Projekat na predmetu "Operativni Sistemi 2"

Lazar Premović
pl190091@student.etf.bg.ac.rs

Februar 2022.

Rezime

Projekat na predmetu "Operativni Sistemi 2" se ove godine sastojao iz implementacije nekoliko algoritama raspoređivanja na operativnom sistemu xv6 tako da je njihova zamena moguća bez ponovnog pokretanja sistema.

Dodatno, ukoliko je projekat bio branjen u predroku bilo je potrebno implementirati i sistem koji će voditi računa o afinitetima procesa i uzeti ih u obzir pri raspoređivanju. Specifikacija sistema za afinitet je u potpunosti prepuštena studentima te predstavlja izuzetnu priliku za implementaciju inovativnih i kreativnih rešenja.

Cilj ovog dokumenta je da predstavi i obrazloži projektne odluke donete pri izradi ovog projekta sa fokusom na sistem za afinitet.

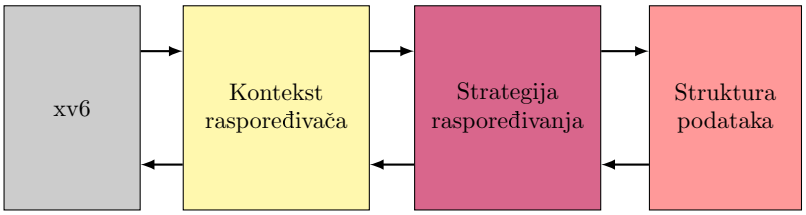
Izvorni kod celokupnog projekta se može naći na:
`https://github.com/lazar2222/OS2Project`

Sadržaj

1 Pregled arhitekture rešenja

Operativni sistem xv6 je baziran na monolitnoj arhitekturi te se njegov raspoređivač nalazi direktno u delu kernela odgovornom za upravljanje procesima.

Kako bi projekat bilo lakše realizovati odlučeno je da se raspoređivač izdvoji u odvojenu komponentu sa slojevitom arhitekturom koja se sastoji od tri sloja: sloj konteksta raspoređivača, sloj strategije raspoređivanja i sloj strukture podataka.



Slika 1: Dijagram slojevite arhitekture projekta

Sloj konteksta raspoređivača je implementiran u fajlovima *scheduler.c/.h*, i odgovoran je za funkcionalnosti zajedničke za sve strategije raspoređivanja.

Sloj strategije raspoređivanja je implementiran za svaki algoritam raspoređivanja i to za **Round robin** algoritam[1] u fajlovima *DRR.c/.h*, za **Shortest job first** algoritam u fajlovima *SJF.c/.h* i za **Completely fair** algoritam u fajlovima *CFS.c/.h* i odgovoran je za funkcionalnosti specifične za svaki algoritam raspoređivanja.

Sloj strukture podataka je implementiran za svaku strukturu podataka koju algoritmi raspoređivanja koriste u pozadini i to u dve varijacije u zavisnosti od toga da li struktura uzima u obzir i afinitet procesa (samim tim se i sistem za afinitet implementira u ovom sloju). Kako sva tri gore navedena algoritma ili ne koriste specifične strukture podataka[2] ili koriste prioritetni red (ovde implementiran kao **heap** struktura podataka) potrebno je imati samo dve implementacije sloja strukture podataka: heap koji ne uzima u obzir afinitet (implementiran u fajlovima *heap.c/.h*) i heap koji uzima u obzir afinitet (implementiran u fajlovima *affinityHeap.c/.h*).

2 Kontekst raspoređivača

Odgovornosti konteksta raspoređivača su:

- Definisanje i implementacija ugovora prema operativnom sistemu.
- Definisanje ugovora koji implementiraju strategije raspoređivanja.
- Definisanje neophodnih konstanti.
- Vođenje računa o trenutno izabranoj strategiji raspoređivanja i sistemu za afinitet.

[1] Ovo je podrazumevani algoritam koji xv6 koristi i implementiran je čisto radi vežbe i provere slojevite arhitekture.

[2] Round robin algoritam koristi samo listu procesa `proc` koja vec postoji u samom kernelu xv6 operativnog sistema.

- Inicijalizacija celokupnog sistema za raspoređivanje.
- Vođenje računa o kritičnim sekcijama pri raspoređivanju na više procesora.
- Vođenje evidencije o broju procesa koji se trenutno izvršavaju na procesorima.
- Promena strategije raspoređivanja i sistema za afinitet.

2.1 Ugovor prema operativnom sistemu

Sprega sa operativnim sistemom je realizovana prostim pozivima funkcija koji su dodatno parametrizovani konstantama iz fajla *scheduler.h*.

```
1  #define TIMER_SOURCE_USER 1
2  #define TIMER_SOURCE_KERNEL 0
3  #define REASON_TIME_SLICE_EXPIRED 0
4  #define REASON_AWAKENED 1

22 void sched_initialize();
23 void sched_perCoreInitialize(int core);
24 int sched_get(int core);
25 void sched_put(int processIndex,int reason);
26 void sched_timer(int user);
```

Primer 1: Fajl:*scheduler.h* Ugovor prema operativnom sistemu

Funkcija `sched_initialize` se poziva samo jednom pri pokretanju sistema (*main.c:34*).
Funkcija `sched_perCoreInitialize` se poziva pri pokretanju sistema po jednom za svako njegovo jezgro (*proc.c:479*) sa identifikatorom koji odgovara tom jezgru.
Funkcija `sched_get` se poziva kada kernel zatraži proces iz liste spremnih (*proc.c:485*) i kao povratnu vrednost vraća indeks izabranog procesa u nizu `proc`.
Funkcija `sched_put` se poziva kada kernel smešta proces u listu spremnih (*proc.c:279,349,535,603,624*), kao argumente prima indeks procesa u nizu `proc` i razlog smeštanja `REASON_TIME_SLICE_EXPIRED` (ukoliko je procesu istekao vremenski kvant) ili `REASON_AWAKENED` (ukoliko se proces odblokirao).
Funkcija `sched_timer` se poziva na svaki prekid tajmera (*trap.c:82,155*), sa argumentom koji označava da li se prekid desio u korisničkom (`TIMER_SOURCE_USER`) ili kernel (`TIMER_SOURCE_KERNEL`) režimu.

Sve gore navedene funkcije prosleđuju pozive odgovarajućim funkcijama izabranog algoritma raspoređivanja (o mehanizmu prosleđivanja će biti više reči kasnije).

2.2 Ugovor prema strategiji raspoređivanja

Kako bi promena raspoređivača bila što elegantnija, sprega sa strategijom raspoređivanja je umesto prostim pozivima funkcija realizovana korišćenjem struktura koje reprezentuju strategije raspoređivanja i sadrže pokazivače na funkcije te strategije raspoređivanja[3].
Funkcije na koje ti pokazivači pokazuju imaju identične potpise i semantiku kao njima analogne funkcije ugovora prema operativnom sistemu.

[3] Nalik projektnom uzorku strategija.

```
14 struct schedulingStrategy{
15     void (*initialize)();
16     void (*perCoreInitialize)(int);
17     int (*get)(int core);
18     void (*put)(int,int);
19     void (*timer)(int);
20 };
```

Primer 2: Fajl:*scheduler.h* Sruktura koje reprezentuje strategiju raspoređivanja

```
15 struct schedulingStrategy* currentSchedulingStrategy;

25  currentSchedulingStrategy=&CFSscheduler;

76  currentSchedulingStrategy->put(processIndex,reason);
```

Primer 3: Fajl:*scheduler.c* Poziv izabrane strategije raspoređivanja

Pokazivač `currentSchedulingStrategy` zajedno sa celobrojnim indikatorom `affinityEN` ujedno i predstavljaju način na koji kontekst raspoređivača vodi računa o trenutno izabranoj strategiji raspoređivanja i sistemu za afinitet. Pokazivač `currentSchedulingStrategy` uvek pokazuje na strukturu tipa `schedulingStrategy` koja odgovara trenutno izabranoj strategiji raspoređivanja dok celobrojni indikator `affinityEN` ima vrednost jedne od tri konstante definisane u fajlu *scheduler.h*: `AFFINITY_ENABLED`, `AFFINITY_ENABLED_AGING` i `AFFINITY_DISABLED` o čijem će značenju biti reči kasnije.

2.3 Inicijalizacija sistema za raspoređivanje

Inicijalizacija sistema za raspoređivanje se odvija u funkciji `sched_initeialize` dok funkcija `sched_perCoreInitialize` samo prosleđuje poziv odgovarajućoj strategiji raspoređivanja.
Inicijalizacija se sastoji iz sledećih koraka:

1. Postavljanje podrazumevane strategije raspoređivanja i sistema za afinitet (ovo uključuje promenljive `currentSchedulingStrategy` i `affinityEN` kao i interne parametre konkretnih strategija raspoređivanja i sistema za afinitet)
2. Inicijalizacija brave kojom se garantuje međusobno isključenje raspoređivača ukoliko postoji više jezgara.
3. Inicijalizacija sistema za afinitet i trenutno izabrane strategije raspoređivanja pozivima odgovarajućih funkcija.

2.4 Raspoređivanje na više procesora

Problem konkurentnosti pri raspoređivanju na više procesora je rešen uvođenjem kritične sekcije tako da samo jedno jezgro može biti istovremeno unutar funkcija `sched_get` i `sched_put`. Ta kritična sekcija je realizovana korišćenjem brava koje su već implementirane u kernelu.

```
73 void sched_put(int processIndex,int reason)
74 {
75   acquire(&sched_lock);
76   currentSchedulingStrategy->put(processIndex,reason);
77   release(&sched_lock);
78 }
```

Primer 4: Fajl:*scheduler.c* Primer međusobnog isključenja

2.5 Vođenje evidencije o broju procesa koji se trenutno izvršavaju

Kontekst raspoređivača vodi evidenciju o broju procesa koji se trenutno izvršavaju na jezgrima sistema (ova informacija je korisna sistemu za afinitet, a o načinu na koji je on koristi će biti reči kasnije). Ovo vođenje evidencije je trenutno implementirano korišćenjem bit vektora gde bit sa vrednošću 1 označava da se na tom jezgru izvršava proces. Ovi bitovi se ažuriraju na osnovu povratne vrednosti funkcije `get` trenutno izabranog algoritma raspoređivanja (ukoliko nema procesa za izvršavanje funkcija vraća -1). Ukoliko se bit vektor promenio, broj procesa koji se izvršavaju se dobija brojanjem bitova sa vrednošću 1.

```
48      int nrp=runningBitFlag;
49      if(val==-1)
50      {
51          nrp&=~(1<<core);
52      }
53      else
54      {
55          nrp|=(1<<core);
56      }
57      if(nrp!=runningBitFlag)
58      {
59          if((nrp&(1<<core))!=0)
60          {
61              runningProc++;
62          }
63          else
64          {
65              runningProc--;
66          }
67      }
68      runningBitFlag=nrp;
```

Primer 5: Fajl:*scheduler.c* Računanje broja procesa koji se izvršavaju

Naknadno je uočeno da ovaj potencijalno spor proces može znatno da se optimizuje. Predložena modifikacija se sastoji od uvođenja dodatnog argumenta koji predstavlja pokazivač na proces koji se do sada izvršavao (ta informacija se nalazi u promenljivoj `c->proc`) tako da se prostim poređenjem rezultata može odrediti da li je potrebno inkrementirati ili dekrementirati broj procesa koji se trenutno izvršavaju.

```
1  //proc je pokazivac na proces koji se do sada izvrsavao
2  //val je indeks novog procesa ili -1 ukoliko ne
       postoji spreman proces
3  if(proc==0 && val!=-1)
4  {
5      runningProc++;
6  }
7  else if(proc!=0 && val==-1)
8  {
9      runningProc--;
10 }
```

Primer 6: Optimizovano računanje broja procesa koji se izvršavaju

2.6 Promena strategije raspoređivanja i sistema za afinitet

```
15 #define SCHEDULER_DRR 0
16 #define SCHEDULER_SJF 1
17 #define SCHEDULER_SJF_PREEMPRIVE 2
18 #define SCHEDULER_SJF_EAGER_PREEMPRIVE 3
19 #define SCHEDULER_CFS 4
20 #define AFFINITY_ENABLED 1
21 #define AFFINITY_ENABLED_AGING 2
22 #define AFFINITY_DISABLED 0

27 int sched_change(int type,int factor,int timeslice,int
       ease);
28 int sched_affinity(int enabled,int initialLaziness,
       int maxAge);
```

Primer 7: Fajl:*scheduler.h* Konstante i funkcije za promenu strategije raspoređivanja i sistema za afinitet

Korišćenjem gore navedenih funkcija i konstanti moguće je promeniti trenutno izabranu strategiju raspoređivanja i sistem za afinitet kao i njihove parametre (ukoliko se za parametar prosledi vrednost -1 taj parametar neće biti izmenjen).

Promena strategije raspoređivanja započinje proverom validnosti prosleđenih argumenata i određivanjem strukture koja pripada novoj strategiji. Potom se ulazi u kritičnu sekciju nad istom bravom koja se koristi za funkcije `get` i `set` kako bi onemogućili raspoređivanje i zabranili prekide[4]. Nakon što smo ušli u kritičnu sekciju možemo da počnemo da modifikujemo unutrašnje stanje raspoređivača. Ukoliko se ne menja strategija raspoređivanja već samo varijanta algoritma (npr. da li dolazi do preotimanja ili ne) dovoljno je samo promeniti vremenski kvant svih procesa koji se izvršavaju[5] tako da poštuju vremenski kvant novo podešene varijante algoritma.

```
101   if(SJFtype!=type) {
102     //Respect preemption and force rescheduling on
        preemptive schedulers
103     //No need to update any other parameters if we are
        not changing scheduling algorithms
104     int nts = type == SCHEDULER_SJF ? 0 : 1;
105     for (int i = 0; i < NCPU; ++i) {
106       if (cpus[i].proc != 0) {
107         cpus[i].proc->timeslice = nts;
108       }
109     }
110   }
```

Primer 8: Fajl:*scheduler.c* Ažuriranje vremenskog kvanta

Nakon toga možemo ažurirati parametre strategija raspoređivanja bez potrebe za nekim detaljnijim proverama. Ukoliko se u potpunosti menja strategija raspoređivanja (tj. menja se pokazivač na strukturu sa pokazivačima na funkcije) potrebno je obaviti još nekoliko operacija. Prvo je potrebno resetovati polja koja sa odnose na raspoređivanje u strukturi procesa a potom i ažurirati unutrašnje strukture ukoliko je potrebno. Unutrašnje strukture je potrebno ažurirati ukoliko se prelazi sa strategije koja ne koristi heap na strategiju koja koristi heap (tada je potrebno ubaciti svaki

[4] Funkcija `acquire` ujedno zabranjuje i prekide

[5] Procesi u listi spremnih će dobiti odgovarajući vremenski kvant pri dohvatanju iz liste spremnih.

proces sa stanjem RUNNABLE u heap) ili obratno (gde je dovoljno samo isprazniti heap i ponovo inicijalizovati novu strategiju). Pri dodavanju u heap ili njegovom pražnjenju potrebno je voditi računa da se te operacije vrše nad pravilnom strukturom podataka u zavisnosti od stanja sistema za afinitet. Kako je sistem za afinitet dodat naknadno odluka nad kojom strukturom podataka se vrši operacija se donosi uslovnim grananjem koje proverava indikator affinityEN umesto da se koristi znatno elegantnije rešenje slično rešenju za strategije raspoređivanja[6]. Nakon ažuriranja struktura podataka konačno se ažurira i pokazivač na strukturu raspoređivača, napušta se kritična sekcija i time je promena strategije završena.

```
130         if(currentSchedulingStrategy==&DRRscheduler)
131         {
132             for (int i = 0; i < NPROC; ++i) {
133                 if(proc[i].state==RUNNABLE)
134                 {
135                     if(affinityEN!=AFFINITY_DISABLED)
136                     {
137                         AHput(i);
138                     }
139                     else
140                     {
141                         heapInsert(i);
142                     }
143                 }
144             }
145         }
146         if(newSS==&DRRscheduler)
147         {
148             if(affinityEN!=AFFINITY_DISABLED)
149             {
150                 AHclear();
151             }
152             else
153             {
154                 heapClear();
155             }
156             newSS->initialize();
157         }
```

Primer 9: Fajl:*scheduler.c* Ažuriranje struktura podataka

Promena sistema za afinitet Ima nešto prostiju strukturu i započinje slično, proverom argumenata i onemogućavanjem raspoređivanja. Nakon toga se resetuju polja koja se odnose na afinitet u strukturi procesa i pozivom odgovarajuće funkcije se kompletno resetuje struktura podataka koja uzima u obzir afinitet. Ukoliko se zbog promene sistema za afinitet u potpunosti promenila struktura podataka koja se koristi potrebno je isprazniti staru strukturu a u novu strukturu dodati sve procese sa stanjem RUNNABLE. Potom se ažuriraju parametri i napušta se kritična sekcija čime je promena sistema za afinitet završena.

[6] Misli se na rešenje sa strukturama koje sadrže pokazivače na funkcije.

3 Strategija raspoređivanja

Odgovornosti strategije raspoređivanja su:

- Implementacija ugovora prema kontekstu raspoređivača.
- Implementacija određenog algoritma raspoređivanja korišćenjem sloja strukture podataka

Kako se implementacija strategije raspoređivanja svodi na implementaciju poznatog algoritma, ova sekcija će se fokusirati na pojedine zanimljive detalje.

3.1 Implementacija ugovora prema kontekstu raspoređivača

Ugovor između konteksta raspoređivača i strategije raspoređivanja je već opisan iznad te će ovde biti samo prikazano kako se on implementira sa strane strategije raspoređivanja. Da bi kontekst raspoređivača mogao da koristi strategiju raspoređivanja potrebno je da ona u .h fajlu izveze globalnu promenljivu tipa `schedulingStrategy` koja sadrži pokazivače na odgovarajuće funkcije te strategije. Strategija raspoređivanja opciono može izvesti i parametre ukoliko ostatku kernela želi da omogući njihovo podešavanje (preporučljivo preko funkcije `sched_change`).

```
1 extern struct schedulingStrategy SJFscheduler;
2 extern int SJFtype;
3 extern int SJFtimeslice;
4 extern int SJFfactor;
5 extern int SJFease;
```

Primer 10: Fajl:*SJF.h* Izvezene promenljive

```
108 struct schedulingStrategy
      SJFscheduler={initializeSJF,perCoreInitializeSJF,getSJF,
      putSJF,timerSJF};
```

Primer 11: Fajl:*SJF.c* Globalna struktura sa pokazivačima na funkcije

3.2 Dodatna polja za potrebe raspoređivača

Kako bi bilo moguće implementirati zahtevane algoritme raspoređivanja potrebno je dodati još neka polja u strukturu koja predstavlja proces u operativnom sistemu xv6.

```
109   //scheduling data
110   int priority;                 // Processes with the
        lowest priority are selected first
111   int timeslice;                // Amount of time
        process runs before its rescheduled (0 for infinite)
112   int executiontime;            // Amount of time
        process has been running without going into
        SLEEPING state
113   int schedtmp;                 // Additional field
        used by schedulers
```

Primer 12: Fajl:*proc.h* Dodatna polja za potrebe raspoređivanja

Polje `priority` predstavlja vrednost na osnovu koje su procesi raspoređeni u prioritetnom redu sloja strukture podataka (prednost imaju manje vrednosti ovog polja) a način računanja samog prioriteta zavisi od konkretnog algoritma raspoređivanja (kod **SJF** to je τ dok je kod **CFS** to vreme izvršavanja).

Polje `timeslice` predstavlja broj prekida od strane tajmera koji trebaju da proteknu pre nego što se proces vrati u red spremnih i da šansu nekom drugom procesu da koristi procesor. Ova vrednost se u zavisnosti od konkretnog algoritma i njegove varijante postavlja pri vađenju procesa iz liste spremnih a potom se dekrementira pri svakom prekidu tajmera. Vrednost 0 označava da proces nikada neće predati procesor.
Polje `executiontime` predstavlja broj prekida tajmera koje je proces proveo na procesoru od kada se odblokirao (vrednost se ne resetuje pri isteku vremenskog kvanta).
Polje `schedtmp` nema neku posebnu semantiku već je data sloboda konkretnom algoritmu raspoređivanja da ga koristi za svoje potrebe. Algoritam **SJF** ovo polje koristi pri željnom preotimanju da čuva τ pri računanju τ_{live}, dok **CFS** ovo polje koristi da čuva trenutak kada je proces ubačen u red spremnih.

3.3 Željno preuzimanje kod SJF algoritma

Željno preuzimanje je mala modifikacija **SJF** algoritma sa preuzimanjem koja pokušava da poboljša performanse u odnosu na standardnu implementaciju **SJF** algoritma sa preuzimanjem. Željno preuzimanje radi tako što pri svakom pokušaju preotimanja računa τ_{live} po formuli:

$$\tau_{live} = (1 - \alpha) \cdot \tau + \alpha \cdot e$$

Gde je e vreme izvršavanja procesa od poslednjeg izračunavanja τ. Ova formula je skoro identična formuli za τ, jedina razlika da njeno izračunavanje pri svakom pokušaju preotimanja ne utiče na τ izračunato pri ubacivanju u red spremnih procesa. Može se uočiti da će u trenutku ubacivanja procesa u red spremnih τ_{live} biti jednako tada izračunatom τ, šta više τ_{live} će kroz vreme linearno težiti ka τ. Tako da τ_{live} predstavlja τ ukoliko bi se proces u ovom trenutku dodao u red spremnih.
Sada koristeći τ_{live} možemo proveriti da li će sledeća predikcija za ovaj proces sigurno biti veća od trenutno najmanje predikcije i ukoliko jeste prepustićemo procesor tom procesu (pritom će prioritet procesa koji je prepustio procesor biti njegovo τ_{live} umesto τ).
Kako bi smanjili režijske troškove u situaciji gde imamo dva procesa koji se izvršavaju znatno duže od njihove predikcije i stalno preotimaju procesor jedan drugom, dodatno se uvodi faktor relaksacije koji umanjuje τ_{live} za neku konstantu i time omogućava procesu da njegovo τ_{live} malo prekorači τ procesa na vrhu prioritetnog reda pre nego što prepusti procesor čime se ubrzava konvergencija τ_{live} ka τ.

```
84          //recheck scheduling calculate comparison
        priority "live tau"
85          p->priority = (SJFtype ==
        SCHEDULER_SJF_EAGER_PREEMPRIVE) ?
        (((p->schedtmp*(128-SJFfactor)) +
        (p->executiontime*SJFfactor)+65) / 128) - SJFease :
        p->priority;
86          acquire(&sched_lock);
87          int hmin;
88          if(affinityEN!=AFFINITY_DISABLED)
89          {
90              hmin=AHmin(mycpu()-cpus);
91          }
92          else
93          {
```

```
94              hmin=heapMin();
95          }
96          if(hmin!=-1 && proc[hmin].priority<p->priority)
97          {
98              release(&sched_lock);
99              yield();
100         }
101         else
102         {
103             release(&sched_lock);
104         }
```

Primer 13: Fajl:*SJF.c* Željno preuzimanje

Poređenje sa alternativnom implementacijom

Alternativna implementacija **SJF** algoritma sa preuzimanjem je da se preuzimanje desi samo ukoliko je τ procesa na vrhu prioritetnog reda manje od τ procesa koji se trenutno izvršava (ova implementacija se ponaša identično kao da se pri svakom prekidu obavezno vrši promena konteksta i ponovno raspoređivanje samo umanjuje režijske troškove). Ova implementacija dovodi do izgladnjivanja ukoliko neki proces naglo i znatno promeni svoje vreme izvršavanja (što se takođe implicitno dešava pri kreiranju novog procesa), dok željno preuzimanje nema ovakav problem jer čim proces znatno prekorači svoje predviđeno vreme dolazi do preuzimanja čime se sprečava izgladnjivanje[7].

3.4 Sprega sa slojem strukture podataka

Sprega sa slojem strukture podataka je realizovana prostim pozivima funkcija tako da je odgovornost na sloju strategije raspoređivanja da uslovnim grananjima odredi koju implementaciju sloja strukture podataka treba da pozove. Iako postoje elegantnija rešenja za ovu spregu (npr. rešenje sa strukturama koje sadrže pokazivače na funkcije) ovo rešenje je izabrano zato što je implementacija sloja strukture koja uzima u obzir afinitet dodata dosta kasno i implementiranje elegantnijeg rešenja bi zahtevalo restruktuiranje projekta, a elegantnije rešenje je ovde navedeno kao potencijalna ideja za dalje unapređenje projekta.

4 Sloj strukture podataka

Odgovornosti strukture podataka su:

- Implementacija ugovora prema strategiji raspoređivanja.
- Implementacija zahtevane strukture podataka sa ili bez uzimanja u obzir afiniteta procesa.

4.1 Ugovor prema strategiji raspoređivanja

Kako se sprega sa strategijom raspoređivanja svodi na pozive funkcija ugovor nije toliko striktan ali uglavnom obuhvata četiri funkcije sa definisanom semantikom koje dozvoljavaju male varijacije u argumentima.

[7] U ovom konkretnom slučaju. **SJF** algoritam sa željnim preuzimanjem je i dalje podložan drugim vrstama izgladnjivanja.

```
void heapInsert(int procIndex);
void heapRemove(int heapIndex);
void heapClear();
int heapMin();
```

Primer 14: Fajl:*heap.h* Ugovor prema strategiji raspoređivanja

```
void AHinit();
int AHget(int core);
void AHput(int procIndex);
void AHclear();
int AHmin(int core);
```

Primer 15: Fajl:*affinityHeap.h* Ugovor prema strategiji raspoređivanja

Funkcije `heapInsert` i `AHput` dodaju proces sa prosleđenim indeksom u red spremnih tako da je održana uređenost po polju `priority` gde manje vrednosti imaju veći prioritet.
Funkcije `heapClear` i `AHclear` uklanjaju sve elemente iz reda spremnih procesa.
Funkcije `heapMin` i `AHmin` dohvataju indeks najprioritetnijeg procesa u redu spremnih (proces sa najmanjom vrednošću polja `priority` u slučaju funkcije `heapMin` dok je za funkciju `AHmin` logika nešto kompleksnija), funkcija `AHmin` u obzir uzima i afinitet i kao argument prima identifikator jezgra za koje se proces dohvata.
Funkcija `heapRemove` uklanja proces na datoj poziciji u prioritetnom redu spremnih procesa (indeksiranje počinje od 1 tako da pozicija 1 odgovara procesu sa najmanjom vrednošću polja `priority`).
Funkcija `AHget` je analogna pozivu funkcije `heapMin()` pa zatim `heapRemove(1)` (tj. vraća najprioritetniji element i uklanja ga iz reda spremnih), razlika je to što funkcija `AHget` uzima u obzir afinitet i kao takva prima identifikator jezgra.
Funkcija `AHinit` inicijalizuje interne strukture podataka sistema za afinitet.

4.2 Izbor i implementacija strukture podataka bez afiniteta

Na osnovu ugovora prema strategiji raspoređivanja možemo razmatrati strukture podataka koje pružaju pogodne performanse potrebnih operacija.
Razmatrane strukture podataka su heap i crveno crno stablo a heap je izabran zbog prostije implementacije a male razlike u performansama.
Implementiran je binarni heap čiji su elementi indeksi procesa u nizu `proc` a čuvaju se u nizu dužine `NPROC+1` počevši od indeksa 1[8] (na indeksu 0 se čuva broj elemenata u heap-u), a operacija uklanjanja je proširena da može da ukloni bilo koji element iz heap-a a ne samo najmanji element (ovo konkretno nije bilo potrebno za implementaciju bez afiniteta ali je sve jedno bilo implementirano jer će kasnije biti korišćeno za implementaciju sa afinitetom).
Osnovna struktura i operacije nad heap-om su implementirane veoma slično i u implementaciji sa afinitetom.

[8] Ovo olakšava izračunavanje indeksa roditelja i dece.

5 Sistem za afinitet

Svrha sistema za afinitet je da vodi računa na kom jezgru se poslednji put neki proces izvršavao (kažemo da neki proces ima afinitet prema poslednjem jezgru na kojem se izvršavao) i da pokuša da taj proces rasporedi na to jezgro kako bi se poboljšale performanse.
Sistemi za afinitet se mogu podeliti na sisteme sa lokalnim raspoređivanjem i na sisteme sa globalnim raspoređivanjem.

Sistemi sa lokalnim raspoređivanjem održavaju odvojenu listu spremnih procesa za svako jezgro (tako da svi procesi u toj listi imaju afinitet prema tom jezgru) i pri odabiru procesa u obzir se uzimaju samo procesi iz te liste, te su režijski troškovi nešto manji nego kod sistema sa globalnim raspoređivanjem. Mana ovog sistema je to što je neophodno implementirati neku tehniku balansiranja broja procesa u listi svakog jezgra.

Sistemi sa globalnim raspoređivanjem održavaju samo jednu listu spremnih procesa[9] tako da u opštem slučaju bilo koji proces može biti izabran da se izvršava na bilo kom jezgru. Kako se nigde implicitno ne poštuje afinitet, sistemi sa globalnim raspoređivanjem moraju imati eksplicitnu logiku za rukovanje afinitetom. Prednost ovakvog sistema je nešto lakša implementacija za performanse uporedive sa sistemima sa lokalnim raspoređivanjem, dok je mana ovakvog sistema povećanje režijskih troškova u odnosu na sisteme sa lokalnim raspoređivanjem.

Za ovaj projekat je izabran sistem sa globalnim raspoređivanjem zbog lakše implementacije, potencijalno boljih performansi za isti uložen trud i činjenice da je raspoređivač bez afiniteta već dizajniran kao globalni raspoređivač.

5.1 Specifikacija sistema za afinitet

Sistem za afinitet je implementiran u dve varijante koje predstavljaju različite kompromise između pravednosti raspoređivanja i režijskih troškova. Standardna varijanta ima nešto manju pravednost raspoređivanja ali zato i manje režijske troškove, dok se varijanta sa starenjem trudi da poboljša pravednost raspoređivanja po cenu nešto većih režijskih troškova.

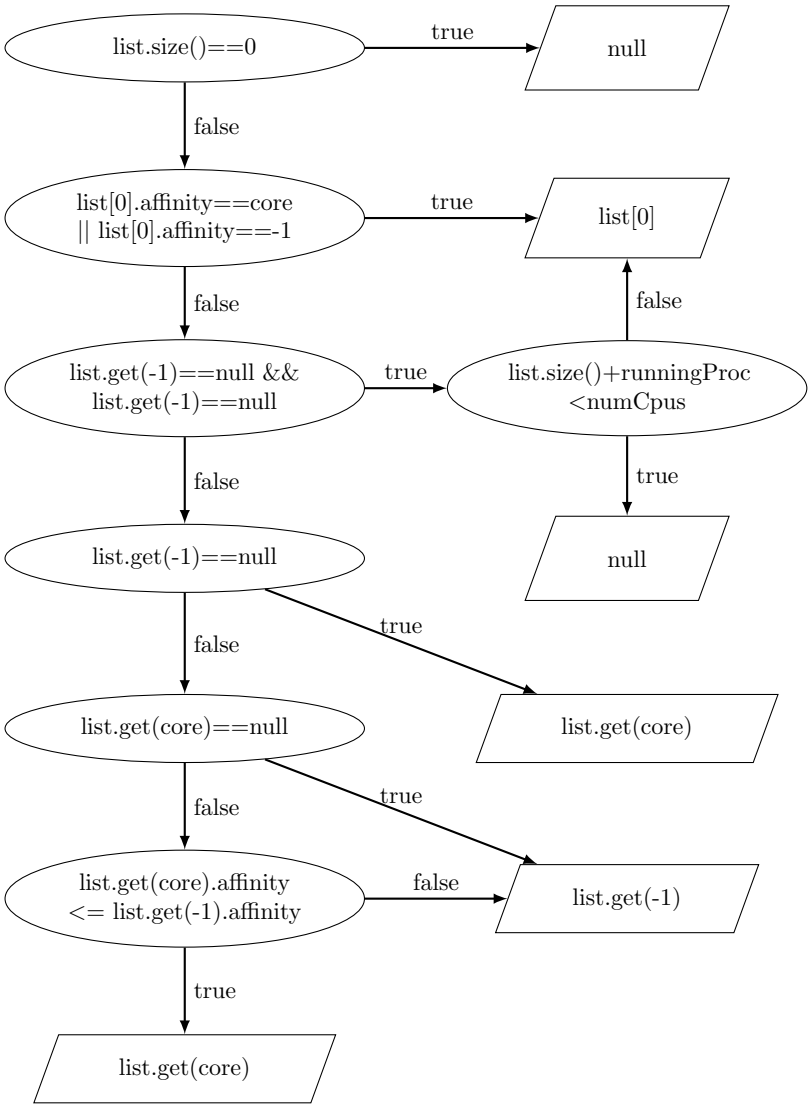
Dobijanje i gubljenje afiniteta
Procesi dobijaju afinitet ka nekom jezgru u trenutku kada bivaju raspoređeni na to jezgro.
Procesi mogu izgubiti afinitet (prelaze u stanje bez afiniteta) na dva načina: preteranim odlaganjem ili starenjem.

Preterano odlaganje: Ukoliko po algoritmu raspoređivanja neki proces treba da bude izabran od strane jezgra prema kome nema afinitet, on će odložiti svoje biranje u nadi da će biti izabran od strane jezgra prema kome ima afinitet. Ukoliko proces odloži svoj izbor više od specifikovanog broja puta[10], on gubi afinitet jer je vreme koje bi bilo dobijeno raspoređivanjem na odgovarajuće jezgro već izgubljeno odlaganjem. Time se sprečava da neki prioritetni proces preterano čeka ukoliko se na jezgru prema kome ima afinitet izvršava neki dugačak proces.

[9] Ili bar privid jedne liste spremnih procesa.
[10] Parametar pod imenom `AHinitialLaziness`.

Starenje: Ovaj način gubljenja afiniteta je implementiran samo u varijanti sa starenjem i ujedno predstavlja i jedinu razliku između dve varijante sistema za afinitet. Kod starenja proces gubi afinitet ukoliko se od njegovog poslednjeg izvršavanja na tom jezgru izvršio bar određen broj procesa[11]. Ovakvo gubljenje afiniteta omogućava elegantan prelaz procesa sa jednog na drugo jezgro bez potrebe za eksplicitnim balansiranjem a bazirano je na činjenici da će nakon izvršavanja nekoliko drugih procesa podaci prvobitnog procesa biti izbačeni iz keš memorije te više ne postoji benefit u raspoređivanju tog procesa na to isto jezgro.

Stablo odlučivanja za izbor procesa



Slika 2: Stablo odlučivanja

list predstavlja listu spremnih procesa, uređenu po prioritetu tako da se na poziciji nula nalazi najprioritetniji proces. *list.get(affinity)* dohvata najprioritetniji proces sa specifikovanim afinitetom ili *null* ukoliko takav proces ne postoji. Promenljiva *core* sadrži identifikator jezgra koje zahteva proces. Promenljiva *runningProc* sadrži broj procesa koji se trenutno izvršavaju na jezgrima dok promenljiva *numCpus* sadrži broj jezgara u sistemu. Vrednost -1 označava da proces nema afinitet.

[11] Parametar pod imenom `AHmaxAge`

Prvo proveravamo da li postoje spremni procesi, ukoliko ne postoje odmah vraćamo *null*.
Nakon toga proveravamo afinitet najprioritetnijeg procesa, ukoliko nema afinitet ili njegov afinitet odgovara jezgru koje je zatražilo proces biramo njega, u suprotnom smanjujemo broj preostalih odlaganja za taj proces i nastavljamo dalje.
Potom proveravamo da li postoje procesi bez afiniteta ili sa odgovarajućim afinitetom u listi spremnih procesa, ukoliko postoje kroz sledeće tri odluke biramo najprioritetniji od tih procesa[12].
Ukoliko ne postoje procesi sa odgovarajućim afinitetom, dolazi do preotimanja (raspoređuje se najprioritetniji proces bez obzira na njegov afinitet) ako i samo ako je broj procesa koji se izvršavaju i broja spremnih procesa veći od broja jezgara (o razlogu za postojanje ovog uslova će biti reči kasnije) u suprotnom se ne raspoređuje ni jedan proces i vraća se vrednost *null*.

Sistem za afinitet mora da balansira između iskorišćenja procesora, poštovanja afiniteta i poštovanja prioriteta kojima upravlja algoritam raspoređivanja[13].

Ova specifikacija sistema za afinitet balansira između poštovanja afiniteta i poštovanja prioriteta tako što uvek bira najprioritetniji proces (ili od svih procesa ili od podskupa procesa sa povoljnim afinitetom) i dodatno **preteranim odlaganjem** garantuje da će najprioritetniji proces biti raspoređen nakon maksimalno `AHinitialLaziness` rundi raspoređivanja.

Između iskorišćenja procesora i poštovanja afiniteta ovaj sistem prioritizuje iskorišćenje procesa postojanjem mogućnosti do dođe do preotimanja (raspoređivanja najprioritetnijeg procesa bez obzira na njegov afinitet). Zato je bitno jako pažljivo odrediti kada do preotimanja može doći. U ovom slučaju do preotimanja može doći samo ukoliko ne postoji ni jedan proces koji ima pogodan afinitet (afinitet prema jezgru koje dohvata proces ili uopšte nema afinitet). Ovaj uslov je baziran na pretpostavci da će prosečno vreme koje neki procesor provede besposlen[14] biti veće od vremena koje se izgubi zbog promašaja u keš memoriji.

Posmatranjem ovakve specifikacije uočeno je da postoje situacije u kojima ovakav uslov dovodi do pogoršanja performansi pa je potrebno dodati dodatne uslove za preuzimanje. U situaciji gde je broj izvršivih procesa[15] manji od broja jezgara možemo primetiti da će bar jedno jezgro biti besposleno i to besposleno jezgro će stalno preotimati procese čim se ubace u red spremnih čak i ukoliko je jezgro prema kome taj proces ima afinitet slobodno (npr. pri prekidu od strane tajmera sva jezgra mogu dodati svoje procese u red spremnih i potom ih opet dohvatiti) što će dovesti do velikog broja promašaja u keš memoriji i time nepotrebno usporiti sistem. Tako da je potrebno eksplicitno zabraniti preotimanje u takvoj situaciji. Dodatna posledica zabrane preotimanja je to da je **starenje** jedina tehnika koja sprovodi balansiranje u takvoj situaciji pa samim tim i parametar `AHmaxAge` diktira maksimalnu degenerisanost raspodele procesa (tj. maksimalan broj procesa koji mogu imati afinitet ka jednom jezgru) te se savetuje da ova vrednost bude relativno mala. Kako je jedini način da se garantuje optimalna raspodela procesa u ovakvoj situaciji to da

[12] Tri odluke su: da li postoji proces bez afiniteta (u slučaju da ne, biramo najprioritetniji sa odgovarajućim afinitetom), da li postoji proces sa odgovarajućim afinitetom (u slučaju da ne, biramo najprioritetniji bez afiniteta) i ukoliko postoje oba koji je prioritetniji (biramo prioritetniji).
[13] U daljem tekstu samo poštovanje prioriteta
[14] Vreme od trenutka kada više ne postoje pogodni procesi u redu spremnih to trenutka dodavanja pogodnog procesa u red spremnih.
[15] Procesi koji se izvršavaju ili su u redu spremnih.

parametar `AHmaxAge` ima vrednost 1 (što je manje od preporučene vrednosti za opšti slučaj), potencijalno unapređenje ovog sistema bi bilo da dok je ovaj uslov ispunjen parametar `AHmaxAge` automatski ima vrednost 1.

5.2 Implementacija sistema za afinitet

Na osnovu gore navedene specifikacije možemo razmotriti koja struktura podataka bi bila pogodna za efikasnu implementaciju.
Pre svega možemo da uočimo da nam je potrebno da održavamo jedan ili više prioritetnih redova sortiranih po prioritetu koji nam diktira algoritam raspoređivanja slično kao i kod implementacije bez afiniteta. Razlika je u tome da je ovde pored dohvatanja najprioritetnijeg procesa potrebno dohvatiti i najprioritetniji proces sa određenim afinitetom. Ukoliko održavamo više prioritetnih redova (po jedan za svako jezgro i jedan za procese bez afiniteta) dohvatanje najprioritetnijeg procesa sa određenim afinitetom je veoma efikasno dok je za dohvatanje globalno najprioritetnijeg procesa potrebno proći kroj najprioritetnije elemente svih prioritetnih redova. Kako bi ovo dodatno optimizovali možemo da oformimo još jedan prioritetni red čiji su elementi najprioritetniji elementi ostalih prioritetnih redova. Da bi ovakva struktura prioritetnog reda nad prioritetnim redovima mogla efikasno da se održava potrebno je dodatno znati na kojoj poziciji se nalazi neki proces u svakom od prioritetnih redova, te je potrebno dodati po jednu heš tabelu za svaki prioritetni red koja mapira indeks procesa u nizu `proc` u indeks tog procesa u tom prioritetnom redu (vrednost -1 označava da se taj proces ne nalazi u tom prioritetnom redu).

Pri implementaciji ove strukture podataka odlučeno je da se prioritetni redovi realizuju pomoću binarnih heap-ova kao i u implementaciji bez afiniteta[16], dok su heš tabele realizovane kao prosti nizovi gde se na poziciji indeksa u nizu `proc` nalazi pozicija u prioritetnom redu. Kako postoji veliki broj heap-ova i heš tabela one su spojene u matrice veličine $brojprocesora + 2$ tako da se na indeksu nula nalazi heap sa elementima drugih heap-ova[17], na indeksu jedan se nalazi heap procesa bez afiniteta a na indeksima većim od jedan se nalaze heap-ovi procesa sa afinitetima prema određenim jezgrima.

```
14 int heaps[NCPU+2][NPROC+1]={{0}, {0}, {0}, {0}, {0},
       {0}, {0}, {0}, {0}, {0}};
15 //HEAP 0 COVERING HEAP
16 //HEAP 1 NO AFFINITY HEAP
17 //HEAP 2 CPU 0 HEAP
18 //..
19 //HEAP X[0] SIZE
20 int backmap[NCPU+2][NPROC]={{0}, {0}, {0}, {0}, {0},
       {0}, {0}, {0}, {0}, {0}};
21 int totalSize=0;
```

Primer 16: Fajl:*affinityHeap.c* Strukture podataka potrebne sistemu za afinitet

[16] Heap-ovi su i ovde realizovani kao niz indeksa gde se na nultom indeksu nalazi broj elemenata.
[17] U daljem tekstu glavni heap.

Dodatna polja za potrebe sistema za afinitet

```
115   //core affinity data
116   int affinity;                 // Index of cpu process
        has affinity to
117   int affinityAge;              // Last scheduling
        round when process executed on a affinitized cpu
118   int laziness;                 // Number of times
        process can defer being scheduled on a non
        affinitized cpu
```

Primer 17: Fajl:*proc.h* Dodatna polja za potrebe sistema za afinitete

Polje `affinity` predstavlja indeks jezgra prema kome dati proces ima afinitet ili -1 ukoliko proces nema afinitet.
Polje `affinityAge` predstavlja broj runde raspoređivanja u kojoj se proces poslednji put izvršavao na jezgru prema kome ima afinitet.
Polje `lazines` predstavlja koliko još puta proces može da odloži svoje biranje, ukoliko ovo polje ima vrednost 0 smatra se da proces nema afinitet.

Dodatne strukture podataka za potrebe tehnike starenja
Da bi bilo moguće implementirati tehniku starenja potrebno je dodati još nekoliko struktura podataka. Pre svega potrebno je u jednom nizu pamtiti broj runde raspoređivanja za svako jezgro. Takođe je potrebno imati po jedan kružni bafer[18] za svako jezgro koji pamti `AHmaxAge` poslednjih procesa koji su se izvršavali na tom jezgru, tako da se provera uslova za starenje svodi na proveru da li je najstariji proces u kružnom baferu tog jezgra i dalje u heap-u tog jezgra i da li se izvršavao od tada[19] (ukoliko je i dalje u heap-u i nije se izvršavao od tada treba da izgubi afinitet).

```
23  int roundCounter[NCPU]={0,0,0,0,0,0,0,0};
24  int agingHeads[NCPU][NPROC];
25  int agingPointer[NCPU]={0,0,0,0,0,0,0,0};
```

Primer 18: Fajl:*affinityHeap.c* Strukture podataka potrebne za tehniku starenja

Ažuriranje heap-a obavlja funkcija `AHheapUpdate` tako da proces na poziciji `procIndex` u heap-u sa indeksom `heapIndex` dovodi na odgovarajuću poziciju i kao povratnu vrednost vraća novi indeks tog procesa. Sama funkcija je u velikoj meri slična funkciji za uklanjanje elementa iz heap-a korišćenoj u implementaciji bez afiniteta. Za razliku od te funkcije ova funkcija mora takođe da održava stanje u heš tabeli konzistentnim i osim spuštanja procesa kroz stablo mora pre toga da pokuša i da podigne proces ka korenu stabla.

Ubacivanje procesa se svodi na ažuriranje dodatnih polja u strukturi procesa, ubacivanje procesa u odgovarajući heap i ukoliko je potrebno (ubačeni proces je najprioritetniji u svom heap-u) ažuriranje glavnog heap-a. Pri ažuriranju glavnog heap-a razlikujemo dva slučaja u zavisnosti da li je heap u koji ubacujemo pre ubacivanja bio prazan (potrebno je ubaciti element u glavni heap) ili ne (potrebno je samo ažurirati glavni heap).

[18] Kružni baferi su implementirani kao matrica sa još jednim nizom pokazivača na najstariji element u kružnom baferu.
[19] Ovo se proverava uslovom `proc.affinityAge != roundCounter[core] - AHmaxAge`, ukoliko je uslov tačan proces se izvršavao od tada.

```
212 void AHput(int procIndex)
213 {
214     totalSize++;
215     int heapIndex=proc[procIndex].affinity+2;
216     proc[procIndex].laziness=AHinitialLaziness;
217     heaps[heapIndex][++heaps[heapIndex][0]]=procIndex;
218     backmap[heapIndex][procIndex]=heaps[heapIndex][0];
219     int lproc=heaps[heapIndex][1];
220     if(AHheapUpdate(heapIndex,heaps[heapIndex][0])==1)
221     {
222         if(heaps[heapIndex][0]==1)
223         {
224             heaps[0][++heaps[0][0]]=procIndex;
225             backmap[0][procIndex]=heaps[0][0];
226             AHheapUpdate(0,heaps[0][0]);
227         }
228         else
229         {
230             heaps[0][backmap[0][lproc]]=procIndex;
231             backmap[0][procIndex]=backmap[0][lproc];
232             backmap[0][lproc]=-1;
233             AHheapUpdate(0,backmap[0][procIndex]);
234         }
235     }
236 }
```

Primer 19: Fajl:*affinityHeap.c* Ubacivanje procesa

Dohvatanje procesa je znatno složenije od ubacivanja i započinje prolaskom kroz stablo odlučivanja koje će izabrati proces koji se dohvata i potom dekrementira polje `laziness` najprioritetnijeg procesa u glavnom heap-u. Nakon toga je potrebno ažurirati sve heap-ove, prvo se proces uklanja iz heap-a kome pripada i pritom se vodi računa o ažuriranju heš mape ukoliko je to nije bio jedini element u heap-u. Sledeći korak je ažuriranje glavnog heap-a, gde postoji veliki broj ivičnih slučajeva u zavisnosti od toga da li se neki od heap-ova ispraznio.

```
120     //UPDATING
121     int heapIndex=proc[ret].affinity+2;
122 
    heaps[heapIndex][1]=heaps[heapIndex][heaps[heapIndex][0]];
123     backmap[heapIndex][ret]=-1;
124     if(heaps[heapIndex][0]!=1)
125     {
126         backmap[heapIndex][heaps[heapIndex][1]] = 1;
127     }
128     heaps[heapIndex][0]--;
129     AHheapUpdate(heapIndex,1);
130 
131     int mainIndex=backmap[0][ret];
132     if(heaps[heapIndex][0]==0)
133     {
134         heaps[0][mainIndex]=heaps[0][heaps[0][0]];
135         heaps[0][0]--;
136     }
137     else
138     {
139         heaps[0][mainIndex]=heaps[heapIndex][1];
140     }
141     backmap[0][ret]=-1;
```

```
142     if(heaps[0][0]!=0)
143     {
144         backmap[0][heaps[0][mainIndex]] = mainIndex;
145     }
146     if(mainIndex<=heaps[0][0])
147     {
148         AHheapUpdate(0, mainIndex);
149     }
150 
151     totalSize--;
152     roundCounter[core]++;
```

Primer 20: Fajl:*affinityHeap.c* Ažuriranje heap-ova pri dohvatanju procesa

Nakon ažuriranja heap-ova u varijanti sa starenjem dolazi do starenja i na kraju se ažuriraju dodatna polja u strukturi procesa u kružni baferi za tehniku starenja.

```
203 
204     //Setting parameters
205     proc[ret].affinity=core;
206     proc[ret].affinityAge=roundCounter[core];
207     agingHeads[core][agingPointer[core]]=ret;
208     agingPointer[core]=(agingPointer[core]+1)%AHmaxAge;
```

Primer 21: Fajl:*affinityHeap.c* Ažuriranje dodatnih polja i kružnih bafera

Tehnika starenja je implementirana nakon održavanja stanja heap-ova pri dohvatanju procesa. Prvo se proverava da li postoji proces na najstarijem mestu u kružnom baferu i da li je to njegovo poslednje izvršavanje. Ako su ti uslovi ispunjeni potrebno je dodatno proveriti da li se taj proces i dalje nalazi u heap-u tog jezgra, ukoliko se nalazi potrebno je oduzeti mu afinitet. Ukoliko se taj proces nalazi na vrhu svog heap-a dovoljno je samo smanjiti njegov `laziness` na 0 čime on efektivno gubi afinitet. U suprotnom je potrebno ukloniti ga iz heap-a tog jezgra (nema potrebe za ažuriranjem glavnog heap-a jer smo sigurni da se ne nalazi na vrhu), ažurirati mu polje `affinity` i ubaciti ga heap procesa bez afiniteta[20].

Ostale funkcije sistema za afinitet su ili analogne funkcijama implementacije bez afiniteta ili imaju proste implementacije pa one neće ovde biti detaljnije razmatrane.

[20] Proces ubacivanja u heap je objašnjen u delu o ubacivanju procesa te ovde neće biti ponovljen

6 Zaključci

6.1 Stabilnost

Kako se od operativnih sistema očekuje da budu veoma stabilni bilo je neophodno ekstenzivno testirati sve izmene operativnog sistema a pogotovu delove koda i strukture podataka od kojih zavisi celokupan raspoređivač tako da je njima posvećenja posebna pažnja. Celokupan operativni sistem je manuelno testiran na otkaze i neočekivana ponašanja (uvidom u trag raspoređivača je provereno da li rezultat raspoređivanja odgovara implementiranom algoritmu) u svakom stanju u kome se raspoređivač može nalaziti i to tako da su pokriveni i svi prelazi između stanja.

Heap korišćen za implementaciju strukture podataka koja ne uzima u obzir afinitet je jedinično testiran upoređivanjem stanja sa naivnom implementacijom na velikom broju nasumično generisanih test slučaja. Sistem za afinitet se u velikoj meri bazira na kodu implementacije koja ne uzima u obzir afinitet te je bilo potrebno testirati samo modifikacije i nadgradnje tog koda. Zbog kompleksnosti tih modifikacija i nadgradnji sistem za afinitet nije jedinično testiran već je testiran pokretanjem u test modu koji je nakon svake modifikacije struktura podataka proveravao njihovu konzistentnost.

Nakon uspešno sprovedenih gore navedenih testova smatram da je ova implementacija sistema za raspoređivanje dovoljno stabilna za potrebe ovog projekta, dok je za neke konkretnije primene potrebno dalje testiranje.

6.2 Performanse

Performanse sistema nisu detaljno testirane ali su preliminarna testiranja pre odbrane pokaza da su performanse unutar margine greške u odnosu na nekoliko drugih radova. Još značajnije preliminarni testovi performansi su pokazali da korišćenje sistema za afinitet sa starenjem ne utiče negativno na performanse u odnosu na raspoređivanja bez afiniteta ili korišćenje sistema za afinitet bez starenja. Preliminarni testovi su takođe pokazali da se određena opterećenja dobro skaliraju sa brojem jezgara.

Na samoj odbrani uočene su čak i znatno poboljšane performanse privatnog testa pri korišćenju sistema za afinitet na sistemu sa 4 jezgra.

U svetlu ovih rezultata smatram da je sistem za afinitet pokazao utemeljenost u praksi i performanse dovoljno dobre za potrebe ovog projekta iako bi za detaljnije zaključke i diskusiju o potencijalnim unapređenjima bilo potrebno dalje testiranje.