**10. Управление на процесите в**

**разпределени системи**

**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, MМU и TLB регистри), управление на swap операции между основната и външната паме (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:

Рroc-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:

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

Други примитиви от тази група са еxit 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 Рroc-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

**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 С 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 междупотребителските процеси освен общи променливи, чийто тип зависи отизползвания език

- променливата или структурата, която е с общ достъп, се декларира съгласноезиковия 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 (иалоцирана в общ сегмент от паметта) със системната заявка Share():toMySharedStruct = Share(0, sizeof(mySharedRecord));

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

**Особености на процесите в**

**разпределените системи**

- ефективното планиране на разпределените приложения (предимнопо 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 сърверни машини както и между сърверите с различно предназначение (напр. обработващи,файлови, комуникационни, за разпределени обекти и др.)

- възможности за мигриране на процеси особено в хетерогеннасреда и необходимата динамична реконфигурация на клиенти исървери (процеси)

- прилагане на обработка с процеси-агенти – равнопоставенипроцеси за обслужване (вместо асиметричния модел клиент-сървер)

**Нишки**

- подпроцесите – традиционно «нишки» са средство за постигане на по-фина грануларност респ. по-оптимално планиране

- при нишките е недопустим свръхтовар като при процесите → по-слабаконкурентност 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 откодирането намногонишковото приложение (→ по-сложно програмиране)

- многонишково програмиране се прилага и при унипроцесорни приложения –

пример: електронна таблица с отделни нишки за потребителски интерфейс и заобработка на формулите

- многонишковата програма заунипроцесор е преносима и за паралелна обарботка - многонишковите програми са по-удобни за настройка – пример: текстов редактор

с отделни нишки за UI,грамат. проверка, форматиране, генерация насъдъжание 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, носвръхтовара е съпоставим с процесния

**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**

- обикновено се конструират с нишка-диспечер, която получава всяка нова заявка за обслужване и я асоциира с някоя от изчакващите изпълнителни нишки –

фиг. 10.18.

- 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 й към сървера.

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 миграрене на данни, които са адреси на файлове с URL идентификация (понеже идентификатора е униресален)

- свързване по стойност – напр. адресиране на стандартна библиотека в С 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; изпълненито 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

**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**

върховете на което са разположени имената на компонентите;

имената във върховете са на компоненти-директории; 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) e допълнително име на компонент: когато графът на имената допуска повече от един път до компонента – пример

в UNIX (фиг.10.23) 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 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, простраството имена се разделя и административно на

неприпокриващи се части – зони – всяка от които се обслужва от съответен сървер на имената

**Domain Name System DNS**

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 63В, а с абсолютния път – до 255В

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:

**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)

**Итеративно решаване на адресите**

при итеративното решаване на адресите пълното име (с път) –

напр. ftp://is.fmi.uni-sofia.bg/t3/rITa1.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. 10.29 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.unisofia.bg/t3/rITa1.pdf – се предава отново на сървера на имената в корена сърверът на имена не връща решения адрес (на следващ сървер) към клиента,

а вместо това предава остътъка 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. 10.30

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 – сърверния; обикновено тези две стъб-части обслужват разчистването, защото разполагат с информация за текущите обръщения могат да маскират тази системна функция от клиентския и сърверния процес граф на указателите с множество на корените, които не се премахват дори и когато няма указатели към тях – напр. потребители, системни услуги – фиг. 10.31. 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

скeлетон-стъба на обектния сървер – фиг. 10.32 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 асооциира 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 идемпотентноста 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 markand- 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)

**Мark-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 “stopthe- 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