9. Управление на процесите

Васил Георгиев

is.fmi.uni-sofia.bg/t3/

v.georgiev@fmi.uni-sofia.bg

Съдържание

- Процеси и нишки
- Мултипроцесинг в UNIX
- Миграция на код
- Идентификация на обекти
- Garbage collection

Процеси

- в ОС процесите са системни и потребителски програми, допуснати до изпълнение, за които динамично се планират необходимите операционни (процесорно време, памет, В/И и др.) и комуникационни ресурси
- планирайки, ОС създава за всеки процес виртуален процесор и позиция в таблицата на процесите с регистърен буфер, карта на процесната памет и на отворените файлове, приоритети, процесно счетоводство и др. – също и за междупроцесна защита
- създаването/превключването на процеси (процесен контекст) е сериозен системен свръхтовар – напр.:
  - алокация на сегмент за данни (евентуално нулиран)
  - зареждане на кодовия сегмент, алокация/зареждане на стека, на регистрите (процесорни р-ри, програмен брояч, стеков указател, MMU и TLB регистри)
  - управление на `swap` операции между основната и външната памет (при мултипрограмиране с повече процеси)

Паралелни процеси

- паралелизма (грануларността) е на ниво програма и процедура
- това ниво съответства на мултипроцесинга, тъй като всяка програма е отделен процес
- при SPMD модел (напр. в UNIX) с примитива `fork` се създава реплика на изпълняващия процес:

```
Proc-id = Fork()
```

(създава се нова реплика на процеса и й се присвоява идентификатор)

- двата процеса (родител и наследник) се различават само по стойността на `Proc-id`, в наследника тя е 0:

```
Proc-id = Fork()
if Proc-id = 0
        then do {child processing}
        else do {parent processing}
```

- други примитиви от тази група са `exit` за прекратяване на процеса наследник и `wait` – за синхронизация (процесът-родител блокира до завършване изпълнението на наследника)
- при процедурен паралелизъм на системно ниво процедурата се асоциира с отделен процес

```
Proc-id = new process(A_procedure)
kill process (A_procedure)
```

Паралелизъм на ниво израз

- това ниво е свързано с езикови спецификации (примитиви за паралелно изпълнение на инструкции)
- напр. примитивът Parbegin/Parend задава бпок от изрази за паралелно изпълнение, по време на което главният процес блокира
- пример – изразът (a+b)*(c+d)–e/f може да бъде изпълнен конкурентно със следната спецификация (псевдокод):

```
Parbegin
  Parbegin
    t1 = a + b
    t2 = c + d
  Parend
  t4 = t1 * t2
  t3 = e/f
Parend
t5 = t4 – t3
```


Паралелен израз в UNIX

- паралелизъм на израз с примитивите `fork-join-quit`:
- `fork label` предизвиква създаване на нов процес-наследник, чието изпълнение стартира от даден етикет (така наследника и родителя може да не са реплики):
- `quit` е примитив за прекратяване на текущия процес
- `join t,lab` е примитив със следната интерпретация:

```
t = t – 1
if t = 0 then go to lab
```

- пример за изчисление на горния израз

```
n = 2
m = 2
Fork P2
Fork P3
P1: t1 = a + b; Join m, P4; Quit;
P2: t2 = c + d; Join m, P4; Quit;
P4: t4 = t1 * t2; Join n, P5; Quit;
P3: t3 = e/f; Join n, P5; Quit;
P5: t5 = t4 – t3
```


Паралелно програмиране в UNIX

- най-разпространената ОС за паралелни системи
- процесите се управляват чрез системни заявки (calls):
- създаване: използва се заявката `fork()` за репликиране на текущия процес-родител
- планиране и контрол – напр. с използване на системния таймер – функциите `timer-init()` и `timer-get()` (в микросекунди) или с използване на семафори
- междупроцесен обемен – чрез алоциране на общи променливи със заявката `Share()`
- паралелните приложения се разработват най-често на C с използване на библиотеката `parallel.h` и се компилират с опция -lpp за зареждане на паралелната библиотека:

```
cc program -lpp
```


Многопроцесно приложение в UNIX

- за вътрешна идентификация на процесите често се прилага и индексиране: пример – функция `Mkps()` за създаване на *n* процеса-наследници със стойност на `ppid` 0 в процеса родител и от 1 до *n* в наследниците:

```
Mkps(n)
  int n;
  {
  int i;
  for {i = 0; i<n; i++}
    {
    switch (fork())
      {
      case 0:  /* process creation*/
        return(i+1);
      case -1:   /* cannot create process*/
        {
        printf("Cannot create process %d\n", i)
        return -1; }
      default:  /* goto next creation */
      }
    }
  return 0;
 }
```

Паралелно програмиране в UNIX

- шаблон на паралелната програма:

```
main (argc, argv)
  int argc;
  char * argv[];
  {
  int ppid, procs;
  scanf(argv[1], "%d", &procs);
  ppid = Mkps(procs); /* creation of argv[1] number of processes*/
  switch (ppid)
    {
    case 0: { /* parent process code */}
    case 1: { /* child1 process code */}
    case 2: { /* child2 process code */}
    ...
    case n: { /* childn process code */}
    default:
      {
      printf("Program error"); break;
    } }
  /* termination of the children: */
  if (ppid != 0)
    {
    printf("child # %d terminates\n", ppid);
    exit(ppid);
  }   }
```


Обмен между процесите

- UNIX няма други средства за деклариране на общи ресурси между потребителските процеси освен общи променливи, чийто тип зависи от използвания език
- променливата или структурата, която е с общ достъп, се декларира съгласно езиковия стандарт:

```
struct  SharedData
        {
        int x, y, z;
        float a, b;
        char* name
        } mySharedRecord, *toMySharedStruct;
```

- всяка [вече] декларирана променлива може да бъде обявена за общ достъп (и алоцирана в общ сегмент от паметта) със системната заявка `Share()`:

```
toMySharedStruct = Share(0, sizeof(mySharedRecord));
```

- резултатът е, че освен декларираща процес, всички негови наследници (създадени след нейната декларация като обща) имат достъп до съответната променлива

Времево планиране на процесите

- времевото планиране е частен случай на планирането по събитие, при който събитието и изтчане на таймер;
  - ползва се системния часовник с импулси на всяка микросекунда; з
  - заявка към системата `timer-init()` стартира (нулира) локален брояч за процеса, а заявката `timer-get()` връща текущата му стойност в микросекунди
  - пример: паралелни процеси с локални променливи за времето

```
main (argc, argv)
  int argc;
  char * argv[];
  {
  double ProcessTime;
  long timer;
  int ppid, procs;
  scanf(argv[1], "%d", &procs);
  ppid = Mkps(procs); /* creation of argv[1] number of processes*/
  switch (ppid)
    {
    case 0:  /* parent process code */
      {
      timer-init()            /* set the clock */
      timer = timer-get();    /* get current time */
      ProcessTime = (timer-get() - timer)/1000000.0;
                            /* get execution time in sec */
      break;
    }
.......
```


Синхронизация с взаимно изключване между процесите в UNIX

- променливи от тип `lock` осигуряват монополен достъп на извършваните върху тях операции за даден процес
- специфичните операции за този тип са l`ockname_create` и `lockname_init`, където `lockname` е множество от следните типове: LOCK, BARRIER, SEMAPHORE и EVENT
- LOCK е тип данни, с който е асоцииран атрибут със стойности PAR_LOCKED и PAR_UNLOCKED и се реализира класическия подход за взаимно изключване; с този тип са асоциирани и операциите `lockname_lock` и `lockname_unlock`
- BARRIER е тип данни, съставен от двойката `(count, flag)` , където `count` задава броя процеси, чието изпълнение трябва да достигне до съответния обект-бариера, преди да продължат; `flag` задава режима на изчакване:
- `flag = SPIN_BLOCK`: блокировка с циклично изчакване
- `flag = PROCESS_BLOCK`: блокировка при достъп до данни
- EVENT е тип данни, съставен от двойката `(event, flag)`, където `event` задава събитие, което трябва да се изпълни преди процеса да продължи (възможно е повече от един процес да чака това събитие); `flag` задава режима на изчакване като при BARRIER
- SEMAPHORE е тип данни, асоцииран с двойката атрибути `(count, flag)`, където `count` задава броя процеси, които имат право на достъп до променливата преди заключването ѝ; `flag` задава режима на изчакване като при BARRIER; с този тип е асоциирана операцията `semaphore_set` за `count`

Особености процесите в разпределените системи

- ефективното планиране на разпределените приложения (предимно по модела клиент-сървер) с прилагане на многонишков подход (multithreading) за припокриване (overlapping) примерно на комуникационните фази с фазите на локална обработка на отделните процеси;
- разлики в планирането при клиентски и сървърни машини както и между сървърите с различно предназначение (напр. обработващи, файлови, комуникационни, за разпределени обекти и др.)
- възможности за мигриране на процеси особено в хетерогенна среда и необходимата динамична реконфигурация на клиенти и сървери (процеси)
- прилагане на обработка с процеси-агенти – равнопоставени процеси за обслужване (вместо асиметричния модел клиент-сървер)

Нишки

- подпроцесите – традиционно нишки (threads) – са средство за постигане на по-фина грануларност респ. по-оптимално планиране
- при нишките е недопустим свръхтовар като при процесите ⟶ по-слаба конкурентност и защита: нишковия контекст се състои примерно от CPU-контекста и текущ статус (напр. блокировка поради синхронна комуникация); така че защитата на нишковите данни в рамките на процеса зависи от кодирането на многонишковото приложение (⟶ по-сложно програмиране)
- многонишково програмиране се прилага и при унипроцесорни приложения –
  - пример: електронна таблица с отделни нишки за потребителски интерфейс и за обработка на формулите
- многонишковата програма за унипроцесор е преносима и за паралелна обарботка
- многонишковите програми са по-удобни за настройка – пример: текстов редактор с отделни нишки за UI, граматическа проверка, форматиране, генерация на съдържание и т.н.

Видове нишки

- в разл. ОС се прилагат нишки в потребителски режим или в режим на ядрото
- при **нишки в потребителски режим** се ползват програмни пакети за многонишкови програми с операции за деклариране на нишките `(create, destroy)`, за синхронизация достъпа до общи променливи – `mutex` (ключалка като семафорите с решаване на блокировката чрез приоритети или FIFO)
- по-нисък системен свръхтовар – без операции върху паметта: при създаване/закриване само заделяне и освобождаване на стека и при превключване - само замяна на стойностите в ЦПУ регистрите
- недостатък: блокирането на една нишка (напр. по В/И) блокира целия процес – т.е. елиминира основно преимущество на многонишковия процес
- **нишките в режим на ядрото** са компоненти на системната библиотека и се изпълняват като процеси на ядрото – създаването и превключването им са с обръщения към системата – преодолява се тоталното блокиране, но свръхтовара е съпоставим с процесния

Леки процеси

- LWP (lightweight process) – хибриден подход – леките процеси се изпълняват като обикновени процеси; един процес може да включва няколко LWP
- същевременно се ползват и пакетите за многонишкови програми при които нишковите операции са в потребителски режим
- многонишковите приложения създават необходимите нишки (потр. режим) и предават [имплицитно] изпълнението им на LWP – фиг. 9.16.
- LWP се създават с обръщение към системата и се асоциират с някоя от активните нишки (съгласно диспечерска процедура)
- изпълнението на "двойката" системен LWP и потребителска нишка протича предимно в потребителски режим – LWP се превключва в контекста на нишката и напр. синхронизацията с `mutex` се изпълнява в потр. режим
- при блокиране на нишка (поради блокиращо обръщение към системата) управлението на сдвоения подпроцес се връща в режим на ядрото/LWP, а междувременно системата предава управлението на друг LWP (който ако не е блокиран, минава в режим на асоцииранатата с него нишка – т.е. потребителски)
- системният свръхтовар е редуциран (в потребителски многонишков режим) и изпълнението на целия процес е свободно от блокировка
- LWP са прозрачни за кода; преносимост за унипроцесорно и паралелно програмиране (във втория случай леките процеси на едно приложение се изпълняват на различни процесори)

Многонишкови клиентски процеси

- обикновено постигат маскиране на комуникационните и синхронизационни закъснения на някои нишки чрез изпълнение на други
- пример: Уеб браузерите (клиент в интерактивен режим) изобразяват веднага заредените елементи и постепенно попълват страницата – след зареждане на [част от] основната страница (най-често текст) се активира нишка за неговото изобразяване, плъзване (scroll), избор и др. функции и друга нишка/и за блокиращото зареждане на по-бавните компоненти (за блокираща заявка към ОС за връзка със съответния сървер/и);
- при повече от една комуникационна нишка се постига паралелизам и на комуникациите/зареждането на останалите компоненти (но само ако сървера разполага със съответна производителност – напр. репликирани сървери (т.е. един адрес но реплики на страниците на няколко машини, които се асоциират прозрачно със заявките на отделните нишки напр по Round Robin)

Многонишкови сървери

- многонишковите сървери обикновено се конструират с нишка-диспечер, която получава всяка нова заявка за обслужване и я асоциира с някоя от изчакващите изпълнителни нишки – фиг. 9.18.
- пример: при файлов и документен сървер еднонишков обслужващ процес ще изпълнява заявките последователно – вкл. и закъснението за достъп до вторичната памет
- многонишковите "диспечер-изпълнител" процеси се базират на блокиращи обръщения към системата в изпълнителните нишки

Миграция на код

- среща се под формата на:
  - миграция на процеси – напр. за балансиране на локалния изчислителен товар между възлите (измерван напр. с дължина на локалната опашка от заявки, натоварване на процесора/обсл. устройство и др.)
  - мигриране на програми за отдалечено изпълнение –
    - при сървера – напр. зареждане в сървера на програма за локална обработка на данни и връщане само на резултата (вместо зареждане на данните при клиента)
  - при клиента – напр. зареждане в клиента на програма за попълване параметрите на заявка и връщането ѝ към сървера (вместо интерактивен обмен със сърверен процес за попълване на заявката)
- миграцията на процес изисква преместване на сегмента код, сегмента данни и сегмента изпълнение (т.е. статус)
  - при сегмента данни: процес свързване (binding) т.е. настройка на адресните аргумент (данни); варианти:
    - свързване по идентификатор – напр. при мигрирането на данни, които са адреси на файлове с URL идентификация (понеже идентификатора е универсален)
    - свързване по стойност – напр. адресиране на стандартна библиотека в C и Java (действителния им идентификатор е локален)
    - свързване по тип – напр. адресиране на локални устройства (принтери, монитори)

Модели за миграция на код

- ниска (weak) мобилност – само на сегментите код и данни; изпълненото стартира отначало – пример: Java аплетите (изисквания за преместваемост на кода)
- висока (strong) мобилност – + сегмента на статуса;
- по инициатива на изпращащия процес – примери: изпращане на програма за изпълнение от изчислителен сървер (изпр. п-с е клиент; за защита е необходима идентификация на клиента) или изпращане на процес за балансиране на товара при групово обслужване (изпр. п-с е сървер)
- по инициатива на приемащия процес – Java аплети (прием. п-с е клиент) или отново за балансиране но при инициатива на приемащ сървер

Миграция на код в хетерогенна среда

- при ниска мобилност (само на код и данни) е необходима прекомпилация на програмата за различни машини/ОС – напр. изпращащия процес има различен изпълним код за всяка от възможните приемащи платформи
- при висока мобилност (код, данни и статус) – с поддържане на машиннонезависим миграционен стек в определени точки на програмата, (в които и само в които може да се извърши миграцията)
  - в процедурните езици (C) това е след изпълнението на текуща функция/метод и преди стартирането на следващ/а – за да не се налага примарно пренос на стойностите на процесорните регистри, които са машиннозависими
- с интерпретирани езици – при скриптовите езици виртуалната машина директно интерпретира програмния код (Tcl) или универсален междинен код, генериран от компилатор (Java)

Имена, адреси и идентификатори

- **имената** са символни низове за идентификация на компоненти – ресурси (възли, устройства вкл. вторични памети, файлове) и обслужвани компоненти (процеси, потребители, съобщения, документи, нюзгрупи, мрежови съединения и др.)
- именуваните компоненти подлежат на управление или промяна посредством съответни точки за достъп – адреси
- в РС са широко застъпени **динамичните адреси** → имената са по-удобни за идентификация на повечето компоненти отколкото динамичните адреси
- същото важи и за **множествените адреси** – един компонент с няколко адреса (точки за достъп) се идентифицира с име, но не и с един от адресите си; пример – разпределена Web услуга, изпълнявана от няколко сървера с различни адреси
- при имената и адресите се допуска моногозначност и промяна
- за прозрачна идентификация се използват **адреснонезависими имена**
- **идентификаторите** са имена, които имат еднозначно-обратимо и устойчиво съответствие с компонентите:
  - всеки идентификатор съответства най-много на един компонент
  - всеки компонент има не повече от един идентификатор
  - идентификаторите не се подменят или пренасят на други компоненти
- *идентификаторите осигуряват лесно сравняване* на идентичността на компонентите (за разлика от имената и адресите поради тяхната многозначност и преходност)
- *имената (когато са потребителски-ориентирани) са по-удобни за потребителите* (отколкото машинно-ориентираните идентификатори и адреси)

Пространство на имената и разрешаване на имената

- пространството на имената се представя с маркиран насочен граф, в листата и върховете на който са разположени имената на компонентите;
- имената във върховете са на компоненти-директории (т.е. списъци с имена и адреси на наследниците)
- обикновено дървото на имената има само един корен
- път в графа на имената – абсолютен (от корена) и относителен път
- графът на имената обикновено е дърво (само с едно входящо ребро за всеки възел – връх, листо) или е ацикличен
- решаване на имената (name resolution) е извличането на идентификатор на компонента при зададено име (и път)
- псевдоним (alias) е допълнително име на компонент:
  - когато графът на имената допуска повече от един път до компонента – пример в UNIX (фиг. 9.23)
  - когато съдържанието на възел-лист от графа на имената не е име на компонент а абсолютен път до името на този компонент
- свързване (mounting) на две пространства имена се реализира като b) – възел от едно пространство (mount point) съдържа идентификатор на възел от друго пространство (mounting point)

Разслоено пространство на имената

- при големите/глобалните РС пространството имена се организира йерархично чрез разслояване, поддържайки общ корен
- обикновено се приема трислоен модел:
  - глобално ниво (global layer) – корена на графа и свързаните с него възли-директории; на това ниво промените на имена са много редки (най-висока стабилност), отделните възли съдържат списък с имена от следващото ниво, групирани по организационен принцип (напр. в DNS областите com, edu, gov, mil, org, net, и на страните)
  - административно ниво (administration layer) – възлите-директории съдържат списъци с компоненти, принадлежащи на обща административна област (напр. списък с отделите на една организация или списък със хостовете в даден интранет или списък на всички потребители от тази област) – относителна стабилност (в DNS sun.com, uni-sofia.bg, fmi.uni-sifia.bg, acm.org)
  - локално ниво (managerial layer) – възлите-директории представят локални компоненти – напр. файловата система на отделени хостове в дадена локална мрежа и отделни локални директолии и файлове за общ достъп – ниска стабилност; поддръжката на такива възли-директории се извършва и от потребителите (в DNS courses.fmi.uni-sofia.bg)
- освен йерархично, пространството имена се разделя и административно на непripокриващи се части – зони – всяка от които се обслужва от съответен сървер на имената

Domain Name System

- DNS е най-голямата разпределна система за имена на компоненти, на която се базира Интернет
- йерархична (т.е. дървовидна) организация на възлите, което позволява ползването на общ етикет за [единственото] входящо ребро и за възела
- етикетите се означават със символни низове без различаване на главни и малки букви до 63B, а с абсолютния път – до 255B
- абсолютният път се отчита от корена и се означава с ".", която може да се пропусне – courses.FMI.uni-Sofia.bg.
- област (domain) е поддърво в DNS, абсолютният път до нея е името на областта
- съдържанието на възела (т.е. интерпретацията на именования компонент) се задава с асоцииран към него списък от ресурсни записи:

Ресурсни записи

```
[RFC1035]
A          1  a host address
NS         2  an authoritative name server
MD         3  a mail destination (Obsolete - use MX)
MF         4  a mail forwarder (Obsolete - use MX)
CNAME      5  the canonical name for an alias
SOA        6  marks the start of a zone of authority
MB         7  a mailbox domain name (EXPERIMENTAL)
MG         8  a mail group member (EXPERIMENTAL)
MR         9  a mail rename domain name (EXPERIMENTAL)
NULL       10 a null RR (EXPERIMENTAL)
WKS        11 a well known service description
PTR        12 a domain name pointer
HINFO      13 host information
MINFO      14 mailbox or mail list information
MX         15 mail exchange
TXT        16 text strings
```


DNS имплементация

- DNS прилага трислоен модел като поддържа глобалното и административното ниво (локалното ниво е файловата система на възлите)
- зоните се поддържат от [репликирани] сървери на имената
- съответствие: между области и зони
  - когато областта е изградена като една DNS зона, в зоновия файл няма сървери на имената в други зони
  - когато областта съдържа подобласти, които са в отделни зони, зоновия файл съдържа запис с името на подобластта, нейния DNS сървер и неговия адрес (вж. жълтия блок в следващия пример)

Област с подобласт

```
[amigo.acad.bg]
acad.bg.          SOA    amigo.acad.bg vedrin.acad.bg.  (200310210128800)
acad.bg.          NS     server = amigo.acad.bg
acad.bg.          NS     server = unicom.acad.bg
acad.bg.          NS     server = ns1.univie.ac.at
croom8            A      194.141.0.97
croom9            A      194.141.0.98
art               NS     server = amigo.acad.bg
art               NS     server = unicom.acad.bg
vtu               NS     server = ns.vtu.acad.bg
ns.vtu            A      194.141.4.1
vtu               NS     server = amigo.acad.bg
vtu               NS     server = unicom.acad.bg
muvar             NS     server = asclep.muvar.acad.bg
asclep.muvar      A      212.39.81.180
muvar             NS     server = dpx20.tu-varna.acad.bg
dpx20.tu-varna    A      194.141.24.4
muvar             NS     server = unicom.acad.bg
muvar             NS     server = amigo.acad.bg
gateN             A      194.141.0.212
dis               A      194.141.0.26
```

Итеративно решаване на адресите

- при итеративното решаване на адресите пълното име (с път) – напр. ftp://is.fmi.uni-sofia.bg/t3/rlTa1.pdf – се предава на сървера на имената в корена (адресът на чиято реплика е преконфигуриран локално)
- коренът решава обикновено само най-външната област т.е. връща адреса на сървер на имена, който я обслужва (в случая .bg)
- процесът продължава надолу по йерархията, докато се стигне до сървер на имена, който връща адрес на протоколен сървер (адреса на файловата система, поддържаща съответния документ или файл – тук ftp) – фиг. 9.29
- DNS-фазата от решаването на адреса се обслужва при клиента от специален процес – name resolver, а последната стъпка с протоколния обмен се изпълнява от друг клиентски процес

Рекурсивно решаване на адресите

- при рекурсивно решаване на адресите пълното име – напр. ftp://is.fmi.uni-sofia.bg/t3/rlTa1.pdf – се предава отново на сървера на имената в корена
- сървърът на имена не връща решения адрес (на следващ сървер) към клиента, а вместо това предава остатъка от името към този адрес/сървер
- стъптака се повтаря, докато не бъде решен адреса на протоколния сървер, който се връща обратно по йерархичната верига към корена
- решеният адрес се предава към клиентския процес от корена, след което отделен клиентски процес обслужва протоколния обмен с така решения адрес – фиг. 9.30
- предимството на рекурсията е съкращаване на комуникациите (статистически) и по-добра възможност за локално кеширане на адресните решения
- недостатък е централизацията на решаването в сървера на корена –
- затова DNS прилага на глобално ниво итеративния подход, а на административно – рекурсивния

Премахване на неадресираните компоненти

- Garbage collection – в РС обръщението към отдалечени компоненти се базира на локални указатели към тях; отсъствието на такива указатели означава че компонента трябва да се премахне, но наличието им не винаги означава актуалност (напр. циклични указатели между два ненужни компонента)
- при разпределените обекти двойката proxy-skeleton: прокси-стъб обслужва клиентския интерфейс към обекта, а скелетон-стъб – сървърния; обикновено тези две стъб-части обслужват разчистването, защото
  - разполагат с информация за текущите обръщения
  - могат да маскират тази системна функция от клиентския и сървърния процес
- граф на указателите с множество на корените, които не се премахват дори и когато няма указатели към тях – напр. потребители, системни услуги – фиг. 9.31.
- компонентите, които не са пряко или косвено достижими от множеството корени, подлежат на премахване
- поддържането на граф на указателите и на списък с недостижимите компоненти в РС се осъществява с модел на комуникации, съобразен с изисквания за ефективност и скалируемост

Броене на указателите

- асоциира статуса на обекта (компонента) с брояч на указателите (напр. клиентски стъбове) към него със съответното инкрементиране и декрементиране; обект с нулев брояч подлежи на премахване; броячът на указателите се поддържа обикновено от скелетон-стъба на обектния сървер – фиг. 9.32
- при РС този подход (приложен без модификации) поражда проблеми поради комуникационни закъснения и загуби – напр:
  - дублиране на инкрементиращи и декрементиращи съобщения, поради загуба на потвърждения от сървера
  - при наследяване (копиране) на указател към друг клиентски процес инкрементиращото съобщение на новия указател може да закъснее след декрементиращото към 0 съобщение за стария указател
- за преодоляване на комуникационните проблеми се прилага броене на теглото на указателите (weighted reference counting), което преодолява проблема с размножаването на указатели при репликиране на клиентските обекти чрез присвояване на [равна] част от теглото на своя указател на всеки новосъздаден указател
- друг подход е броенето на генерациите указатели (generation reference counting), при който освен брояч на поредните указатели се асоциира и с брояч на генерацията: ако напр. клиентски обект от k-генерация създаде n нови обекта (които се явяват k+1 генерация), след което изтрие своя указател, скелетонът в обектния сървер отразява G(k) = G(k)-1 и G(k+1) = n.

Списък на указателите

- принципно различен подход за garbage collection е вместо да се броят указателите, скелетонът да регистрира прокси-стъбовете, които извикват обекта, в списък на указателите (reference list) с идемпотентни операции за включване и изключване (мощността на всяко прокси в списъка е 1)
- допълнително предимство на идемпотентността е, че заявките могат да се изпращат няколкократно (напр. за отказоустойчивост) без да се променя резултата в списъка – което не е валидно при броячите
- този метод се прилага в Java RMI – при отдалечено обръщение към обект викащия го процес изпраща на скелетона своя идентификатор и след получаване на потвърждение [за включване в списъка указатели] процесът зарежда обектното прокси в адресното си пространство
- ако отдалечен процес П1 предаде копие от обектното прокси на друг п-с П2, П2 изпраща заявка/и за включване в списъка на скелетона и инсталира прокси-стъба след потвърждение
- проблем при горния сценарий: заявка от П1 до скелетона за изключване от списъка преди П2 да заяви включване – ако списъка междувременно стане празен, скелетонът може да изтрие обекта; срещу това се прилага заявка от П1 (също с потвърждение към П1) за предстоящо включване на П2, така че скелетонът поддържа списък на текущите и на предстоящите заявки

Недостижими компоненти

- недостижими компоненти (подлежащи на изтриване) са компонети без път от указатели към тях от някой корен
- те не се засичат по никой от горните методи, а чрез проследяване (tracing-based garbage collection) – проследяване на указателите към всички компоненти (метод с ниска скалируемост!)
- при **унипроцесорите** проследяването се прави по метода mark-and-sweep:
  - с фаза на маркирането на достижимите от корените компоненти и
  - фаза на изчистването, при която системата открива в паметта компоненти, нефигуриращи в маркирания списък, които се изтриват
- вариант: компоненти с открит указател към тях, но преди да е извършено проследяване на техните указатели, се маркират междинно като “сиви” (традицинно “бели” са компоненти, към които не са открити указатели, а “черни” са достижими компоненти, за които проследяването е завършило)

Mark-and-sweep за разпределени системи

- всеки п-с Пi стартира собствен колектор, който оцветява прокси- и скелетон-стъбовете, както и самите обекти с Б, Ч и С в следните стъпки:
- първоначално всички компоненти са оцветени с Б
- обекти от адресната област на Пi, които са достижими от Пi (явяващ се локален корен), се оцветяват С, също така се оцветяват и прокси-стъбовете, заредени от този обект; което означава че техните разпределени обекти са също С
- до скелетоните съответстващи на “сивите” прокси-стъбове се изпраща съобщение, което оцветява С самите скелетони и техните обекти (скелетоните и техните обекти са отдалечени по отношение на оцветяващия колектор на Пi)
- прокси-стъбовете, заредени от отдалечен обект, оцветен С, също стават С; тогава отдалеченият обект и неговия скелетон-стъб стават Ч и скелетонът връща съобщение на адресиращите го прокси-стъбове
- прокси-стъбовете, получили това обратно съобщение се оцветяват Ч
- колекторите продължават рекурсивно до завършване на оцветяването т.е. до оцветяване с Б и Ч (накрая няма С-компонети няма)
- втората фаза е премахване на всички Б-компоненти: обекти, скелетони и прокси-стъбове, (заредени от Б-обектите или асоциирани с тях)

Условие за проследяване

- методът mark-and-sweep изисква графа на достижимост да не се променя докато трае оцветяването и изтриването – т.е. спиране на изпълнението на процесите (“stop-the-world”); в разпределен вариант това означава, че всички процеси трябва да синхронизират моментите на стартиране на проследяването и на след това на възстановяване на изпълнението си
- за по-добра скалируемост (вкл. преодоляване на ефектите от “stop-the-world”) се прилага проследяване в групи от процеси:
  - процесите се разделят на групи, в които се извъшва групово проследяване – асинхронно на останалите групи
  - след като са изчистени всички групи, се извършва глобално проследяване, което се очаква да е по-бързо, тъй като вече са изчистени повечето Б-компоненти