane P_2 je $35/100 = 0{,}35$, pa je ukupna iskorišćenost procesora 75%. Kao rezultat, čini se da ove zadatke možemo rasporediti na način da oba ispune svoje krajnje rokove i dodatno ostave CPU besposlenim.

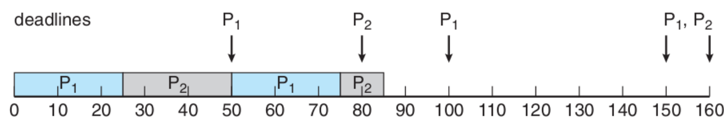
Pretpostavimo da procesu P_2 dodelimo veći prioritet od P_1. Izvršenje P_1 i P_2 u ovoj situaciji prikazano je na slici 21.

Kao što vidimo, P_2 započinje izvršenje prvi i završava se u trenutku 35. U ovom trenutku P_1 započinje i dovršava svoj CPU ciklus u trenutku 55. Međutim, prvi krajnji rok za P_1 bio je u trenutku 50, pa je planer doveo do toga da P_1 propusti svoj krajnji rok.

Pretpostavimo sada da koristimo raspoređivanje sa monotonom stopom, u kojem dodelimo procesu P_1 viši prioritet od P_2, jer je perioda proces P_1 kraća



Slika 22: Raspoređivanje sa monotonom stopom gde je procesu P_1dat viši prioritet



Slika 23: Raspoređivanje sa monotonom stopom gde nije zadovoljen krajnji rok

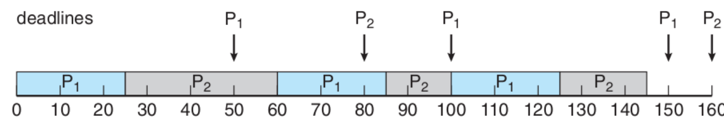
od periode procesa P_2. Izvršavanje procesa u ovoj situaciji prikazano je na slici 22. P_1 započinje prvi i završava svoj CPU ciklus u trenutku 20, čime ispunjava svoj prvi krajnji rok. P_2 počinje da se pokreće u ovom trenutku i izvršava se do trenutka 50. U ovom trenutku ga zamenjuje P_1, mada P_2 još uvek ima 5 milisekundi u svom CPU ciklusu. P_1 dovršava svoj drugi CPU ciklus u trenutku 70, a zatim planer nastavlja P_2, koji dovršava svoj CPU ciklus trenutku 75, takođe ispunjavajući svoj prvi krajnji rok. Sistem je u stanju mirovanja do trenutka 100ms, kada se ponovo pokreće P_1.

Raspoređivanje sa monotonom stopom smatra se optimalnim u smislu da ako se dati skup procesa ne može rasporediti ovim algoritmom, ne može se rasporediti ni bilo kojim drugim algoritmom koji koristi statičke prioritete. Posmatrajmo sada skup procesa koji se ne mogu rasporediti koristeći algoritam raspoređivanja sa monotonom stopom. Pretpostavimo da proces P_1 ima period od $p_1 = 50$, a CPU ciklus $t_1 = 25$. Za P_2, odgovarajuće vrednosti su $p_2 = 80$ i $t_2 = 35$. Raspoređivanje sa monotonom stopom bi dodelilo procesu P_1 viši prioritet, jer ima kraću periodu. Ukupna iskorišćenost CPU-a u oba procesa je $(25/50) + (35/80) = 94\%$, i zato se čini logičnim da bi dva procesa mogla biti raspoređena i takođe ostaviti CPU sa 6 procenata slobodnog vremena. Na slici 23 je prikazano raspoređivanje procesa P_1 i P_2.

U početku se P_1 pokreće dok ne dovrši CPU ciklus u trenutku 25ms. Proces P_2 se tada pokreće i izvršava sve do trenutka 50ms, kada ga P_1 zamenjuje. U ovom trenutku, P_2 procesu još uvek ostaje 10 milisekundi u CPU ciklusu. Proces P_1 traje do trenutka 75, što znači da P_2 propušta krajnji rok za završetak svog CPU-a ciklusa u trenutku 80ms. Iako je optimalno, raspoređivanje sa monotonom stopom ima svoje limitacije: CPU iskorišćenost je ograničena, što znači da nije uvek moguće u potpunosti maksimizirati iskorišćenost CPU-a. Najgori slučaj za iskorišćenost CPU-a prilikom raspoređivanja N procesa je

$$N\left(2^{\frac{1}{N}} - 1\right)$$

Sa jednim procesom u sistemu, maksimala iskorištenost CPU-a je 100 posto,



Slika 24: EDF raspoređivanje

ali pada na otprilike 69% kako se broj procesa približava beskonačnosti. Sa dva procesa, maksimalna iskorišćenost procesora ograničena je na oko 83%. Kombinovana iskorišćenost procesora za dva procesa prikazana slikama 21 i s22 je 75%, pa je algoritam zakazivanja sa monotonom stopom garantovao uspešno raspoređivanje dva procesa tako da se mogu ispuniti njihovi krajnji rokovi. Za dva procesa u primeru prikazanom na slici 23, kombinovana iskorišćenost CPU-a je približno 94%, te raspoređivanje monotonom stopom ne može garantovati njihovo raspoređivanje tako da budu ispunjeni njihove krajnji rokovi.

11.4 EDF (*Earlist-Deadline First*) raspoređivanje

Ovaj algoritam dinamički dodeljuje prioritete u skladu sa krajnjim rokovima. Što je krajnji rok skoriji, to je prioritet procesa viši, a što je kasniji, prioritet procesa je niži. Uz ovakvo raspoređivanje, kada proces postane spreman za izvršavanje, mora javiti sistemu informaciju u vezi sa njegovim krajnjim rokom. Prioriteti se tada, eventualno, moraju korigovati, kako bi se uzeo u obzir krajnji rok novopristiglog procesa. U ovome je i glavna razlika EDF algoritma u poređenju sa prethodno opisanim gde su prioriteti bili nepromenljivi u sistemu.

Da ilustrujemo raspoređivanje korišćenjem EDF algoritma, ponovo raspoređujemo procese prikazane na slici 23, koji nismo uspeli da rasporedimo i da ispunimo zahteve krajnjeg roka korišćenjem raspoređivanja sa monotonom stopom. Podsetimo da P_1 ima karakteristike $p_1 = 50$ i $t_1 = 25$ i da P_2ima karakteristike $p_2 = 80$ i $t_2 = 35$. EDF raspoređivanje ovih procesa prikazano je na slici 24.

Proces P_1 ima najraniji krajnji rok, tako da je njegov početni prioritet veći od prioriteta procesa P_2. Proces P_2 počinje da se izvršava nakon CPU ciklusa procesa P_1. Međutim, dok raspoređivanje sa monotonom stopom omogućava da P_1 prekine P_2 na početku svog sledećeg perioda u trenutku 50, EDF omogućava procesu P_2 da nastavi sa izvršavanjem. P_2 u trenutku 50ms ima veći prioritet od P_1, jer je njegov naredni krajnji rok (u trenutku 80ms) raniji od roka P_1 (u trenutku 100ms). Dakle, i P_1 i P_2 ispunjavaju svoje prve krajnje rokove. Proces P_1 ponovo počinje da se izvršava u trenutku 60ms i završava na vreme i svoj drugi CPU ciklus u trenutku 85, takođe ispunjavajući svoj drugi krajnji rok (100ms). P_2 počinje da se izvršava u ovom trenutku, ali se izvršava samo do trenutka 100ms da bi P_1 počeo izvršavanje svog sledećeg perioda. P_2 se prekida jer P_1 ima raniji krajnji rok (trenutak 150ms) od P_2 (trenutak 160ms). U trenutku 125ms, P1 dovršava svoj drugi CPU ciklus, a P2 nastavlja sa izvršavanjem, završavajući u trenutku 145ms i ponovo zadovoljavajući svoj krajnji rok. Sistem je u stanju mirovanja do trenutka 150ms, kada se ponovo pokreće

P_1.

Za razliku od algoritma raspoređivanja sa monotonom stopom, za EDF raspoređivanje nije neophodno da procesi budu periodični. Dodatno, kod EDF nije neophodno da proces zahteva konstantnu količinu vremena za svaki CPU ciklus. Jedini zahtev je da proces najavi svoj krajnji rok planeru kada postane spreman za izvršavanje. Atraktivnost EDF algoritma ogleda su u tome što je on teoretski optimalan. Teoretski, EDF može rasporediti procese na takav način da svaki proces može ispuniti svoj krajnji rok, a da pri tome iskorišćenost CPU-a bude 100%. U praksi je, međutim, nemoguće postići ovaj nivo iskorišćenosti procesora usled trajanja zamene konteksta i rutina za obradu prekida.

11.5 Posix real-time raspoređivanje

POSIX standard obezbeđuje ekstenziju za procesiranje u realnom vremenu (POSIX.1b). Ovde ćemo prikazati POSIX API koji se odnosi na raspoređivanje niti u realnom vremenu. POSIX definiše dve klase raspoređivanja za real-time niti:

- SCHED_FIFO
- SCHED_RR

SCHED_FIFO raspoređuje niti u maniru FCFS pri čemu koristi FIFO red. Međutim, ako se koristi ovakvo raspoređivanje, niti istog prioriteta neće dobijati jednake intervale CPU vremena. U skladu sa tim, real-time nit sa najvišim prioritetom na početku reda čekanja spremnih procesa će dobiti CPU na korišćenje sve dok se ne terminira ili blokira. SCHED_RR koristi round-robin metod raspoređivanja, koji je sličan sa SCHED_FIFO sa izuzetkom da dodatno uvodi vremenski kvant i ravnomernu raspodelu vremena između niti istog prioriteta. POSIX takođe obezbeđuje još jednu klasu SCHED_OTHER, ali njena implementacija je nedefinisana i na različitim sistemima može biti drugačija.

POSIX API specificira dve funkcije za čitanje i postavljanje željenog metoda raspoređivanja:

- pthread_attr_getsched_policy(pthread_attr_t *attr, int *policy)
- pthread_attr_setsched_policy(pthread_attr_t *attr, int policy)

Prvi parametar u obe funkcije je pokazivač na skup atributa za datu nit. Drugi parametar je ili pokazivač na celobrojnu vrednost u slučaju prve funkcije koja kroz taj podatak vraća trenuti metod raspoređivanja, ili samo celobrojna vrednost (SCHED_FIFO, SCHED_RR, SCHED_OTHER) koja postavlja željeni način raspoređivanja u slučaju druge funkcije. Ove funkcije vraćaju ne-nulte vrednosti u slučaju greške.

12 Raspoređivanje procesora u Linux operativnom sistemu

Raspoređivanje procesa u Linux operativnom sistemu je imalo zanimljivu istoriju. Pre verzije kernel 2.5, Linux kernel je raspoređivao procese korišćenjem varijacije tradicionalnog algoritma za raspoređivanje Unix-a. Međutim, kako ovaj algoritam nije dizajniran sa SMP sistemima na umu, nije adekvatno podržavao sisteme sa više procesora. Dodatno, ovakvo raspoređivanje je rezultovalo lošim performansama kod sisteme sa velikim brojem procesa koji se izvršavaju.

Sa verzijom kernela 2.5, planer je modifikovan tako da koristi novi algoritam za raspoređivanje, poznat kao O(1), koji se izvršavao u konstantnom vremenu bez obzira na broj procesa u sistemu. O(1)planer je takođe pružio poboljšanu podršku za SMP sisteme, uključujući afinitet procesora i balansiranje opterećenja između procesora. Međutim, u praksi, iako je O(1) planer postigao odlične performanse na SMP sistemima, rezultovao je lošim vremenima odziva u slučaju interaktivnih procesa koji su uobičajeni za mnoge desktop računarske sisteme. Tokom razvoja kernela 2.6, planer je ponovo revidiran, a u verziji kernela 2.6.23, *potpuno fer planer* (eng. *Completely Fair Scheduler* - CFS) postao je podrazumevani algoritam za raspoređivanje u Linux operativnom sistemu.

12.1 Klase raspoređivanja i prioriteti

Raspoređivanje u Linux sistemu zasniva se na *klasama raspoređivanja*. Svakoj klasi se dodeljuje odgovarajući prioritet. Korišćenjem različitih klasa raspoređivanja, kernel podržava različite algoritme raspoređivanja na osnovu potreba sistema i njegovih procesa. Na primer, kriterijumi za raspoređivanje Linux servera mogu biti različiti od onih za mobilni uređaj na kojem se izvršava Linux. Da bi odlučio koji će se zadatak pokrenuti sledeći, planer odabira zadatak s najvišim prioritetom koji pripada klasi s najvišim prioritetom. Standardni Linux kerneli implementiraju dve klase raspoređivanja:

1. podrazumevana klasa raspoređivanja korišćenjem CFS algoritma

2. klasa raspoređivanja u realnom vremenu

U nastavku ćemo posmatrati svaku od navedenih klasa, a, naravno, nove klase raspoređivanja mogu biti dodate.

CFS planer dodeljuje procenat CPU vremena svakom zadatku. Ovaj procenat se izračunava na osnovu tzv. *lepe vrednosti* (eng. *nice value*) dodeljene svakom zadatku. Lepe vrednosti se kreću od -20 do +19, pri čemu niža lepa vrednost ukazuje na viši relativni prioritet. Zadaci sa nižim lepim vrednostima dobijaju veći procenat CPU vremena u poređenju sa zadacima koji imaju više lepe vrednosti. Podrazumevana inicijalna lepa vrednost je 0. Izraz *lepa vrednost* se koristi jer zadatak koji poveća svoju lepu vrednost sa, recimo, 0 na +10, ponaša se lepo prema drugim zadacima u sistemu smanjujući svoj relativni prioritet. CFS ne koristi diskretne vrednosti vremenskih kvanata i umesto toga

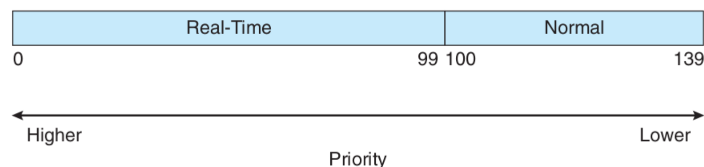
definiše tzv. željeno kašnjenje (eng. *targeted latency*). Željeno kašnjenje je interval vremena tokom kojeg bi se svaki zadatak (koji bi se mogao izvršavati) najmanje jednom pokrenuo. Proporcionalni delovi CPU vremena dodeljuju se zadacima u odnosu na željeno kašnjenje. Pored podrazumevanih i minimalnih vrednosti, željeno kašnjenje se može povećati ako broj aktivnih zadataka u sistemu naraste preko određenog praga.

CFS planer ne postavlja prioritete direktno. Umesto toga, on vodi evidenciju koliko dugo se svaki zadatak izvršavao praćenjem tzv *virtuelnog vremena izvršavanja* (eng. *virtual runtime*) svakog zadatka koristeći promenljivu *vruntime* za svaki zadatak. Virtualno vreme izvršavanja povezano je s *faktorom opadanja* (eng. *decay factor*) na osnovu prioriteta zadatka: zadaci nižeg prioriteta imaju veći faktor opadanja od zadataka višeg prioriteta. Za zadatke sa normalnim prioritetom (lepe vrednosti 0) virtuelno vreme izvršavanja identično je stvarnom fizičkom vremenu izvršavanja. Dakle, ako se zadatak sa podrazumevanim prioritetom izvršava 200 milisekundi, njegovo virtualno vreme izvršavanja će takođe biti 200 milisekundi. Međutim, ako zadatak nižeg prioriteta traje 200 milisekundi, njegovo virtualno vreme izvršavanja biće veće od 200 milisekundi. Slično tome, ako se zadatak sa višim prioritetom izvršava 200 milisekundi, njegovo virtualno vreme izvršavanja će biti manje od 200 milisekundi.

Da bi odlučio koji će se zadatak pokrenuti sledeći, planer jednostavno bira zadatak koji ima najmanju *vruntime* vrednost. Dodatno, zadatak s višim prioritetom koji postaje spreman za izvršavanje može prekinuti zadatak nižeg prioriteta.

Posmatrajmo kako CFS planer radi na primeru sa dva zadatka koji imaju jednake lepe vrednosti, a od kojih je jedan CPU-kontrolisan, a drugi U/I kontrolisan. Tipično, U/I kontrolisan zadatak će se izvršavati samo kratak period vremena pre nego što se blokira nakon U/I zahteva. Sa druge strane, CPU-kontrolisani zadatak će se izvršavati koliko god CPU vremena da mu je dodeljeno. Kao rezultat, *vruntime* vrednost će vremenom biti niža za U/I-kontrolisani zadatak, u poređenju sa CPU-kontrolisanim zadatkom. To će da dovede do toga da on ima viši prioritet u odnosu na CPU-kontrolisani zadatak: ako se CPU-kontrolisani zadatak izvršava u momentu kada U/I-kontrolisani zadatak postane spreman za izvršavanje (na primer nakon što je završen U/I zahtev koji se čekao), U/I-kontrolisani zadatak će prekinuti CPU-kontrolisani zadatak i izvršavaće se odmah.

Linux operativni sistem podržava i raspoređivanje u realnom vremenu, kao što smo naveli ranije, implementacijom POSIX standarda. Svaki zadatak koji se raspoređuje bilo korišćenjem SCHED_FIFO bilo SCHED_RR algoritma, izvršavaće se sa višim prioritetom u poređenju sa „normalnim“ zadacima (koji nisu real-time). Linux u tu svrhu koristi odvojene opsege prioriteta, jedan za real-time zadatke, a drugi za normalne zadatke. Real-time zadaci koriste statičke prioritete koji su u opsegu od 0 do 99, a normalni prioritete koji se nalaze u opsegu 100 do 139. Ova dva opsega mapiraju se u globalnu shemu prioriteta gde manja numerička vrednost predstavlja viši relativni prioritet. Normalni zadaci dobijaju prioritete u skladu sa svojim *lepim vrednostima*, pri čemu se vrednost -20 mapira na prioritet 100, a vrednost +19 na vrednost 139. Ova



Slika 25: Prioriteti raspoređivanja u Linux operativnom sistemu

shema prikazana je na slici 25.

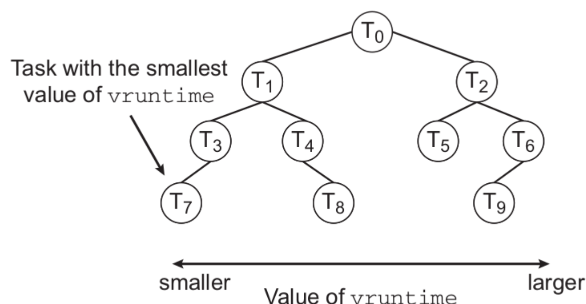
12.2 CFS implementacija

CFS planer koristi efikasan algoritam kako bi odabrao koji zadatak da se izvršava sledeći. Prvo ćemo se upoznati sa *balansiranim binarnim stablima pretrage* (eng. *Balanced binary search tree*).

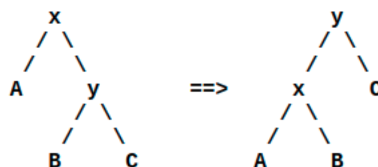
Binarno stablo pretrage je stablo kod kojeg svaki čvor ima vrednost (koja se zove i ključ), a za svaki čvor važi da su svi čvorovi sa manjim vrednostima ključa od njega u njegovom levom podstablu, a svi čvorovi sa većim vrednostima ključa u desnom. Balansirano binarno stablo pretrage je, dodatno, stablo kod kojeg za svaki čvor važi da je apsolutna vrednost razlike visina njegovih podstabala najviše 1.

CFS algoritam svaki zadatak koji je spreman za izvršavanje smešta se u *balansirano binarno stablo pretrage*, pri čemu koristi *vruntime* tog zadatka kao vrednost ključa novokreiranog čvora u binarnom stablu. Primer jednog takvog stabla može se videti na slici 26.

Kada zadatak postane spreman za izvršavanje, dodaje se u stablo pretrage. Ako zadatak iz stabla više nije spreman za izvršavanje (na primer, blokiran je dok čeka na U/I), on se uklanja iz stabla pretrage. Karakteristika binarnog stabla pretrage je takva da se na levoj strani stabla nalaze zadaci koji su dobili manje procesorskog vremena, a sa desne zadaci koji su ga dobili više. Takođe, kao posledica osobina binarnog stabla pretrage, najlevlji čvor odgovara čvoru sa



Slika 26: Balansirano binarno stablo, struktura korišćena od strane CFS planera



Slika 27: Jednostruka rotacija

najmanjom vrednosti ključa, tj. zadatku sa najmanjim *vruntime*, a za CFS to je zadatak koji ima najviši prioritet.

Pošto je binarno stablo pretrage balansirano, pronalaženje najlevljeg čvora zahteva O(ld$N$) operacija, gde je ld logaritam sa osnovom 2, a N broj čvorova u stablu. Ipak, da bi bio efikasniji, Linux planer kešira ovu vrednost u promenljivoj *rb_leftmost*, tako da se pronalaženje sledećeg zadatka za izvršavanje svodi na čitanje ove keširane vrednosti.

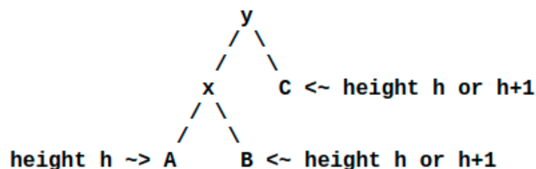
Dodavanje novog čvora u stablu pretrage bi trebalo da bude trivijalno-krene se od korena stabla, pa se ide desno ili levo u zavisnosti od vrednosti ključa novog čvora i ključa korena stabla, a onda se ta operacija ponavlja i za svaki sledeći čvor koji se obiđe, sve dok se ne dođe do pozicije na koju treba dodati novi čvor. Međutim, nakon dodavanja novog čvora, potrebno je proveriti da li je stablo i dalje balansirano, i ako nije, izvršiti korekcije kako bi stablo ponovo bilo balansirano.

Sada malo više detalja oko samog algoritma i rada sa binarnim stablom pretrage.

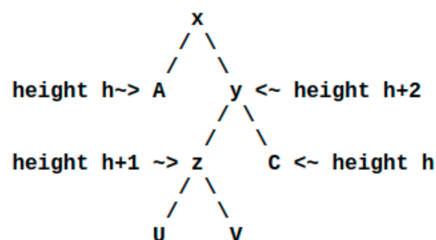
Visina u nekom čvoru predstavlja dužinu najdužeg puta od tog čvora do listova stabla, uzimajući u obzir i početak i kraj. U skladu sa tim, visina lista je 1, a visina ne-praznog stabla je visina njegovog korena.

Pretpostavimo da imamo stablo T kod kojeg je levo podstablo visine h, a desno visine $h+2$. Tada stablo T očigledno nije balansirano, ali pretpostavimo prvo da su oba njegova podstabla balansirana. Da bi se stablo izbalansiralo, potrebno je izvršiti tzv *jednostruku rotaciju.* Na sledećem primeru (slika 27) ćemo prikazati kako se ona vrši, pri čemu su x i y vrednosti ključa odgovarajućih čvorova, a A, B i C su podstabla.

Jednostavno je proveriti da jednostruka rotacija održava osobine binarnog



Slika 28: Visina stabla nakon rotacije



Slika 29: Proširenje podstabla B

stabla pretrage u vezi poretka. Na primer, sve vrednosti ključeva u podstablu B moraju biti veće od x, a manje od y, a to je zadovoljeno i pre i posle rotacije. Takođe, postoji slična jednostruka rotacija na drugu stranu, koja se izvodi kada je visina levog podstabla $h{+}2$ a desnog h.

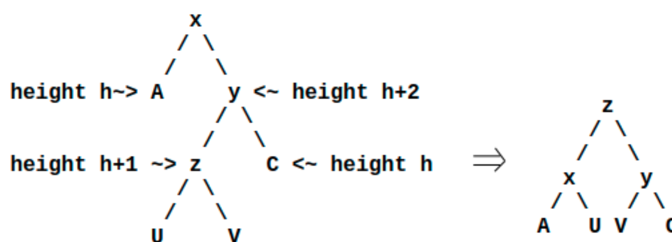
Da li smo ovakvom rotacijom rešili problem neizbalansiranog stabla? Barem jedno od podstabala B ili C mora imati visinu $h{+}1$, jer u suprotnom čvor y ne bi imao visinu $h{+}2$. Pošto je podstablo u čvoru y balansirano, znamo da i B i C imaju visine h ili $h{+}1$. Nakon rotacije dobijamo situaciju sa slike 28:

Pretpostavimo da B ima visinu h. Tada, podstablo iz čvora x ima visinu $h{+}1$ i kompletno stablo je balansirano. Međutim, ako podstablo B ima visinu $h{+}1$, tada podstablo u čvoru x ima visinu $h{+}2$ i ako podstablo C ima visinu h, kompletno stablo nije balansirano. Očigledno, u tom slučaju jednostruka rotacija nije dovoljna.

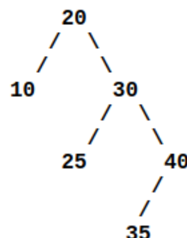
Pošto B ima visinu $h{+}1$, to podstablo ne može biti prazno, tako da možemo da ga proširimo na još jedan nivo (slika 29).

Sličnim razmatranjem kao ranije, zaključujemo da oba podstabla U i V imaju visine ili h ili h - 1. *Dvostruka rotacija* podiže čvor z između čvorova x i y, što rezultuje stablom prikazanim na slici 30.

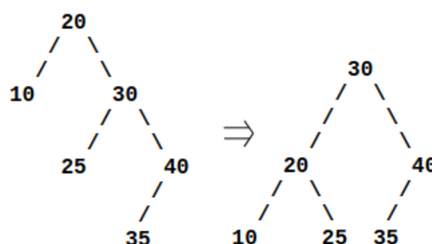
Podstabla u čvorovima x i y imaju jedno podstablo visine h i drugo podstablo visine h ili h - 1, tako da su oba podstabla balansirana, kao i kompletno stablo. Nije teško ustanoviti da su relacije između ključeva u novom stablu i dalje zadovoljene: na primer, svi ključevi iz podstabla V su veći od z, manji od y i



Slika 30: Dvostruka rotacija



Slika 31: Primer odluke o tipu rotacije

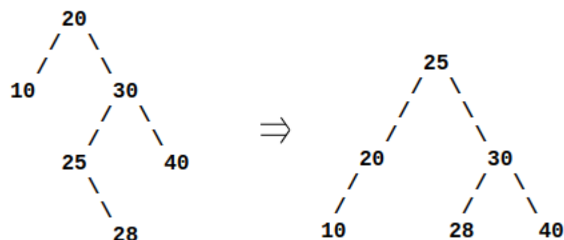


Slika 32: Rešenje jednostrukom rotacijom

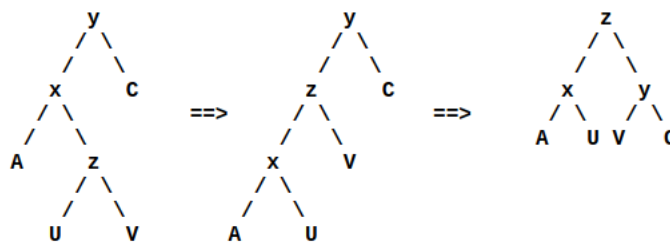
veći od x, a sve to je zadovoljeno i kod stabla dobijenog dvostrukom rotacijom. Takođe, kao i u prošlom slučaju, postoji slična dvostruka rotacija na drugu stranu kada levo podstablo ima visinu za 2 veću od desnog podstabla.

Ako je stablo ili podstablo izbalansirano po visini, nema potrebe vršiti rotaciju bilo kojeg tipa. Međutim, ukoliko nije, potrebno je odlučiti da li je potrebna jednostruka ili dvostruka rotacija. To se vrši tako što se krene od korena stabla (ili podstabla) i prave se dva koraka u pravcu višeg (dubljeg) podstabla. Na primer, posmatramo stablo na slici 31.

Levo podstablo ima visinu 1, dok desno ima visinu 3, što daje indikaciju da je potrebno vršiti rotaciju. Nakon dva koraka u pravcu višeg (dubljeg) podstabla



Slika 33: Primer dvostruke rotacije



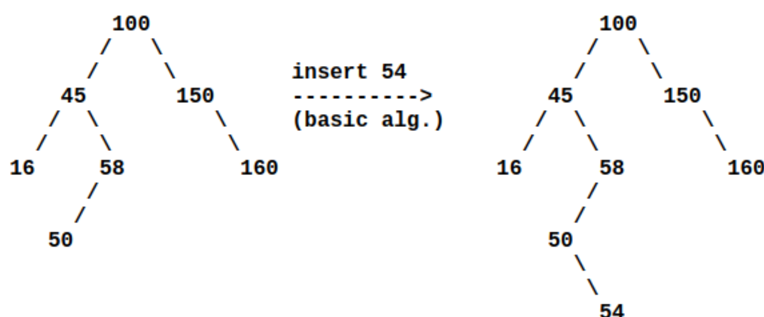
Slika 34: Dvostruka rotacija kao dve jednostruke rotacije

od 20 idemo na 30 a onda na 40. Ako su oba koraka učinjena u istom smeru (oba desno ili oba levo) potrebno je uraditi jednostruku rotaciju. U primeru sa slike 31, jednostrukom rotacijom će se dobiti stablo kao na slici 32.

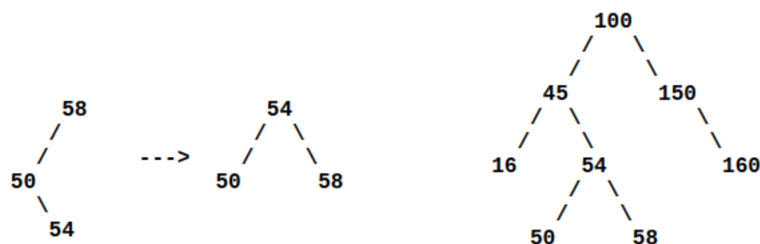
Međutim, ako su dva koraka u suprotnim smerovima (levo pa desno ili obrnuto), dvostruka rotacija je potrebna. Na primeru sa slike 33, dva koraka će nas od 20 dovesti do 30 (u desno) a onda do 25 (u levo). Kao rezultat, potrebna je dvostruka rotacija, nakon koje dobijamo stablo sa desne strane.

Dvostruka rotacija se zapravo sastoji od dve jednostruke rotacije (otud naziv dvostruka rotacija). Jedino je važno obratiti pažnju da će se ove dve jednostruke rotacije vršiti u suprotnim smerovima - jedna sa desna na levo, a druga sa leva na desno (ili u obrnutom rasporedu). Na primeru sa slike 34 krajnje desno stablo je dobijeno najpre jednostrukom rotacijom sa desna na levo (u suprotnom smeru) u čvoru x, a zatim jednostrukom rotacijom sa leva na desno u korenu stabla.

Na kraju, kada se vrše rotacije u datom čvoru, važno je da su podstabla izbalansirana (na početku, u prvom primeru, smatrali smo da je to zadovoljeno). Ova osobina je obezbeđena tako što se stablo balansira „od dole na gore". Na primer, ako treba da se doda čvor u stablo, kreće se na dole u pravcu ka mestu na kojem se čvor dodaje. Nakon toga se vraća istim putem nazad. U svakom čvoru se proverava da li je rotacija potrebna i ako jeste, izvršava se. Kao rezultat, moguće je da je potrebno izvršiti više od jedne rotacije. Čak i ako je tako, vrši



Slika 35: Dodavanje čvora u stablo



Slika 36: Dvostruka rotacija na putu „od dole ka gore" i kompletno stablo nakon toga

se jedna po jedna „od dole na gore". Evo primera koji pokazuje ovaj postupak koji je dat na slici 35. Na početku se samo dodaje čvor prateći putanju do mesta na kojem se čvor dodaje.

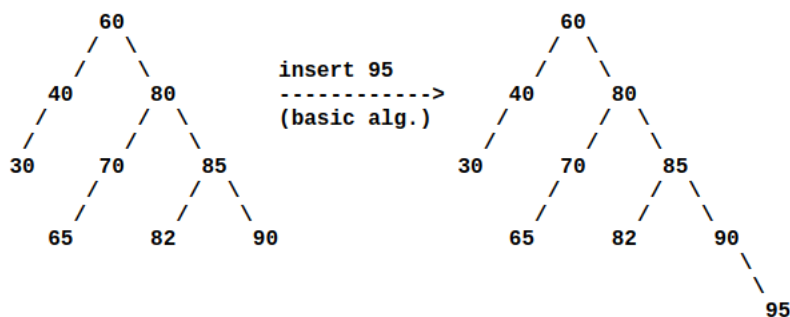
Nakon toga idemo istim putem na gore i proveravamo da li je potrebno izvršiti rotaciju. Čvor koji sadrži 50 je već izbalansiran, ali čvor sa ključem 58 nije. Ako se u njemu naprave dva koraka u pravcu višeg podstabla ide se levo pa desno. Dakle, dvostruka rotacija je potrebna i dobijamo stablo prikazano na slici 36.

Prateći silaznu putanju na gore, proverava se još čvor sa ključem 45 i čvor sa ključem 100. Pošto su oba ova čvora balansirana, nema potrebe za dodatnim rotacijama.

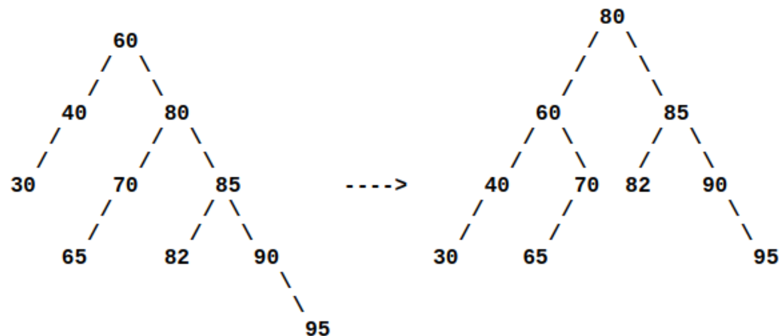
Slika 37 prikazuje još jedan primer. Nakon dodavanja čvora sa ključem 95, vraćamo se na gore istim putem.

Čvorovi koji sadrže ključeve 90, 85 i 80 svu već izbalansirani, ali koren nije jer je visina levog podstabla 2, a desnog 4. Ako se iz korena stabla krene dva koraka u pravcu viših podstabala, oba koraka su u desno, tako da potrebno izvršiti jednostruku rotaciju da bi se binarno stablo pretrage izbalansiralo. Rezultat nakon jednostruke rotacije prikazan je na slici 38.

Brisanje čvora iz stabla, kao što smo rekli, vrši se kada proces (zadatak)



Slika 37: Dodavanje čvora u stablo



Slika 38: Jednostruka rotacija koja daje balansirano stablo

predstavljen tim čvorom više nije spreman za izvršavanje. Postoje tri različita scenarija koji mogu da se pojave prilikom brisanja čvora iz stabla:

1. Čvor je list (nema decu čvorove) - tada se taj čvor samo ukloni iz stabla (čvorovi 95, 82, 65 i 30 u finalnom, balansiranom stablu sa slike 38)

2. Čvor ima jedno dete čvor - čvor se briše, a njegov roditelj čvor se postavlja da bude roditelj čvor njegovom dete čvoru (npr. čvor 40 u balansiranom stablu sa slike 38, gde se za novog roditelja čvora 30 postavlja čvor 60)

3. Čvor ima dvoje dece čvorova - u tom slučaju se čvor ne briše, pronalazi se minimalni ključ u desnoj podgrani i ta vrednost se upisuje umesto prethodne, a čvor sa minimalnim ključem u desnoj podgrani (sada duplikat čvor) se uklanja. Pošto će on imati minimalnu vrednost u podgrani, taj čvor sigurno *neće imati* levu podgranu (da ima, bilo koji ključ iz leve podgrane bi bio manji od njega), tako da se uklanjanje duplikat čvora svodi na brisanje (ako nema ni desnu podgranu) ili brisanje i postavljanje njegovog roditelja za roditelja njegovog jedinog deteta (ako ima desnu podgranu).

U slučaju 3 od gore, moguće je umesto pronalaženje minimalnog ključa u desnoj podgrani, tražiti maksimalan ključ u levoj podgrani i njega koristiti prilikom zamene. U oba slučaja zadovoljeno je da će stablo nakon ažuriranja zadržati svoje zahtevane karakteristike:

- minimalni ključ u desnoj podgrani je veći od bilo kog u levoj, a manji od bilo kojeg u desnoj, pa se bezbedno može staviti umesto vrednosti čvora koji želimo da brišemo

- maksimalni ključ u levoj podgrani je manji od svih u desnoj podgrani, a veći od svih u levoj, tako da se i on može koristiti kao nova vrednost čvora koji se briše.

Kao primer za slučaj 3, posmatramo čvor 85 iz konačnog, balansiranog stabla sa slike 38. Minimalna vrednost u njegovom desnom podstablu je 90, koja se upisuje na mesto čvora 85. Čvor koji je prvobitno imao ključ 90 je sada duplikat i, kao što smo rekli, on ne može imati levu podgranu, dok u ovom slučaju ima desnu. Pošto je tako, duplikat čvor se briše a za roditelja čvora 95 postavlja se čvor koji je imao vrednost 85 a sada ima vrednost 90.

Kao poslednja digresija na temu balansiranih binarnih stabala pretrage, obzirom da se *vruntime* procesa (zadatka) koristi kao ključ u stablu, postavlja se pitanje šta raditi ako se pojave dva čvora sa istim ključevima (situacija koja realno može da se dogodi). Da bi stablo zadržalo minimalne dimenzije, a da bi i dalje svi prethodno opisani algoritmi na balansiranom binarnom stablu pretrage mogli da se primenjuju, svakom čvoru u stablu dodeljuje se, osim ključa i brojač, inicijalno postavljen na 1 prilikom dodavanja čvora u stablo. Tada, ako se pojavi potreba za dodavanjem novog čvora čiji ključ već postoji u stablu, novi čvor neće biti kreiran, već će se samo ažurirati brojač čvora koji ima taj ključ. Prilikom dodavanja čvora, brojač se povećava za jedan, a prilikom brisanja on se smanjuje za jedan. Prilikom odabira koji će se proces (zadatak) naredni izvršavati, u slučaju kada dva ili više procesa imaju iste ključeve i to ključeve koji su minimalni u celom stablu, selekcija procesa se vrši koristeći FCFS ili neki drugi algoritam raspoređivanja.