1. Модели машинна архитектура и обработка. Класификация и метрика. Мултипроцесори: UMA, NUMA, COMA. Векторни и потокови машини и систолични матрици. Мултикомпютри.

Класове компютърни архитектури. Комп. арх. дефинира компонентите и организацията на една система. Фон Неймановата арх. се използва при възли и мрежи при некласическа организация (систолични, потокови, логически и редукционни модели и невронни мрежи). Ще разгледаме класификацията на Майкъл Флин за архитектури по управление на потока инструкции и покота данни (операнди) – SISD **(фиг.1.1.)**, SIMD, MISD, MIMD. SISD е класическа архитектура. Останалите се използват от машини за паралелна обработка. SIMD се използва за векторна обработка, фина грануларност. MISD – за конвейрна обработка (обработващи фази върху вектор) – систолични масиви, MIMD – обикновено с локална и глобална памет; за средна и едра грануларност. Класификацията на паралелните архитектури е технологично-ориентирана: мултипроцесори, мултикомпютри, потокови машини, матрични процесори, конвейерни векторни процесори и систолични матрици – частично съответствие с класовете на Флин. **HW/SW (хардуерен/софтуерен) паралелизъм**. Паралелизмът представлява максималният брой инструкции на 1 програма, които може да се изпълняват при обработката на тази програма. За паралелно изпълнение на програми е небходима едновременно апаратна и програмна поддръжка. Апаратния (хардуерния) паралелизъм се обуславя се от архитектурата и ресурсите, които са баланс между производителността и цената. Характеризират се с пикова производителност и средно натоварване. Той задава зависимостта по ресурси. Програмен паралелизъм се обуславя от зависимостта по данни и по управление. Реализира се като: 1) паралелизъм по управление - конвейризация, мултиплициране на функционални възли. Обслужва се паралелно, прозрачно за програмиста.2) паралелизъм по данни - типичен за SIMD и MIMD. **Метрика: ускорение и ефективност.** Ускорението (speed up) е S(n)=T(1)/T(n), а ефективността – E(n) = S(n)/n (нормирана стойност на ускорението); n е броят процеси, ако арх. е фон Нойманова, то n=1, ако имаме повече от 1 процесор: n>1; Редно е S(n) > 1. Най-добрият случай е – включвайки n процеса да намалим времето n пъти **(фиг. 1.2.)**: Линията HW е на ъгъл 45 градуса; SW се определя от контекста на проблема и е независима от n. При стойности над HW – аномалии, под нея – ограничения са наложени. Пр. ако имаме масив от n елемнта – макс. паралелизъм е n (осигуряваме асинхронна операция върху всяка негова клетка). Графиката винаги започва от т.(1,1). Нашето ускорение (кривата) се стреми към SW. Ако сменяме параметрите, то ще получим фамилия от криви. Обикновено имаме нужда от синхронизация в края или началото на програмата. **Делене на обработката: грануларност. (фиг. 1.3.)** Важни са свойствата linearity (линейност – стремеж ускорението да бъде плътно до линията HW – виж горе) и scalability (мащабируемост). Още по-важно е грануларността – размерът на използваните процеси и начина, по който те разпределят проблема. Нивата на грануларност са 5. Фината грануларност (coarse) е на ниво компилатор – при цикли, вектори – прилагане на еднотипна операция върху няколко елемента. Средната грануларност ена ниво подпрограми и процедури, редът, в който ще се изпълняват клоновете на програмата. Очакваме, че при фина гранулация ще нараства броят на процесите, но това увеличава и използваните ресурси. **SIMD(Single instruction, Multiple Data).** Използва се най-вече при машини за векторна обработка**.(фиг. 1.4.)**. Обобщеният модел включва контролно устройство и еднотипни обработващи модули с достъп към обща памет. Програмно-апаратна зависимост на паралелизма /ускорението – пример за изпълнение на програма на SIMD машина **(фиг.1.5А., фиг. 1.5Б.).**Процесорните елементи изпълняват операциите във формат битове или думи. локалната памет за данните може да бъде разпределена, обща или йерархична (със свързваща мрежа). Особености: 1)опростена архитектура спрямо MIMD поради общото контролно устройство (за дешифриране и зареждане на инструкциите) и съответно поддържане само на едно копие от кода за инструкции;2) скаларните операции (включително контролната логика) се изпълняват от контролното устройство – евентуално конкурентно на паралелната обработка на данни в обработващите устройства;3) имплицитна синхронизация между отделните обработващи устройства (при MIMD – експлицитна). Примери – фамилия Connection Machine на Thinking Machine Co. При SIMD се достига най-голям паралелизъм – в някои изчислителни центрове броят на изпълнителните елементи е над 10000. **MISD (Multiple Instruction – Single Data) (фиг.1.6.)** Това е архитектурния принцип на всички конвейри – вкл. на процесорния конвейер – обработката се разделя на последователни фази; обработката на следващата инструкция (при най-фина грануларност) или на следващия процес започва веднага щом предходния процес освободи първата фаза. Закъснението при отделните фази (stages) трябва да е равно, не трябва да има бавни. Прилагат се и функционални (или циклични) конвейри например с фазите: четене на инструкциите от обща памет, зареждане в обработващото устройство с евентуално буфериране,обработка, пренос на резултата към общата памет (буфериране), запис в общата памет. Същуствеват няколко нива на конвейеризация: инструкционно, субсистемно (обикн. при аритметична обработка – нелинейни конвейри с фази add, mul, div, sort…) и системно ниво (процеси, също и програмна организация) на конвейризация. **Систолични матрици (Systolic Arrays) -** представляват модификация на MISD на субсистемно ниво, специализирана архитектура за определени алгоритми – с многодимензионни конвейри т.е. фиксирана мрежа от обработващи устройства. Имат ограничено приложение – ЦОС (цифрова обработка на сигнали – DSP), обработка на образи и др. Имат опростени процесорни елементи и комутационна съобщителна мрежа с ограничен набор шаблони. управлението е по инструкции (control flow – не data flow) но програмирането е като при потоковите архитектури. Архитектурата включва обработващ масив (с комутатор) и управляващ модул, който настройва масива, предава данните и извлича резултатите (+ контролен възел – хост) **(фиг. 1.7.).** Производителността се понижава значително при интензивен вход/изход. Има и топологични шаблони: 1) систолични вектори – по същество конвейри; 2) двудименсионни масиви –обикновено регулярни с коеф. на съседство най-често 4 или 6 **(фиг. 1.8., фиг.1.9.)** Тенденцията е към елементи за фина грануларност – на инструкционно ниво – снабдени с няколко високоскоростни дуплексни серийни канали (броя на които определя валентността – коеф. на съседство). Пример: iWrap серия на Интел и университета Carnegie-Mellon – процесорната клетка се състои от: iWrap компонент с изчислителен и комуникационен агент и страницирана памет с директен интерфейс към компонента. Пример: умножение на матрици в двумерен систоличен масив с коеф. на съседство 6 **(фиг. 1.10.)MIMD (Multiple Instruction – Multiple Data) (фиг. 1.11.)** Това е архитектурния принцип на всички мултипроцесори и мултикомпютри. Процесорите са автономни и могат да изпълняват различни програми (вкл. локално копие на ОС!). Имат общ ресурс с разпределен конкурентен достъп – памет или комуникационна среда. Организация: 1)автономни (локална памет) - общо адресно пространство (общодостъпна памет);2) магистрални – комутационни. Характеризират се с универсални, отказоустойчиви, по-едра грануларност. Обикновено се изграждат с масови процесори (вместо специализирани процесорни елементи с ограничени функции). Наличието на автономна локална памет ги разделя на:1) системи с обща памет; синоними: мултипроцесори | [shared-memory |tightly-coupled] systems | Global-Memory MIMD, GM-MIMD | Uniform Memory Access System – UMA;2) системи с обмен на съобщения; синоними: мултикомпютри, [distributedmemory | loosely-coupled] systems | Local-Memory MIMD, LM-MIMD | Non-Uniform Memory Access System – NUMA (поради наличието на локална и отдалечена памет). Разполагат с глобално и локално адресно пространство; виртуалната памет поддържа глобално адресно пространство на страниците (не на ниво думи), което се управлява от разпределена ОС (РОС) за мултипроцесори и хомогенните мултикомпютри. При мултикомпютри общата виртуална памет се поддържа и с обмен на съобщения. Хетерогенните мутликомпютри използват мрежови ОС (МОС), при които нивото на достъп е разпределена файлова система (напр. базирана на DNS) с ползване на примитиви от типа rlogin, rcp... **Мултикомпютри (разпределени машини).** Използват NUMA(Non-Uniform Memory Access System). Характеризират се с разпределената обща памет (distributed shared memory DSM): програмната имплементация на обща памет в система с автономни възли (и адресни пространства). Има виртуално общо адресно пространство от страници (не думи) – 4/8 kB – (което позволява програмиране за мултикомпютъра като за виртуален уникомпютър). При отсъствие на страница от локалната памет възниква вътрешно прекъсване (memory trap) и зареждане на страницата в локалната от отдалечената памет. Възможно е репликиране на страници само за четене (read only). **(фиг. 1.12)** – 1,2,3 са компютри – процесите в компютрите са свързани помежду си с обща памет. Обаче, ако 1 иска да достъпи страница №10, тя трябва да е read only – така се имитира общо адресно пространство. Ако страницата е и за запис, се прилагат различни мерки за поддържане на свързаност. Принципът е приложим и при системи с обмен на съобщения – Message passing distributed systems. **Архитектура с обща памет(мултипроцесори) – UMA -** (uniformly shared memory access) - еднакъв достъп на процесорите -силносвързани системи. Характеризират се с: 1) обща шина - разширение от унипроцесинг към мултипроцесинг; недостатък – трябва да итерираме достъпа; 2) комутируема матрица (crossbar switch) **(фиг. 1.13)** – свързваме някоя обща памет с определен процесор – рядко разпространение;3) многоканални мрежи **(фиг. 1.14).** Паралелните интерфейси са бързи на много къси разстояния **(фиг. 1.15.)** – пистите, по които тече токът; ако увеличим големината на пистите, скоростта пада. Следователно трябва да променим формата им, с цел да се различат отделните изпращачи на сигнали. При кодиране на сигнала **(фиг. 1.16.)** – ако искаме да изпратим 0, забавяме честотата (това се използва за моделите, които се слагат на 32 или 64b магистрала. Видове синоними: симетричен (централизиран В/И) и асиметричен (специализиран процесор за В/И) мултипроцесинг - обикновено хомогенни системи. **NUMA и CUMA** - NUMA(non-uniformly shared memory access) – йерархия на общата памет - локални, глобални и/или клъстерни памети **(фиг. 1.17.)** и CUMA (cache only shared memory access) - паметта е лакална (cache) но йерархията и позволява част от нея (“директория”) да се адресира отдалечено **(фиг. 1.18.).** И двата модела се използват при мултикомпютрите. **Потокови архитектури (Data Flow)** При класическите фон Нойманови архитектури (вкл. модификациите по Флин) програмата е последователност от инструкции, която се изпълнява от контролно устройство – control flow. При потоковите архитектури операциите се изпълняват веднага при наличие на операндите (и наличие на операционен ресурс) – контрола се осъщесвява чрез планиране на операндите т.е. данните; концептуално всички инструкции с готови операнди могат да се изпълнят паралелно (на практика конкурентно). Програмите за потокови архитектури се представят с потокови графи (обикн. с текстов синтаксис) – възлите представят операции, а дъгите – информационните връзки на операндите; нивото на паралелизъм обикновено е инструкционно **(фиг. 1.19.)** – X = (A+B)*(C-D): 1)Add A, B; 2)Store T1; 3)Sub C, D; 4)Store T2; 5) Mul T1, T2; 6) Store X. (A,B,C,D – имена на променливи, които компилаторът транслира до относителни адреси; когато програмата се зареди – тези адреси са вече абсолютни. Процесорът работи с относителни адреси). **Статични потокови архитектури.** При тях програмният (потоковия) граф е фиксиран. За изпълнение на повече от една програма се използват различни варианти на зареждането на данните, които се генерират на етапа компилация. Този модел не поддържа процедури, рекурсия и обработка на масиви. Организация – **фиг. 1.20.** .Съществуват статични потоци с реконфигурация - логическите връзки между процесорните елементи се установяват на етапа зареждане на програмата: топологията на връзките се решава от компилатора и след зареждане на програма остава фиксирана при изпълнението; Особености: 1) физическите канали съществуват, но са комутират; 2) броя алоцирани (заредени) процесори обикновено е по-малък от инсталираните процесори поради ограничения в комутацията – логическата връзка между процесорите е дърво, не всички процесори в листата на което се използват;3) пример – MIT Data Flow Machine – клетките памет съответстват на информацията във възлите на потоковия граф – т.е. инструкционните блокове (tokens) – когато блока е комплектован с операнди, той се предава като операционен пакет към елемент за обработка; пакета с резултата се връща в клетъчната памет**(фиг. 1.21.)**. **Динамични потокови архитектури.** Базират се на логически канали между процесорите, които могат да се реконфигурират по време на изпълнение подобно на система с обмен на съобщения – с маркирани блокове (tagged tokens). Дъгите в потоковия граф могат да съдържат повече от един блок едновременно (но с различни марки!). Операциите се извършват когато възела получи блокове (с еднакви марки) на всичките си входящи дъги. Циклични итерации могат да бъдат изпълнявани паралелно: за целта всяка итерация се представя като отделен субграф като маркировката се разширява с номера на итерацията **(фиг. 1.22.)** (само при информационна независимост на итерациите!). Пример – Manchester Data Flow Machine MDM: цикличен конвейер, в който блоковете циркулират и се управляват от ключов модул. Компонентите са: 1) Блоков буфер (token queue) – за съхраняване на междинни резултати (ако се произвеждат по-бързо отколкото е последващата им обработка) – капацитет 32К блока и производитилност 2.5 МБлока/Сек; 2) Комплементираща памет (matching store) – за комплементиране на блоковете с еднакви марки – процеса е апаратен и поддържа до 1.25 МБлока; 3) Памет инструкции (instruction store) – n-торките (обикновено 2ки) операнди-блокове се пакетират с инструкции и адрес (етикет) на резултата и се предават за изпълнение. **Съпоставка на компютърните архитектури.**

Тип	Принцип на действие	Интерфейс	Приложимост	Сложност	Ефективност
SIMD	*спонтанен*	*директен*	*средна*	*висока*	*висока*
MIMD	*сложна абстракция*	*най-сложна организация*	*висока (универсални)*	*Висока*	*средна*
MISD	*спонтанен*	*директен*	*ниска*	*ниска*	*висока*
Систолични	*сложна абстракция*	*директен*	*ниска*	*Средна*	*висока*
Потокови	*сложна абстракция*	*сложна организация*	*висока*	*висока*	*висока*

Мрежи за връзка. Осъществяват комуникациите между процесорните възли при всички видове мултипроцесори и мултикомпютри – статични и динамични (базират се на [каскади от] комутируми блокове - ключове). Топологии на свързване: пълен граф, линия и пръстен, двудименсинна циклична и ациклична мрежа, хиперкуб (n-куб), двоично дърво, shuffle exchange. При мултипроцесорите комуникационния метод е чрез обща шина – централизирано се свързват с паметта. При мултикомпютрите – суперканал – имаме арбитраж на заявките – broadcasting (един предава към всички в своята група). Логическата топология е * - т.е. от всеки към всеки. Има отлагане на заявката, достъпът не е веднагически. Мрежите, в които няма broadcast, използват някаква топология – разпределяне: това е централизиран подход, който усложнява схемата (напр. P2P). Топологията дефинира релация на съседство – връзката между съседи е пряка, а между несъседи – непряка. За да се поддържат топологии, се използват каскадни комутатори – превключват серийните канали, свързващи двойки възли **(фиг. 1.23.)**. **Хар-ка на мрежите за връзка.** 1) разстаяние *dij* -диаметър на мрежата *D* = max{*dij* , за всяка двойка(*i*, *j*)} –изисква по-голям брой канали между възлите, респ. валентност; 2) валентност на възлите (degree) 3) сечение (bisection width) *S* = min{AllLinks(X, Y): ||X| - |Y|| ≤ 1}; 4) разширяемост.

Топология	Брой възли	Валентност
Линия и пръстен	d	2
Двоично дърво	2^d - 1	3
Shuffle exchange	2^d	3
Двудеменсионна мрежа	d^2	4
Хиперкуб	2^d	D
Пълен граф	N	N-1

2. Процесорни архитектури

Main frame – това е широк архитектурен клас от компютри, които прилагат някой от следните процесорни архитектури: 1) Скаларни процесори CISC (Complex Instruction Set Computer); RISC (Reduced Instruction Set Computer); 2) Суперскаларни CISC; RISC, само че RISC се използва по-често по технологични причини;3) Процесори VLIW – Very Long Instruction Word; 4) Векторни; 5) Суперконвейрни [super piperline];
Основни характеристики на всички архитектурни класове процесори: 1) Процесорни цикли CPI; 2) Тактова честота CR.Тези два параметъра на пръв поглед са независими, но между тях съществува корелация, която може да се представи в диаг. на технологичното пространство: (фиг. 2.6) На диаг.та: колкото сме под правата CPI=1, толкова по-добре. Там са тези процесори, които имат ниво на паралелизъм изпълняват повече от 1 инструкция за 1 цикъл. Ако сме над CPI=1, то трябва повече от 1 цикъл за 1 инструкция. Процесорните архитектури RISC интерпретират прости команди (около 5 процесорни фази), докато CISC интерпретират по-сложни команди (около 7-8 проц. фази).
Фази на инструкционен конвейер: Процесорната обработка на типична инструкция реализира MISD паралелизъм на инструкционно ниво и минава през фазите: 1) извличане-fetch (от Instruction cache - обикновено 1 инстр. за цикъл), 2) декодиране-decode (от Instruction cache -установява функцията за изпълнение и необходимите ресурси - регистри, магистрали, устройства), 3) изпълнение - execute (от Data cache - състои се от 7-8 устройства, някой от които са свързани с паметта, в зависимо от операцията се избира устройството), записване – write back (от Data cache – ако има резултат, то той се записва на мястото на генерирания адрес). (фиг. 2.7)
Времедиаг. на инструкционен конвейер: 1) Закъснението м/у 2 последователни инструкции е една фаза при скаларните процесори (фиг. 2.8) 2) Поради ресурсен конфликт м/у фазите на извличане и запис по-чести се прилага закъснение на 2 фази м/у инструкциите (фиг. 2.9) с легенда (фиг. 2.1) **Синхронни линейни конвейерни процесори:** 1)ЛКП е каскада от k процесорни фази (stages - Si), която изпълнява фиксирана функция върху данните, преминаващи през устройството от входа (S1) през последователните фази (Si→Si+1) към изхода му Sk. Те не са динамично (runtime) настройваеми т.е. са статични. Изпълняват операционни, аритметични и обменни инструкции. 2) Синхронните ЛКП са с интерфейс между фазите, който представлява синхронизиращи буферни ключове (latches) с общ такт. Ключовете са регистри които изолират входовете от изходите и предават данните синхронно във всички фази. Фазата с най-голямо закъснение определя общия такт и общата производителност (Ppeak): $P_{peak}=f=1/\tau$, $\tau = \tau_{max}+d_{lach}$, където τmax е времето на най-бавната фаза, а dlatch е закъснението. Проявява се и фазово отместване s (skew[ing]) на такта при предаване на тактовия сигнал между фазите. Затова се избира $\tau = \tau_{max}+d_{lach}+s$. (фиг. 2.2)
Асинхронни линейни конвейерни процесори: Те контролират потока данни с “Hand Shaking” протокол - Ready/Ack между Si→Si+1. Подходящи са за комуникационни канали при системи с обмен на съобщения. Производителността на отделните фази може да варира. (фиг. 2.3) **Нелинейни конвейерни процесори НЛКП:** 1)Те са динамични, настройваеми, допуска се разклонение - обратна връзка (feedback) и пре-предаване (feedforward) на данните за обработка. Изходът може да не е от последната фаза. (фиг. 2.4) 2) Карта на резервацията. Тук не е тривиална като при ЛКП. За различните функции може да варира по устройства и време (тактове).Тя се дава и съвместимостта на последователните функции по устройства т.е. зависимостта им по ресурси. (фиг. 2.5)
Анализ на закъснението при НЛКП: 1)Закъснението (latency) се представя от броя процесорни тактове *k* между две последователни инициирания на функции.2) Опита за повече от едно иницииране едновр. на едно устр. е колизия, която се избягва чрез планиране на последователността от инициирания. 3) Когато закъснението е такова, че предизвиква колизия, то е забранено закъснение. Трябва да се избере последователност от закъснения, така че да не предизвиква колизия. Пример за две забранени закъснения с карта на резервацията (фиг. 2.5). 4)**Цикъл на закъснението** е последователност от закъснения, която се повтаря неопределено дълго. Интервалите между две последователни инициирания на функции в цикъла на закъснението може да са еднакви, (константен цикъл), но може и да са различни, при което се изчислява средно закъснение. Чрез коеф. на запълване на цикъла се получава ефект. на конвейера.
Инструкционен конвейер: 1)ИК е специализиран за обработка на последователните инструкции в машинния код чрез припокриване(overlapping); 2)Типичната инструкция минава през фазите извличане, декодиране, издаване, изпълнение и записване; 3) Архитектурата на процесорния конвейер 4)Преподреждане на инструкциите за по-голям коефициент на запълване на цикъла
Обработка на преходите: 1)Конвейеризацията се лимитира от зависимостта по данни и от инструкциите за преход; 2)Производителността при програма с 20%/10% вероятност за условен преход между последователните инструкции, 50% вероятност за изпълнение на условието и 8-фазен конвейер е 41%/25% по-малка отколкото производителността при програма, в която поне едната вероятност е 0. Затова при конвейерни процесори е желателно алгоритъма да се кодира с минимум условни преходи. 3)Предвиждането на преходите се използва за да се отложи прехода докато се изпълнят определен брой инструкции, независими от условието на прехода. То може да бъде базирано на кода на програмата - статично или на историята на изпълнението – динамично.
Архитектура на набора инструкции: Разграничават се класовете RISC и CISC по следните параметри:1)формат на инструкцията и на данните; 2) режими на адресация; 3) регистърно адресиране (регистри с общо назначение); 4) управление на изпълнението на програмата.
CISC (Complex Instruction Set Computer): Това е Класическа архитектура (първите процесори са ограничен набор инструкции). Увеличения набор инструкции настъпва с микропрограмирането с промяната на SWcost/HWcost . Параметри:1) 120 – 350 инструкции с няколко формата на инструкциите и данните; 2) 32 – 64 регистъра с общо предназначение 3)4 – 16 режима на адресиране; 5)голяма част от изразите на HLL(high level language) са микрокодирани (т.е. имат съответствие в набора инструкции). Има скаларни CISC процесори – за операции върху скаларни данни и частична конвейеризация поради зависимостта по данни между последователните инструкции и ресурсения.
RISC (Reduced Instruction Set Computer): 25% от маш-ите инструкции кодират 90% от HLL програмата и се изпълняват 95% от процесорното време. Подходи за оптимизация: 1)трансформиране на микропоргмна памет в регистърен cache 2)FPU и други специализирани устройства на процесорния чип 3) суперскаларни процесори 4)броя на инструкциите е < 100 – с фиксиран формат (предимно регистър-регистър) 5) до 5 режима на адресиране, инструкциите са предимна от тип load/store 6)“регистърни фалове” – по 32+ вътрешни регистри за бързо превключване между процесите 7) едночипови, затова висока тактова честота CR и нисък CPI т.е. висок MIPS коефициент 8) скаларните RISC процесори са подобни на скаларните CISC но при еднаква тактова честота производителността може да е по-ниска поради по-малката плътност на кода 9)необходимост от ефективен компилатор за постигане на високо ниво конвейризация на ниво инструкция 10)суперскал. RISC архитек .

Показатели	CISC	RISC скаларен
Бр. инструкции	128-256-300	24-32
Формат на инструкции	16-64 бита, т.е. инструкцията е с плаваща дължина	32 бита, т.е. инструкцията е с фиксирана дължина
Формат на адреси	8-12 бита, различни начини на адресиране на операционната памет, къси/дълги	Регистър – регистър, 3-4 броя на регистърните формули
CPI брой процесорни тактове	8-20 процесорни такта, т.нар. инструкции с различна степен на сложност	3-6 процесорни такта, инструкциите са с фиксирана дължина – опростени
CM управляващ контролен модул	Базира се на микропрограмиране	С помощта на апаратна логика(АЛ) hardware control

Суперскаларни процесори (RISC и CISC) - Повече от 1 инструк. на такт, поради наличието на няколко инструкционни конвейера – съответно няколко резултата от всеки инструкционен цикъл. Разлика им с векторните процесори е че векторните реализират SIMD на инструкционно ниво, докато суперскаларните реализират MIMD. Паралелизма се реализира на инструкционно ниво – само между логически независими инструкции, като кратност на инструкцията е m = 2 до 5, докато при скаларните процесори m = 1. На (фиг. 2.10) е показан суперскаларен RISC процесор.
VLIW процесори: Комбинират концепцията за хоризонтално микрокодиране и суперскаларна архитектура: дълги инструкции (стотици битове), които задават по няколко операции над операндите.Те се различават от суперскаларните процесори по: 1)бързото и просто декодиране на инструкциите, понеже една VLIW инструкция замества няколко суперскаларни; 2) имат по-ниска плътност на кода, но по-висок паралелизъм на инструкционно; 3)непреносим обектен код, понеже нивото на паралелизма при различните процесори е различно; 4)инструкционния паралелизъм се задава на етапа компилация – т.е. статичен, няма динамична диспечеризация и синхронизация. На (фиг. 2.11). е показан VLIW процесор.
Векторни процесори: Те са специализирани копроцесори за векторни операции – операндите в отделната инструкция са масив[и]. Дългите вектори (надвишаващи дължината на регистърните файлове) се сегментират. Инструкциите са тип: 1)регистър-регистър – кратки (адресират се регистърни файлове); 2)памет-памет – дълги (защото съдържат адреси от основната памет), те могат да обработват по-големи масиви с различна дължина. Типични векторни операции са: 1) зареждане на вектор от паметта на компютъра: V1←Mn; 2)запис: V1→Mn; 3)ескалиране: S1•V1→V2; 4) векторна операция, при която и двете операнди са вектори и резултата е вектор: V1•V2→V3; 5) редукция от векторни операнди и резултат скалар: V1•V2→S1. 6) зареждане вектор-вектор: • V1 →V2. 7) редукция на единичен вектор: • V1 → S1. 8) аналогични инструкции от тип памет-памет – операндите са от вида *Mi*(1 : *n*).
Суперконвейерна архитектура: При степен *n* цикъла на суперконвейера е 1/*n* от базовия цикъл на фазите. (фиг. 2.16). Закъснението за една операция е равно на базовия цикъл, но ILP е *n*. $T(1, n) = k + (N-1)/n$; $S(1, n) = n*(N+k-1)/(n*k+N-1) \to n$ за $N\to\infty$, където k е фазов базов конвейер, а N е последователни независими инструкции ; Cray1: *n*=3. (фиг. 2.12)
Суперконвейрна суперскаларна архитектура: Степента е (*m*, *n*) като *m* е кратността на едновременно издаваните инструкции (т.е. на супрескаларност), а *n* е кратността на супреконвейера (1/n от кратността на базовия цикъл между групите последователни инструкции). Закъснението за една операция е равно на базовия цикъл, но ILP е *n*. $T(m, n) = k + (N-m)/mn$; $S(m, n) = m*n*(k+N-1)/(m*n*k+N-1) \to nm$ за $N\to\infty$, където k е фазов базов конвейер, а N е последователни независими инструкции ; DEC Alpha: *n*=6, *m*=2.
Intel Pentium: С въвеждането на Pentium арх-рата,Intel прилага предимствата на некласическа паралелна архитектура в производството на процесор, предназначен типично за масови компютри: 1)суперскаларен процесор с ниво на инструкционния паралелизъм m=2 (3 за P4) – едновременна обработка на 2 целочислени операнда по модела MIMD (когато последователните инструкции нямат зависимост по данни или управление!); 2)всеки инструкционен конвейер се състои от 5 фази: извличане, декодиране,адресна генерация (типичен CISC процесор с много режими на адресация на ОП), изпълнение и запис; 3)изпълнението на последователни инструкции от всеки конвейер е със закъснение 1 фаза (извличане): два самостоятелни кеша: Instruction cache и Data cache по 8 KB.4) за ефективно съчетаване на работата на конвейрите се използват двата инструкционни конвейра; 5) със същата цел Data cache е с двупортова организация – по един самостоятелен порт за всеки от инструкционните конвейери; 6)cache буферите са с асоциативна организация на достъпа, т.е. асоциативната памет има 32-байтов TLB (Translation Lookup Buffer) с последните адреси, така че търсенето на зарежданата страница става в 32 адреса; 7)планирането на активните страници в cache е по дисциплината LRU(Least Recenty User); 8)изискването за свързаност (кохерентност) между данните в cache и в ОП се постига чрез специален протокол – MESI – което позволява изграждането на мултипроцесорни; 9)интегрирано FPU(Float Pointing Unit) устройство с 8-фазов конвейер (извличане, декодиране,адресна генерация изпълнение, обработка мантиса, обработка експонента, обработка приближение и запис, който може да изпълнява и две FP инструкции едновременно (когато едната от тях е присвояване). (фиг. 2.13).**Intel Multicore:** Intel Core Microarchitecture е технология, която прилага интегриране на машинната архитектура на симетричния мултипроцесинг в микропорцесор. Суперконвейрни суперскаларни ядра имат 14 фази с по 4 инструкции. Има две/четири независими ядра – NUMA мултипроцесинг с локализиран L1-cache за всяко ядро и общ L2-cache. Достъпа до L2-магистралата има интегриран арбитраж.

3. Паралелно програмиране. Принципи на разделяне и балансиране на програмите. Синхронни и асинхронни паралелни приложения. Параметри, метрика, анализ. Системни средства за паралелно програмиране.

Последователни и паралелни програми

Програмата се състои от процеси, които могат да бъдат изпълнявани последователно или кон-курентно. 1) При изпълнение на програма в ***среда за последователното програмиране***: -про-грамата се състои от един процес; -резултатът от изпълнението й с еднакви данни винаги е един и същ; -изпълнението на всяка инструкция е последователно и независимо от изпълне-нието на др. инструкции. 2)При изпълнение на програмите в ***среди с мултипрограмитене***: -програмата се състои от един птоцес; -управлението се предава последователно м/у различни процеси; -м/у отделните процеси съществува зависимост по време на изпълнение, но резулта-та от изпълнението им се запазва. 3)При изпълненеи на програмите в ***среди за паралелно про-грамиране***: -програмата се състои от множество паралелни (конкуриращи се) процеси; - тя включва освен управляващ код и данни, също и инструкции за синхронизация и обмен м/у про-цесите, които съставляват нейния планиращ процес (scheduler); -резултатът от изпълнени-ето на паралелната програма може да зависи от работата на планиращия процес.

Паралелни процеси

Процесите, изпълняващи програмата за паралелна обработка, могат да бъдат алтернативно: 1)***Реплики***, изпълняващи еднакви подпрограми в/у различни данни – модел SPMD (Single Program Multiple Data).Разликата от SIMD е, че в този случай синхронизацията се извършва на ниво подпрограма, а не на ниво инструкция и затова SPMD обслужване се изпълнява на MIMD компютри; 2) ***Различни подпрограми*** – модел MPMD (Multiple Program MultipleData). При този подход отделните подпрограми-процеси се пораж-дат като дъщерни на един (главен) процес.

Граф на процесите ([precedence | dependency] graph)

1)Зависимостта по данни и управление се изследва (чрез графи) на различни нива – блок,израз, променлива; 2) Компилаторите обикновено изследват графа на зависимостие на ниво израз и променлива – пример за серията изрази: S1: A=B+C, S2: B=A+E, S3: A=A+B. S1, S2, S3 – statements – представя-ме си ги като изпълнявани от различни процеси, намиращи се в един и същ контекст; те ще се конкурират за достъп до едни и същи променливи: A, B. Изразите се изобразяват като възли в графа на зависимостите, а дъгите са зависимостите като началото на дъга е променлива (аргумент или стойност) на израз, а край – същата променлива от следващ израз – освен когато началото и края на дъгата са аргументи (от дясната страна) на изразите. (**фиг.3.1**)

Типове зависимости в графа на процесите :(фиг. 3.2)

1) ***Зависимост по данни (data flow)***: резултата от израз е аргумент на следващ израз (прена-реждането на изразите или паралелното им изпълне-ние променя резултата на следващия из-раз) – тази зависимост е непреодо-лима; 2) ***Антизависимост (anti-dependency)***: аргумента на израз е резултат от следващ израз (пренареждането на изразите или паралелното им изпълне-ние променя резултата на анализирания израз) – тази зависимост може да бъде преодоляна чрез репликиране на променливите; 3) ***Зависи-мост по изход (data output)*** – резултатите от два израза се записват в една и съща променлива (пренареждане или паралелно изпълнение про-меня стойността на тази променлива)– тази зависимост може да бъде преодоляна чрез репли-киране на променливите; 4) ***Зависимост по вход (data input)***: два израза имат общ аргумент – тази зависимост няма значение при съвременните програмни системи (поради средствата за конкурентен достъп); 5) ***Зависимост по управление (data control)***: условно изпълнение на из-раз, където условието е резултат от предходен израз (разновидност на зависимостта по дан-ни).N.B.: За по-висок паралелизъм на кода се отстраня-ват антизависимостите и зависимостите по изход.

Пример за отстраняване на зависимости

Изходен код	Код с намалена завис.
for i=1, n, 1 **x**=A[i]+B[i] Y[i]=2***x** **x**=C[i]*D[i] P=**x**+15 endfor	for i=1, n, 1 **x**=A [i]+B[i] Y[i]=2***x** **xx**=C[i]*D[i] P=**xx**+15 endfor

Модели обща памет

1) В паралелните системи достъпът до общата памет и ресурси за В/И е конкурентен и се ба-зира на схемите за PRAM (Parallel Random Access Machine) – автономни процесори с конкурен-тен достъп до обща памет (която включва и В/И канали); ***2)в модела PRAM*** се предлагат 4 схе-ми за отстраняване на конфликтен конкурентен достъп до общото адресно пространство: **-EREW (Exclusive Read, Exclusive Write)** – резервиране на конкурентния достъп да даден адрес за двата типа операции; -**CREW (Concurrent Read, Exclusive Write)** – няколко про-цесора могат да четата едновремнно даден адрес, но операциите за запис са монополни; -**ERCW (Exclusive Read, Concurrent Write)** – допускат се няколко едновременни операции на запис но монополно четене; -**CRCW (Concurrent Read, Concurrent Write)** – конкурентните опера-ции са без ограниче-ние; ***3)**EW*** схемите съответстват на изискванията за консистентност (съгласуваност и детер-министичност) на данните и се прилагат като универсални при повечето паралелни алгоритми; ***4)***Конкурентните операции за запис при *****CW схемите*** имат ограничено приложение при ня-кои класове паралелни алгоритмиза обработка на графи и числова обра-ботка, при които по-стигат по-високо бързодействие от схемите с резервиран запис

Модел с обмен на съобщения

При обмен на съобщения всяка двоика процеси е свързана с комуникацио-нен канал, редставен с точно една променлива – последователните съобщения са стойностите на тази променлива; дефинирано е състояние на канала – напр. четене на променливата-канал се допуска само ко-гато състоянието му не е празен (респ. при запис – да не е пълен); синхронният и синхронният канал са с еднакъв режим на достъп но асинхронният има капацитет = размера на буфера (>1).

Паралелни алгоритми

1) Паралелните алгоритми са междинното звено във веригата на паралелна-та обработка (меж-ду изчислителния проблем и паралелната система) – архитектура, система/среда, програма, ал-горитъм, изчислителен проблем; ***2)*** Паралелния алгоритъм е абстрактно (формално или нефор-мално) представяне на изчислителен проблем като набор от процеси за едновре-менно изпъл-нение; ***3)*** Основните характеристики на паралелния алгоритъм (които отсъстват при посл. алго-ритми) са: ***брой процеси и логическата им организация*** (***master-slave***: примерно клиент-сър-вър – двата са инициати-ните процеси = master; slave- дефинират изпълнител-ната част на зада-нието), ***разпределение на данните*** (***декомпозиция +*** възможности за ***разпределена алокация***: конвейеризация), *точки на синхронизация* (оптимизиране), *модел на междупроцесния обмен* (основно обща памет – обмен на съобщения);***4)*** Разл. конкретни реше-ния на горните характе-ристики пораждат цял клас от ПА, базирани на един последователен алгоритъм.

Фази на проектирането на паралелен алгоритъм

Проектир. на парал. алгоритъм минава през следните фази: ***1) Разделяне*** (***partitioning***) – декомпозиция на проблема: **по данни** (главно **SPMD**) или **по функции** (главно **MPMD**) – разде-лянето се извършва с оглед на специфика-та на проблема; целта е да се дефинират множество подзадания; гранулар-ността при тази фаза не отчита особеностите на архитектурата, която ще се използва за обработка – резултатът от фазата е дефиниция на отделните задания; ***2) Кому-никации (и зависимост)*** (***communication***) – формулира информационните или контролните за-висимости между отделните подзадания; комуникациите се представят като канали и съобще-ния, които се предават по тези канали; архитектурата за обработка се игнорира и на тази фаза, но специфицирането на каналите помага да се оцени алгоритъма по комуникационна слож-ност; ***3) Формиране*** (***agglomeration***) – след оценка на изчислителната и комуникационната сложност на формулираните подзадания и прилежащите им комуникации, те се групират в за-дания, при което се отчитат характеристиките на архитектурата на обработка – основно брой процесори/възли и комуникационен модел – и в резултат се постига оптимизиране по следните характеристики: грануларност и балансираност (с оцека на изчислителната сложност на отделните задания), евентуално репликиране на данни и подзадания, оптимизиране на ко-муникациите (с оцека на комуникационната сложност на отделните задания), евентуално за-пазване на линейност (скалируемаст), технологично оптимизиране (напр. намаляване на раз-ходите за кодиране на заданията); ***4)Разпределяне*** (***mapping***) – незадължителна фаза (отсъст-ва при проектиране на паралелен алгоритъм за системи с динамично планиране- обикн. мулти-процесор с разпределена ОС), която се със-тои в разпределяне на формираните задания (или евентуално групи от задания) по обработва-щите възли на системата със кодиране на съответното решение. N.B.: обикновено се използва специален език за спецификация на зареждането и евентуално за настрока на комуникацион-ните канали напр. в системи с комутируеми канали, така че от алгоритъма се изисква да специ-фицира и комуникацио-ния граф на системата за обработка(**фиг. 3.3**).

Метрика и анализ на производителността

1) Сложността на последователните алгоритми (брой операции) се оценява като функция само на размера на проблемната област и следова-телно може да се оцени абстрактно от архи-тектура-та; при паралелните алгоритми тя е функция на архитектурата и на средата за паралел-на обработка (особено при динамично планиране); ***2)***Основен фактор при паралелните алго-ритми е ***степента на паралелизъм P*** – макс. бр. опера-ции, които могат да се изпълнят пара-лелно при обработката на алгоритъма – това е архитектурно-независима величина; при размер на проблема **W** не повече от **P(W)** процесора могат де се ползват ефективно; съществено е съотношението между паралелните и последователните сегменти на паралелни алгоритми.

Закон на Amdahl (1967):

При наличие на две интензивности (R – rate = интензивност) на обработка на даден порблем – ***високо-паралелна Rh (rate-high)*** и ***ниско-паралелна Rl (rate-low)***, които са в съотношение f:(1-f) по брой на генерирани резултати (междин-ни и крайни) – общата интензивност на обработка е $R(f) = [f/Rh + 1-f)/Rl]$-1, следователно $f \rightarrow 1$ $R(f) \rightarrow Rh$ и при $f \rightarrow 0$ $R(f) \rightarrow Rl$. (N.B.: макар че е формулиран за темпове на обработка, закона е в сила и се прилага за агрегирана степен на паралелизма на заданието). (фиг.3.4)

Ускорение и ефективност

1) При ***оценка или измерване на ускорението*** (***Sp = T1/Tp***) се приема, че всички процесори в двата случая са с идентична производителност; поради наличие на комуникационни и синхро-низационни закъснения **1 < *Sp* < *p*; *2)*** ***Аномалии***: -***суперлинейно Sp > p*** може да се наблюдава при неоптимален последователен алгоритъм или особени характеристики на проблема, които изявяват нисък капацитет на използвания хардуер: напр. при голям размер на данните е въз-можно значително закъснение на последователната обработка на проблема поради бавни операции с външната памет, докато при паралелна обработка с разделянето на данните между възлите този проблем отпада ; - ***немонтонно Sp1 > Sp2*** за ***p2 > p1*** – често срещана аномалия.***3) Ефективността***, която е нормализирано ускорение (***Ep = Sp/p = T1/(pTp) < 100%***), характе-ризира частта от общото време за паралелна обработка,през която процесорните елементи се използват.

Пример за оценка на ускорението и ефективността

Хиперкуб от p процесора изчислява сумата на n числа. Времето за локална операция „+" на две числа и времето за предаване резултата на съседен процесор е 1. Паралелният алгоритъм: чи-слата се сумират локално за време n/p, след което локалните (първоначално p на брой) парци-ални суми се предават на съсед (1) и сумират (1) за 2lb$p$ (фиг.3.4). $Sp = n/[n/p + 2\mathrm{lb}p] = np/[n + 2p\mathrm{lb}p]$; $Ep = n/[n + 2p\mathrm{lb}p]$. Получените зависимости показват обичай-ният ефект при по-голямата част от ПА на намаляване на ефективността с нарастване на p (при фиксиран размер на проблема n) – ефект от закона на Amdahl

Фиг. 3.5 Описание от лекции
(1) $\frac{n}{p}$ (2) Преносът от долните към горните – единица време за операция. (3) Преносът от горните-предни към горните задни. (4) Преносът към едно от задните.

Цена и коефициент на използване

1) Цена (cost) при обработката на парал. алгоритъм с **p** процесора за **Tp** единици време (N.B. единица време е времето за изпълнение на една елементарна операция) е ***Cp = pTp ,*** т.е. **Cp** е макс. бр. операции, които биха могли да се извършат за времето на обработка на съответния парал. алгоритъм; ***2) Коеф. на използване (utilization)*** при обработката на парал. алгоритъм, състоящ се от **Op** действителния бр. операции с **p** процесора е ***Up = Op/Cp = Op/(pTp)***, т.е. **Up** е отношението на действителните към потенци-алните операции при обработка на съответния парал. алгоритъм.

Темп и излишък

1) Темпът на обработка (execution rate) е архитектурно-зависим параме-тър и се представя с няколко скали: ***MIPS*** (унипроцесори, мултипроцесори), ***MFLOPS*** (SIMD, числова обработка), ***MOPS*** (SIMD), ***LIPS [# logic inferences p.s.]*** (**AI** приложения, където AI – изкуствен интелект). Освен по архитектурен критерий, изборът на скала зависи и от типа парал. алгоритъм, които се обработват; ***2) Излишък (redundancy)*** при обработката на парал. алгоритъм, състоящ се от ***Op*** на брой операции при обработка с **p** процесора е **Rp = Op/**$\mathbf{O_1}$ > 1 (където $O1$ е броя операции при обработка на уникомпютър), т.е. **Rp** е критерий за свръхтовара, който се поражда от паралеланата обработка на алгоритъма; **p** и **n** (размера на проблема) са аргументи на **Rp,** но в зависи-мост от класа парал. алгоритъм участват с различна тежест. Освен ***redundan-cy*** има и (System) ***overload.***

Алгоритмична сложност

Коректността на даден парал. алгоритъм е архитектурно-независима, но неговата ефективност зависи от изпълнителната платформа, поради което е целесъобразно сложността му да се оценява и като функция на разпределя-нето (mapping). По принцип **алгоритмичната сложност *O*** оценява времевите и пространствени характеристики на обработка – **времевата сложност *T*** се задава в брой елементарни операции и комуникации (от който се получава времето за обработка в дадена архитектура), а **пространствената сложност *M*** в брой алоцирани регистри и клетки памет (т.е. $O = O(T, M)$); Оценкта се дава обикновено като долна и горна граница на тези величини или с приближение – асимптотична сложност.

Паралелно програмиране в разпределени системи

Прилага моделите: 1) Разпределена обща памет (**DSM =Distributed Shared Memory**): ключалки семафори, монитори, бариери; 2) Обмен на съобщения (**Message Passing Systems**): - *приложно-ориентиран междинен слой* (**MPI** и **PVM** – процедурен модел; **RMI и Corba** – обектен модел); -йерархични (**master-slave**, **client-service** - Jini) и нейерархични модели (**P2P - Jxta**).

Конвенционален псевдокод за паралелни алгоритми

Псевдокодът е приложим за определени класове архитекту-ри – обикновено се взима като предпоставка най-разпространения PRAM модел за паралелен достъп до обща памет – CREW декларация на процедури и функции е разширена със запис на модела за паралелна обработка и броя алоцирани процесори:	**Procedure: <name> ({list of parameters})** Model: <model name> with p = f(n) processors **Input: <input variables>** **Output: <output variables>** **Declare: <[global and] local variables>** **Function: <name> ({list of parameters})** Model: <model name> with p = f(n) processors **Input: <input variables>** **Output: <output variables>**

Блок FORALL

Този блок се прилага за имитация на паралелно изпълнение на вложения в негосегмент (набор изрази) – асинхронно (в MIMD) или синхронно (в SIMD) синтаксис:

FORALL identifier: RangeType IN {PARALLEL | SYNC} Statement_1 … Statement_K END	**identifier** е упр. пром., деф. в граници-те на блока; по 1 процес се създава за вс. нейна ст-ст (м-вото ст-сти трябва да е крайно); в созд. процеси identifier има разл. ст-сти; **RangeType** е типът на упр. пром., чиято мощност освен това задава и бр. парал. процеси; **PARALLEL** или **SYNC** задава типа парал. обработка – съотв. асинхронен илисинхронен

Изпълнението на блока завършва след изпълнение на всеки от процесите. Асинхронната обработка означава, че част от процесите могат да се плани-рат след изпълнението на другите (когато броят им е по-голям от броя процесори).

Пример за блок FORALL

8 процеса за асинхронна паралелна бработка на функция с аргумент – номера на процеса	версия
FORALL x:[1..8] IN PARALLEL y = some_function(x); **END**	FORALL x ∈ X IN PARALLEL do y = some_function(x);

Израз do IN PARALLEL

Този израз се прилага като директива в различни блокове.

Пр.: при парал. векторна бработка интаксис:	Пр.: за вс. елем. на масивите се формира отделен процес

for <израз в/у инд. на масив> do IN PARALLEL
Statement_1
Statement_2
…
Statement_K
end IN PARALLEL
	for i = 1 to n do IN PARALLEL
	read(A[i], B[i])
	if (A[i] > B[i])
	then write(A[i])
	else write(B[i])
	endif
	end IN PARALLEL

Синхронизационни конвенции, семафори

1) Синхронизационните схеми биват: контрол на достъп (семафори и монитори) и контрол за последователност (бариери). ***2)*** Променлива от тип ***семафор*** се асоциира с всеки адрес за общ достъп и върху нея се извършват операциите: *установяване на състоянието* (активно или пасивно) (***wait***), *блокиране на процес* (***wait***), *възстановяване от блокиране* (**signal**). ***3) Wait(S)*** е заявка за достъп до критичната зона, която се потвърждава ако S>0 (и S се декрементира); в противен случай процесе блокира и изчаква. ***4)*** Signal***(S)*** освобождава критичната зона, инкрементира **S** и възстановява чакащ процес.

Синхронизиращ псевдокод със семафор	P1: wait(S1) {critical section 1} signal{S1} P1: wait(S1) {critical section 2} signal{S1}

Синхронизация с монитори 1) Мониторите са разширение на семафорите, което се състои както от данните за контрол на достъпа – ***condition variable***, така и от процедурите – ***signal*** и ***wait*** 2) При дефиниране на ***condition variable*** се създава и опашка на идентификаторите на чакащи процеси, които се възстановяват и получават достъп до критич-ната зона с операцията ***signal***	Monitor Resource_alloc Var Resource_in_use: Boolean; Resource_is_free: Condition; Procedure Get_resource begin is (Resource_is_free) then wait(Resource_is_free) Resource_in_use = true end ... Procedure Release_resource begin Resource_in_use = false signal(Resource_is_free) end end Monitor

Синхронизация с бариери 1) С бариерите се осъществява контрол за последователност – напр. за запазване на зависи-мостта по данни 2) Бариерата също се състои от буфер за готови изчакващи процеси и боряч	***Псевдокод без синхронизация*** For I = 1 to N do IN PARALLEL { S1: A[I] = func_a(A[I]) S2: B[I] = func_a(B[I]) S3: C[I] = func_c(A[I], B[I]) } ***Псевдокод с бариерна синхронизация*** For I = 1 to N do IN PARALLEL { S1: A[I] = func_a(A[I]) S2: B[I] = func_a(B[I]) **BARRIER(2)** S3: C[I] = func_c(A[I], B[I]) }

Задачи на балансирането на изчислителния товар (Load Balancing – LB, Resource Management, Resource/Job Scheduling)

1) Минимизиране времето за решаване на даден пробл. при парал.обраб. чрез изравняване на локалното натоварване на обраб. възли. ***2) Целта*** може да бъде не пълно изравняване а недопускане на възел в престой, докато трае парал. обраб. ***3) В грид*** – пропорцинално натоварване на ресурси с различна собственост и администрация. ***4)Източници на дисба-ланс:*** нерегулярност на проблема при паралелизъм по данни; недетерми-нистични алгоритми за обработка (напр. при неизвестен бр. итерации за достигане до решението – търсене в графи и др.); невъзможно или некомпе-тентно декомпозиране – при паралелизъм по данни или по упр-е.

Статично балансиране

1) Разпределянето на заданията по възли и алоцирането на ресурси се извършва (и е известно) преди да стартира паралелната обработка – ***планиране, комплементиране*** (**mapping**, **matchmaking**, **scheduling**). ***2)Подходи*** за статично балансиране: **RR** – ***циклично алоциране*** на заданията по обработващи процеси; ***стохастично разпределяне; рекурсивно разделяне*** – при алгоритмите за графи – бисекция (разделяне на проблема на подпроблеми с очаквана еднаква сложност на обработка и с генериране на минимален синхронизационен и комуникационен свръхтовар); ***генетич-ни и Монте Карло алгоритми*** – свързани са с генериране на възможни варианти на декомпозицията и оценяването им, така че да се избере оптималния.

Недастатъци на статичното балансиране

1) Проблемна предварителна оценка на сложността на подпроблемите, получени при декомпозицията. ***2)*** Не може да отчете текущото състояние на ресурсите по време на обработката – фоновото натоварване на ресурсите (процесорницикли, памет, комуникационни канали), както и реалните синхронизационни и комуникационни закъснения – ограничено приложе-ние за синхронни алгоритми. ***3)*** При недетерминистични алгоритми за обработка, напр. при неизвестен брой итерации за достигане до решението – търсене в графи и др. – статично решение на задачата за товарен балнс е невъзможно освен чрез прилагане на по-фина грануларност и откриване на край (distributed termination detection).

Динамично балансиране

1) Разпределянето на заданията ***по възли и алоцирането на ресурси*** се извършва по време на паралелната обработка и е известно едва след приключването й. ***2) Централизиран подход*** – ***master-slave*** обработка; декомпозицията, разпределянето на заданията и ресурсите, откриването на край или алтернативно интегрирането на резултата са функции на един ***master*** процес. ***3) Разпределен подход*** – декомпозиция на управляващия процес в йерархия от упр. процеси или асоцииране на упр. функции с всеки от обработващите процеси. (фиг. 3.6)

Описание на фиг. 3.6, където I- Information, L- Location, T- Transfer
(I) Услугата за наблюдение се нарича monitoring – следи натоварването в отделните възли. (L) Match-ва даден ресурс. (T) Извършва се трансфер на данните.

Централизирано динамично балансиране

1) Главния процес функционира като ***пул от задания*** (***work pool***) и получава заявки за ново задание от готовите изпълнителни процеси; изпълнителните процеси са обикновено реплики (модел SPMD). ***2) Пулът от задания*** се прилага при матричните изчисления, при алгоритмите "разделяй и владей". ***3) Нерегулярните и динамичните товари*** също са подходящи за ***work pool*** обработка – в посл. случай генерираните от обработката нови задания се присъединяват в опашката на пула заедно с тек. резултат от изпълнителния процес (фиг. 3.7) ***4)Осн. предимство*** на централизираното динам. баланси-ране е лесното установяване на изпълнение на условието за край – при празен пул и прекратена работа на изпъл. процеси; при някои алгоритми за търсене условието за край се открива от някой от изпълнителните процеси и се предава към главния процес заедно с резултата. ***5) Недостатък*** е възможността за възникване на тясно място и ниската линейност. (фиг. 3.9 – BDP = Birth-Death Process)

Разпределено динамична балансиране

1) Пряк подход е ***разпределяне на функциите*** на упр. процес по поддържа-не на динамичния пул от задания на йерархичен слой на упр. процеси – фиг. 3.8. ***2) Оптимизацията*** в горния случай е предимно в избора на брой упр. процеси от втори ниво или евентуално избор на броя упр. нива. ***3)*** При някои алгоритми се практикува развито ***йерархично дърво*** – обикновено двоично, тъй като разделянето на [под-]проблема на две очаквано равно части е по-лесно за алгоримиране и за прилагане на рекурсия.

P2P динамично балансиране

1) То е ***форма*** на по-пълно прилагане ***на разпределеното динам. баланси-ране***. Премахва се разделението на упр. и изпълнителни процеси като всеки процес извършва и двете функции. **2)** Формално и опростено цялото задание може да бъде предадено за ***изпълнение в един процес/възел***, след което се извършва неговата декомпозиция и послеващ балансиращ трансфер на генерираните подзадания межу възлите. ***3)*** В този случай ***декомпозицията*** е желателно да бъде ***или тривиална*** (примерно при матрични изчисления), ***или*** пък да бъде ***опростена*** (примерно бисекция на проблема без първоначален анализ колко са потенциалните обработващи процеси, какво е тяхното текущо натоварване и каква е оптималната грануларност).

Параметри на P2P динамичното балансиране

1) Подобни ***балансиращи схеми*** се наричат ***дифузионни***, тъй като реализи-рат балансирането чрез трансфер на подзадания към "съседни" възли; релацията за съседство в случая може да изхожда от конкр. топология на изпълнителната платформа, но може да бъде и подчинена на разл. стохас-тични принципи – напр. на случ. избор от опр. бр. (оптимизационен параметър!) "съседи". ***2)*** В горния случай като средство за повишаване на линейността на алгоритъма се избягват схеми, когато всички възли са "съседни"; вместо това се ***формират виртуални топологични структури*** – линия, пръстен, хиперкуб и др. (обикн. нейерархични) топологии; когато валентността на процесите е по-голяма от 1, може да се прилага циклично или случайно тъсене на "съсед" за балансиращ трансфер. ***3)*** Др. важен парам. на P2P балансиране е ***инициативата*** (или момента за активиране на лок. балансираща процедура): инициатива на донора, инициатива на приемника.

Системи за динамично балансиране

1) Информационна, локационна и трансферна стратегия – функции, граф, разпределение; (клъстерно, мултиклъстерно и c2c планиране). ***2) Синхронно балансиране*** – co-scheduling: **Koala.** ***3) Асинхронно балансиране*** – htc (High troughput computing), volunteer computing: **Condor/Condor-G , Boinc;** балансират се нископриоритетните процеси на опортюнистичните потребители във фонов режим (background priority).

Еталонни паралелни алгоритми

1) Асинхронни алгоритми – Mandelbrot set. При **Mandelbrot set:** имаме една функция в/у комплексната равнина. Тя се прилага в/у комплексен елемент и резултатът също е компл. число. Така итерираме и получаваме числа близки илине до (0,0). Изследваме бр. необходими итерации за достигане на (0,0). Ако същ. такъв брой, казваме, че това е число от ***Mandelbrot set***. Това наподобява фрактални изчисления. Ако пуснем теста на ***Mandelbrot*** в/у една технология: имаме матрица, чиито елемент са цветове и оцветяваме в по-светло, ако проблемът се е решил по-бързо. **Фиг. 3.10.A** →тъмното петно значи технологията не е много подходяща. Ако имаме 4 ядра разделяме матрицата на 4 равни части. **Фиг.3.10.B** → в това ядро (долу в дясно на схемата) тъмното надделява, значи на него се е паднала най-тежката задача. ***2) Локално-синхронни*** алгоритми – Water simulation, odd-even sort. При Water simulation: синхронно е, защото стъпка по стъпка се развива симула-цията. Можем пак да направим декомпозиция на тероида: гранулността тук не е много ефективна, т.к. всеки процес ще има по 4 съседни и трябва да има комуникация м/у тях – деф. на релация на съседство. Например модела хищник-жертва. **Фиг.3.11** → при стартиране на симулация обектите преми-нават в съседно квадратче, ако то е свободно – това важи да жертвите, а хищниците отиват там, където има жертви. ***3) Глобално-синхронни*** алгорит-ми – n-body simulation, Ray tracing. При ***n-body simulation*** имаме n точки с планарни координати. Точките нямат форма и размери, но имат свойства – напр. маса и ускорение (пулс). **Фиг. 3.12.A** →Каква ще бъде траекторията на всяко едно от тези тела? Функция на взаимодействие м/у двойките тела. Всяко тяло има свой пулс и трябва да знаем масата и текущото положение. Вземаме гравитационната сила и така може, да изчислим новото място. Това правим за всеки. Но някои могат да се сблъскат, т.е. пак е нужна комуника-ция. Може пак да разпределим м/у p процесора, които ще си общуват чрез shared memory или message passing. **Фиг. 3.12.B** → динамичен LB – рекур-сивно разделя на подобласти.

4. Модели на софтуерната архитектура. Спецификация с UML и ADL.

1. Модели софтуерна архитектура

Софтуерната архитектура представя (моделира) програмния проект (процес на обслужване) като съставен, т.е. разпределен процес от софтуерни компоненти. Моделирането на разпределена софтуерна архитектура е първата и най-важна фаза на проектиране, настройка, тестване, разгръщане и документация на разпределени среди за обслужване. Моделът на дадена софтуерна архитектура описва декомпозицията на процеса на компоненти, функционалната им композиция, прилагания архитектурен стил, качествените (нефункционалните) атрибути на услугата – QoS. За представяне се използват графи и техните разширения. Описанието е чрез диаграми или текстови еквиваленти. Целта на описанието е за визуализация, спецификация, конструриране и документиране => обикновено моделът включва много повече от една диаг.. Описанието (моделирането) стартира от по-упростени концепции на бизнес-модела или потребителския сценарий. Напр. едномерен модел с блокова диаг. (ненасочен граф) – фиг. 4.1(разглеждаме диаг. за бизнес приложение за електронна търговия, която представлява неориентиран граф). За по-пълно функционално и нефункционално описание на проекта се прилагат многомерни модели, напр. „4+1” модели включващи логически изглед, изглед процеси, изглед проектиране, физически изглед, потребителски интерфейс изгледи.

2. Спецификации с UML

UML е средство за декомпозиране на проекта(софтуерната архитектура) в обектен модел. Този модел се състои от множество диаграми, представящи различни аспекти на проекта. UML-модели на софтуерна архитектура се използват за ОО-спецификация, анализ, проектиране и документиране на софтуерни проекти.
Спецификациите са в **2 групи диаграми:**
1**)структурни диаграми** – **статично** описание(изреждане) на елементите в системата (йерархична библиотека класове и статични връзки между класове като наследяване („is a”), асоциация (“uses a”), агрегация(“has a”), обмен(method invocation)).
2) **функционални (behavioral) диаграми** – **динамично** описание на функциите(„поведението”) на инстанциите на класовете (т.е. обектите) с диаграми на интеракцията, колаборацията, акцията и конкурентноспособността между обектите.
UML диаграмите могат да се транслират до HLL с общо приложение

3. Структурни и функционални диаграми

Структурни UML диаграми:

Class – изброяване и статични връзки между класовете (независещи от взаимодействието им по време на изпълнение). Те са най-разпространеното описание при всеки модел. При тях се прави статично изброяване на съставните блокове на модела като **класове**. Задава се **„речника”** на модела в съответствие с проблемната област. Класовете се описват с техните атрибути, които са име, интерфейс, методи, свойства. Достъпността (видимостта) на атрибутите се описва като public, private, protected, default. Описват се и отношенията между класовете – наследяване, асоциация, агрегация (чрез дъги), а също и мощността на тези отношение: 1:1, 1:много и т.н. (чрез маркировки в края на дъгите). Пример: фиг. 4.2 – система за потребителски заявки. Всяко блокче съдържа име на класа, атрибути и методи.Изполсват се още ромб към корена – агрегация, стрелка към базовия клас – наследяване, нейерархична дъга – асоциация, които показват връзките между класовете. Слага се и маркировка на мощностите в двата края на дъгите.
Object – извлечение от клас диаг.та. Описва обектите като инстанции на класовете, т.е. примерно подмножество обекти за дадена клас-диаг., и тяхното взаимодействие в определени специфични(конкретни) моменти от изпълнението на системата.Пример:фиг. 4.3
Composite – Диаг. на съставните структури – описание на структурата на даден компонент като съставящи го класове и компонентните интерфейси. Задава „речника” – логическите операции, които съществуват. Описва връзката между обектите (runtime), с което разширява „речника” на модела. Обектите и връзките се анотират с етикети – съответно на ролята(бизнес или функционална логика) и отношението им („колаборацията”). Пример:фиг.4.4
Component – описание на системата като структура от компоненти, интерфейсите между тях, и общите системни интерфейси. Компонентите са изпълними software – модули за многократно използване при проектирани, които се представят със своя интерфейс. В UML те са със скрита структура (черна кутия)[но при различните технологии се прилагат и компоненти тип „сива” или „стъклена кутия”], напр. jar в компонентната библиотека JavaBean или dll в .NET. Компонентната диаг. представя съответствието между изискваните(полукръгче) и имплементираните(кръгче) интерфейси – фиг. 4.5. Компонентите в даден проект може да са готови – COTS – и специфични.
Package – Йерархична пакетна структура на организация на класовете в директории(т.е. групирани файлове) – пакети от класове и пакети от пакети – фиг.4.6
Deployment – Диаг. на разгръщането – описание на изпълнителната инфраструктура: сървери, изпълняващи компонентите, системно осигуряване и мидълуер, интерфейси и протоколи, вътрешна и външна мрежова свързаност – фиг.4.7

Функционални UML диаграми:

Use case – Диаг. на случайна употреба – описват потребителските сценарии на заявки към системата и техните реакции като граф от актьори, случаи на употреба(потребителски функции) и връзките между тях– за описание на **функционалните и нефункционалните изисквания** към системата. Акторите са крайни потребители или други системи, приложения и устройства. Случаите (uses cases) са комплексни функционални модули от разпределеното приложение/проект, който описва отделни стъпки от цялостната бизнес-логика. Описанието на случаите се допълва в други диаграми с пред- или след- условията на изпълнението им като последователности от стъпките на общото приложение при конкретно негово изпълнение. Връзките между сценариите (фиг. 4.8) се маркират с <<include>> от случай, който използва друг случай за изпълнение на дадена функция (насочена стрелка), <<extend>> от случай, който извиква друг такъв за изпълнение на функция по изключение (т.е. като опция, която се изпълнява само по изключение). Диаграмите на случайна употреба са основа на описанието и [началните] им версии се използват за основа на структурните и sequence диаграмите.
Activity – Диаг. на дейностите – описание на контролния и контекстния обмен между класовете като мрежа от акции, които системата изпълнява, за да осъществи реакциите по потребителския сценарий – **оркестрация на акциите.** Този вид диаграми описват проекта като **потоков** (workflow) бизнез процес, състоящ се от дейности – activities. Дейностите капсулират логиката на взимането на решение, конкурентното изпълнение на функции, обработката на изключения и прекратяването на процеса (termination). Потоковата activity диаг. (фиг.4.9) се състои от една начална точка и поне една крайна точка (плътен кръг и ограден кръг), точките на решаване (означават се с ромбче), други дейности (заоблен правоъгълник), конкурентно разпределяне и събиране на потоците (дебела черта) (N.B. – събирането на два и повече потока се счита за синхронизатор (следващите го дейности не могат да се стрaртират без завършване на всички предхождащи го)) , събития (events-опция) – представят обмена на съобщенията (signals) между конкурентните акции (насочени многоъгълници с етикети).
State Machine – Диаг. на машина на състоянията – описание на жизнения цикъл на **обектите като машина на състоянията** – диаграми на състоянията и преходите(активни вътрешно обусловени и реактивни външно обусловени преходи). Състоянията се описват с блок, съдържащ име, списък променливи и activity. Логиката на състоянията е реактивна, т.е. се базира на външни събития (events). Диаг.та на фиг.4.10 се състои от една начална точка и поне една крайна точка (плътен кръг и ограден кръг), насочени маркирани дъги на преходите, състоянията, които може да са комплексни състояния, съставени от допълващи се State Machine диаграми.
Interaction Overview – Диаг. за преглед на взаимодействието – описва **потока команди** между обектите (control flow) и е комбинация от Action и Sequence диаграмите. Този вид диаграми се състоят от кадри (frames), които представляват други диаграми на проекта, маркирани с указател (reference) или със самите диаграми, маркирани с тип – напр. sd, cd, ad. На фиг.4.11 дъгите отразяват контролния поток на взаимодействието.
Sequence – Диаг. на последователност – **нареден (т.е. времеви) списък от съобщенията** между обектите. Те отразяват относителната последователност от контролни съобщения между обектите – фиг 4.12
Communication – аналогично на Sequence диаг.та, но структурирана като **комуникационни канали**, които съдържат определен брой последователности
Time Sequence – времево описание на преходите между вътрешните състояния на обектите и на различимите външни събития(от потребителския сценарий) като последователност от съобщения. Прилага се за RT приложения или системи – RTOS, ES

4. Модели на изгледи

„4+1” моделиране – представя разпределена софтуерна архитектура с 4 основни изгледа и един допълнителен – логически, развоен, процесен и физически + сценарий на функциониране, който често се придружава и от изглед на потребителския интерфейс – фиг.4.13.
Сценарния изглед и асоциираният с него интерфейсен изглед описват потребителските функции на приложението, както и основните нефункционални изисквания. Той произтича от потребителското задание, а в UML се специфицира с диаг. на потребителските случаи(use case диаграми).
Логическия изглед описва декомпозицията на разпределеното приложение с оглед на реализираните функции. Този изглед представя основните блокове или компоненти. В UML се специфицира клас-диаг. (статична), допълнена с една или повече динамични диаграми – най-често последователностни.
Развойният изглед и асодиираният с него интерфейсен изглед описват потребителските функции на приловението както и основните нефункционални изисквания. Този изглед произтича от потребителското задание. В UML се специфицира с диаг. на потребителските случаи (use case).
Процесният изглед описва декомпозицията на разпределеното приложение с оглед на реализираните функции. Този изглед представя основните блокове и компоненти. В UML се прецифицира с клас диаг. (статична), допълнена с една или повече динамични диаграми – най-често последователностни или на дейностите(фиг.4.14).
Физическият изглед описва цялата разпределена софтуерна архитектура на платформата + приложението – инсталация, конфигурация, разгръщане, Компонентите са на ниво процесори или поне процеси. Връзките между тях са на ниво комуникационни канали. Този изглед представя бабасябети (или картирането – mapping) на компонентите от развойния изглед върху инфраструктурните възли (фиг.4.15)

5. Спецификации с ADL(Architectural Description Language – графична спецификация на модели за разпределена софтуерна архитектура). Съществува свободно разпространявана среда за спецификации на ADL – модели AcmeStudio с автоматична генерация на Java и C++

5. Обектни, потокови и контекстни модели на софтуерната архитектура

ОО принципи: капсулиране – осигурява видимост на функциите и прозрачност за имплементацията . (Например скрит вътрешен контекст и процедури. Частните променливи в класовете са неустойчиви, а публичен интерфейс е устойчив.)
наследственост – осигурява адаптивност на кода чрез наследяване и допълване на спецификациите – т.е. от общо (родителски клас) към частно (наследен клас, дериват)
полиморфизъм – осигурява адаптивна функционалност чрез развитие на наследяването. Има два вида полиморфизъм – вертикален (отмяна и предефиниране на атрибути в дериватите) и хоризонтален (презареждане на нов контекст за същия клас).
АДТ: Математически модел за определен клас от структури от данни , които имат подобно поведение. АДТ се определя косвено, само от операции, които могат да се извършват върху него и от математически ограничения на ефектите (а вероятно и разходите) на тези дейности. Класовете са имплементации на АТД с публичен интерфейс от атрибути и операции, а обектите са имплементации на класове, които се явяват техни «типове».
– UML-спецификация на клас с +/- модификатори на достъпността на атрибутите и операциите – (*фиг. 5.1*).
Видове отношения между класовете: статични (1.конструкция на комплексни класове от класове (композиция и наследяване); 2.статична консистентност (т.е. логичност) на зависимите класове – като при базите данни (агрегация и асоциация)) и динамични (примерно, обмен на съобщения).
Композицията е дефиниране на клас като съставен от други класове. Компонентите са активни, докато е активен и съставния клас и не се включват в други класове. (пресилено ограничение за garbage collection – чрез конструктурите и деструктурите на класовете); в UML – плътен ромб към главния клас с етикети на мощността - (фиг.*5.2)*
Агрегацията е аналогично отношение на класовете, но без изброените ограничения – *(фиг. 5.3)*
Асоциацията е обобщена композиция – *(фиг. 5.4)* ; характеризира се с:
1) име (етикет), което отразява свързващата функционална логика – напр. «Customer places an/some Orders», 2) мощностите на асоцииране, 3) 2 асоциативни типа на връзката между двата класа (задават тип композиция към инициираща клас)
- навигационната посока към инициирания клас – т.е. указателите на асоциираните класове са налични като атрибути в инициираща клас (плътна линия)
- зависимост посока към зависимия клас – зависимия клас извиква операция на асоциирания клас или променя негов атрибут (пунктир)
4) иницииращият клас може да асоциира повече от един класове
Наследяване и полиморфизъм: наследяването отразява взаимстване на повтарящите сеатрибути – деривата наследява всички публични атрибути (без частните).
Полимофизмът е механизъм за диверсификация на дериватите при изпълнение *(фиг. 5.5)*. В UML наследявянето се означава с триъгълна стрелка към основния клас. В примера двата деривата се различават по методите на идентификация. Клиентът зарежда cookie в браузъра си, регистрираният потребител изпраща парола и ползва отстъпка (и двете функционалности отсъстват в базовия клас)
Наследяване и композиция: и двете черти поддържат взаимстването на атрибути между класовете (reuse), но с различен обхват на приложение съгласно принципите: наследяване се прилага при is-a отношение между деривата и базовия клас композиция (или агрегация) се прилага когато отношението е has-a (Пример: базови класове Person и University, класът Student може да бъде дериват на двата класа или да има атрибути с указатели към двата класа или комбинация от двата подхода Student IS-A Person → Student е уместно да бъде наследствен дериват на Person Student HAS-A University → Student е уместно да има атрибут с указател към University)
Наследяването е противопоказно за капсулацията (локалността) на кода, тъй като промяна на атрибут в базовия клас предизвиква каскадни промени в дериватите - пример *(фиг. 5.6.1 и 5.6.2)* - Student и Professor като деривати на Person (легитимно но ниска капсулация) и като агрегиращи PersonalHandler (с прозрачна конверсия на обръщението към атрибути).
ОО анализ: анализът предхожда проектирането и имплементацията и се състои в структуриране на предметната област и представянето й като набор класове с определена функционалност. Обикновено се състои в описание на потребителския сценарий чрез диаграма на случаите, от която се извлича и аналитичната (или принципна) клас-диаграма. 1) Диаграма на случаите (Ivar Jacobson 1987) – пример за OPS (Order Processing System) *(фиг. 5.7)* : определя типовете потребители на системата (напр. клиент, счетоводство, доставка); определят се основните случаи, които ще се детайлизират като [една или повече] операции в етапа на проектирането (напр. случая добавяне на изделие в пазарската количка би изисквал и операция със складовата БД).
2) Принципна клас-диаграма: е абстрактно описание на класовете на системата – по-близко до сценариите и функционалността, отколкото до имплементацията (не отчита производителност на модулите, технологии и технологичност на проектирането и експлоатацията). Състои се от гранични, същностни и контролни класове (boundary, entity, control). Граничните класове се извличат от интерфейсните случаи и са ориентирани към имплементация с GUI (Web форми, прозорци, браузър-плугини) или като междинни интерфейси (middleware wrappers) към други системи. Същностните класове отразяват информационния слой (напр. клиентската или продуктова идентичност са същностни класове). Контролните класове отразяват отделните случаи т.е. операциите, които свързват граничните и същностните класове (пример *(фиг. 5.8)* – принципна КД на OPS.
ОО проектиране: проектирането е самостоятелна фаза в развоината дейност на разпределените системи (Може да се приложи подход, различен от този на фазата на анализа – потоков(event driven), контекстен (data driven), структурен (с функции)) Целта на проектирането е декомпозиция на системата на технологични модули – (класовете се описват с техния интерфейс т.е. публичните им атрибути и операции, и се специфицират след това на фазата на имплементацията). Различават се високо и ниско ниво на проектирането. Високото ниво идентифицира класовете напр. с приложение на CRC-карти и клас-диаграми за статичните отношения (specification/compile time) между класовете. Ниското ниво детайлизира проектираните класове и тяхното динамично взаимодействие (run time) с диаграми за взаимодействието (най-често с диаграми на последователността или на комуникациите) и на машината на състоянията (state machine) – като се използват диаграмите на случаите от фазата на анализ.
1 стъпка - прилагат се CRC карти (Class-Responsibility-Collaborator – Kent Beck & Ward Cunningham, 1989) и/или клас диаграми за пълно (а не принципно като при анализа) описание на класовете. CRC картата на всеки клас съдържа таблица с описание на името, функционалните задължения (responsibility – заданията които изпълнява + контекста им) и списък клобориращи класове за изпълнение на тези задължения (пример за OPS от *(фиг. 5.8):* RegistrationPage и RegistrationController – *(фиг. 5.9).*
2 стъпка - описва се взаимодействието между обектите от стъпка 1 и се прилагат диаграми на последователността или на комуникациите. Моделът се състои от последователни стъпки, описани чрез обмен на съобщения (пример – диаграма на последователността за случая Registration – описва обмена между класовете RegistrationPage и RegistrationController – (фиг. 5.9): В горната част на диаграмата са взаимодействащите обекти – с означения <object_name>:<class_name> (името на обект може да отсъства). Връзките отразяват дейностите на съответните обекти и носят съответните етикети – включително new за създаване на обект от клас-колаборатор. В примера само обектите successPage и failurePage са именувани – за разлика от останалите класове – тъй като се предават алтерантивно от RegistrationController към RegistrationPage)
3 стъпка - описва динамичното поведение на по-сложните класове за целия им цикъл на живот (напр. контролните класове) – с диаграми на машината на състоянието. ДМС се извлича от диаграмите на случаите, в които участва дадения клас. В ДМС отделните състояния означават стабилност на колекцията от променливи на средата и от вътрешни променливи на класа. Вътрешните променливи на класа обикновено задават граничните стойности, с които се сравняват променливи на средата (условие за преход между състоянията на класа) и евентуално се изпълнява преход в друго състояние. За по сложните класове ДМС е съставна – включва и sub-state диаграми, но:Сложният клас е желателно да се представи от няколко класа, ако логическата му функционалност не се описва от едно просто изречение; това се отразява обратно и в CRC-модела.
4 стъпка – представлява подробно описание на интерфейсите на всеки клас – изброяват се атрибутите и операциите и тяхната публичност (с + и - в UML). Публичната част от интерфейса е фиксирана и не трябва да се променя в следващата след проектирането фаза – имплементацията. Публичният интерфейс се състои главно от дефинирани константи и операции: Операциите в публичния интерфейс са 4 категории:конструктор, деструктор, аксесор и мутатор. Определянето на публичните атрибути (константи) се базира на следните фактори: 1) Какви са външните стойности, които класът използва в своите операции – от CRC-диаграмата – напр. класът RegistrationPage използва Име и Парола *(фиг. 5.10)*. 2) Какви са възможните състояния на класа от ДМС – те се включват като атрибути (но обикновено частни). 3) От мощността на асоциациите: 1..1 асоциация изисква скаларен атрибут-указател към асоциирания клас, а 1..* асоциация – атрибут-колекция (вектор). 4) Други атрибути, необходими за изпълнение на операциите – обикновено са локални.
Предимства и недостатъци на ОО архитектурите: предимства: непосредствена връзка с потребителските сценарии и проблемната област; взаимстване (reuse) и капсулиране на имплементацията; лесно допълване чрез полиморфизма и класовете-деривати; устойчивост на системата, поради защитеност на локалните атрибути; удобен преход към други модели и най-вече към компонентна архитектура. Възможни проблеми: непредвидени странични ефекти при взаимодействието на много обекти, включително при асоциации 1..*; интерфейсите и вътрешната имплементация на класовете – макар и пордукт на отделни фази – не са толкова разграничени, колкото при компонентните архитектури. Обикновено се разработват съвместно, което снижава нивото на абстракция (и сложност) на цялата архитектура, а също обичайно води до по фина грануларност в сравнение с компонентните архитектури; наследствеността между класовете често води до грешки в спецификацията и следва да се прилага много внимателно.
Потокови (Data Flow) архитектури: представят обработката като последователност от трансформации (т.е. групи операции) върху последователност от набори структурирани еднотипни данни. Системата се декомпозира на функционални модули или подсистеми (паралелизъм по управление - аналогия с [нелинейните] конвейри). Интерфейсът между модулите може да е във формата на потоци (streams), файлове, канали (pipes, асинхронни потоци) и др. Основният паралелизъм е по данни, тъй като ритъмът на обработка се задава от наличието на данни за обработка. По тази причина – отсъствието или минимизирането и импликацията на контролния поток – ПА са подход и стил, приложим предимно при автоматизирани процеси на обслужване – напр. езикови компилатори, автоматизирани системи с пакетно обслужване като разпределените транзактивни системи, вградените системи. Топологията на пренос на данните между модулите се задава експлицитно с блок-диаграми *(фиг. 5.11)*. Обработката е асинхронна. Модулите поддържат само интерфейс по данни, не и контролен интерфейс и не се адресират взаимно – адресацията е само чрез предаваните данни. По механизма на свързване между модулите (т.е. на обмен) се разграничават: пакетна обработка (Batch Sequential), филтрирани канали (Pipe & Filter) и контролни процеси (Process Control).Фиг.5.12
1.Пакетна обработка (Batch Sequential): Това е най-старият модел на СА за обслужване в транзактивни системи и класическите ОС със стандартен файлов IO и редиректори. Приложението е скрипт с команди за изпълнение на съответните модули в UNIX, DOS, Tcl/Tk – напр. myShell.sh

```
exec searching kwd <inpitFile >matchedFile
exec counting <matchedFile >countedFile
exec sorting <countedFile >reportFile
```

Този стил е приложим и в съвременните ОО езици, където отделните обработващи модули, входът и изходът се представят като методи и атрибути на класа. Приложимост: 1) Данните (включително междинните резултати!) са оформени в пакети – файлове, т.е. с последователен достъп. 2) Модулите се представят като програми, които се активират със скрипт или като резидентни модули, които сканират входните си файлове. 3) Неприложима СА за интерактивен интерфейс. 4) Широко приложение за асинхронни паралелни процеси – данните се декомпозират като множество входни файлове, а обработващите модули се репликират в множество възли (принцип на обслужване в пакетната фонова обработка – Condor, Boinc).Фиг.5.13
2.Филтрирани канали (Pipe & Filter): приложението се декомпозира на източник на данните, филтри, канали(pipes) и консуматор на данните (sink). 1) Данните са последователни FIFO потоци (буфери, опашки) от байтове, символи или записи, които представят в последоветилен вид всички структури – вкл. и по-сложни, които се сериализират – (в ОС marshalling/unmarshalling). 2) Филтрите трансформират потока данни – без необходимост да изчакват готовност на целия пакет за разлика от пакетната обработка! Те записват изходните данни в канал, който ги предава на друг асинхронно работещ филтър. Има 2 типа филтри: 1*) активен филтър – изпълнява операциите pull/push върху пасивни канали – каналите осигуряват съответните операции, а инициативата е на филтъра. В Java PipedWriter и PipedReader класовете предоставят този интерфейс за канали 2*) пасивен филтър – предоставя push/pull интерфейси на каналите. 3) Каналите преместват, а по същество съхраняват, потока данни, които се обменят между два филтъра.
Клас-диаграма на СА с филтри и канали *(фиг. 5.14)* – активният модул е с плътни интерфейсни линни. Филтърът е свързан с до 3 класа – източник на данните, консуматор и канал.
(Блокова и последователностна диаграма на ФКСА – *(фиг. 5.15)*. ФКСА се организира лесно в пакетните ОС (напр. в Unix who | wc -l означава пасивен канал между две операции – в случая who генерира списък от потребителите, wc брои думите в списъка (спрямо стандартни разделители); поддържат се канали с имена, а филтри могат да са произволни процеси в основен и фонов режим (fore- и background)))
Макар, че управлението е по данни, паралелизма е управление и архитектурата е приложима, когато обработката може да се раздели на асинхронни модули. Реализира се модела производител/консуматор. Не се поддържа динамичен и интерактивен интерфейс – ограничение, което е предимство при дадени приложения. Приложението се ограничава от формата на данните в каналите – обикновено се използва ASCII код.
3.Контролни СА: Прилагат се при вградените системи (ВАС) – компютърно контролиране на процеси в реално време с или без човеко-машинен интерфейс. При вградените системи управлението е на база на сканиране на променливи на средата, извличани като поток данни от сензори и управляващо въздействие чрез компютърно контролирани актуатори (напр. автомобилен ABS – *(фиг. 5.16)*).
При КСА процесът се разделя на няколко модула, но те са от 2 типа: 1) контролни модули – за следене и манипулиране на променливите на средата и състоянието. 2) изпълнителски модули – за управление на актуаторите. (Връзките между модулите са чрез поточни данни).
Типове контролни потоци при КСА: 1) контролирани променливи – характеристики на ВАС (сила на ток, налягане и др.), физически контроли на изпълнителните актуатори – те се измерват текущо от сензорите и се съпоставят с контролните константи т.е. целевите стойности. 2) входни променливи според проблемната област (скорост, налягане, температура, влажност, GPS координати).
Контекстни архитектури (Data Centric): характеризират се с централизирано хранилище на данните, които са достъпни за всички компоненти на системата, така че декомпозицията е на модул за управление на достъпа до данните и агенти, които извършват операции върху тях. Интерфейсът между агентите и данните може да е явен (напр. RMI или RPC) или имплицитен (напр. транзактивен). В чист вид КАрх не предвиждат преки комуникации между информационните агенти – *(фиг. 5.17)*.
Модулът данни изпълнява операции по извличане или регистриране и промяна на записи – по 2 въможни модела: 1) хранилище (repository) (с активни (инициативни) агенти). Хранилището е обикн. Е организирано като СУБД, CORBA, UDDI или Web-услуги. 2) черна дъска (с инициатива на модула данни). Агентите са абонати за събития (event listeners), които настъпват при промяна в данните и на които абонатите отговарат реактивно (често при AI-разпределени приложения, охранителни системи за разпознаване на звук и образ, системи за управление на бизнес ресурси – складове, транспорт).
1. Контекстни архитектури с хранилище: макар и с управление по данни, за разлика от потоковите архитектури за пакетно обслужване на транзакции, тези архитектури поддържат интерактивните UI.
Пример: клас-диаграма на университетска информационна система – *(фиг. 5.18)*.
Класът Collector поддържа вектор на колекция от студентски записи и затова агрегира клас Student, като поддържа UI за извличане, добавяне и промяна на записите за студентите. Класът Student е интерфейс към таблицата на студентите, чиито инстанции представят по един запис (т.е. ред) в нея. Диаграмата на последователността *(фиг.5.19)* представя споделянето на данните чрез класа Student между няколко клиенти. Релационните СУБД са обичайната платформа за имплементация на тези архитектури, тъй като поддържат свързаност (консистентност) на разпределения достъп до данните, както и множество системни средства за операции, базирани на метаданни. За по-висока отказоустойчивост и защита на данните се прилагат разпределени хранилища. Основен недостатък е статичната структура на данните – еволюция в структурата на релационните таблици се прилага трудно, струва скъпо и надеждността й се проверява трудно.
2. Контекстни архитектури с черна дъска: ориентирани са главно към проблеми, решими с методите на AI – най-вече разпознаване на шаблони в различни области (първите приложения от края на 1970те са експертни системи в метеорология, изображения, звук, молекулярна химия). Те декомпозират решаването на проблеми на два[+] дяла: 1) черна дъска, съхраняваща данни – факти и хипотези т.е. еволюционни модели над фактите 2) източници на знания – паралелно работещи агенти, които съхраняват различни страни (данни, организирани като знания) от проблемната област. Всеки ИЗ капсулира специфичен аспект от проблема и е отговорен за частни хипотези и решения като част от общото решение 3) [контролер – система за начално зареждане и управление на разпределеното приложение]. При контекстните арх. се запазва блок-д-та от *(фиг. 5.17)*, но контролният поток е само от ЧД към ИЗ: Имаме неявни (имплицитни) обръщения към регистрирание в ЧД агенти-източници. Обръщенията възникват при промени в данните и се предават към абонираните за тези промени ИЗ, които изпълняват реактивно заложените в тях логически правила за извод. Този асиметричен механизъм на обмен е известен като модел publish/subscribe (pub/sub) в общите комуникации. Контекстните арх. с черна дъска се класифицират като слабо-свързана (loosely coupled) РС поради асинхронния комуникационен модел с обмен на публикувани съобщения към абонатите (за разлика от силно свързаните (tightly coupled) системи с хранилища, където транзактивното обслужване е свързано със заключване на данните за конкурентен достъп). (Клас-диаграма на такава архитектура –> *(фиг. 5.20)*). Класовете-източници KnowledgeSource съхраняват специфичните правила за логически изводи, регистрират се в съответната ЧД, абонират се за оповестяване на промени в данните на ЧД и евентуално генерират реакции с изменения в локалния си или общ (ЧД) контекст; форматът на знанията и правилата за всеки ИЗ може да е специфичен. ЧД управлява общия котекст, регистрира промените в него, оповестява абонатите и регистрира евентуалните реакции, както и съхранява крайното решение. Контролерът инициира ЧД, множеството на ИЗ, инспектира състоянието им и публикува крайното решение.
(Последователностна диаграма на архитектурата –> *(фиг. 5.21)*).
(Блок-диаграма на КАЧД на система за туристически консултации –> *(фиг. 5.22)*).
Обединява множество резервационни агенции – пътни, хотелски, за атракции, за коли под наем, кредитни и т.н. Клиентските заявки се публикуват на ЧД и се оповестяват съответните агенти, чрез реакциите, на които се изготвят един или повече планове за туристическо пътуване и съответното финансиране. Всички операции се инициират по данни, а се поддържа и UI. Типично за КАЧД клиентският интерфейс през контролера е минимален. Примерно еднократен, но интерфейсът за управление на агентите може да е итеративен.
КАЧД е подходяща архитектура за комплексни неизследвани и особено мултидисциплинарни проблеми, които са без детерминистично решение и с представяне на котекста във формите на AI, както и неподходящи за търсене на решение с пълно обхождане на проблемния домен, поради изчислителната сложност или непълнота/неточности в данните. Може да се генерират оптимално или няколко субоптимални решения или решения на частни подпроблеми. Подходяща е за разпределена обратка с умерена скалируемост поради централизирания контекст. Проблем е еволюцията в структурата на контекста, поради обвързаност с агентите на знания. Отсъствието на междуагентни комуникации води до необходимост от централизираната им синхронизация (например приоритетна) на достъпа до общия контекст. Трудно се формулира условие за край на обработката, поради недетерминистичния характер на проблемите.

6. Йерархични, асинхронни и интерактивни модели на софтуерната архитектура. Организация, компоненти, разслояване. Методи на анализ и проектиране.

Йерархичните архитектури декомпозират системата по управление на йерархични модули - функциите се групират по йерархичен принцип на няколко нива. Координацията обикновено е между модули от различни нива (вертикална свързаност) и се базира на явни("заявка-отговор") съобщения. Ниските нива функционират като усуги към непосредствено по-високите нива. Услугите са имплементирани като функции и процедури или пакети от класове. Между нивата се постига пълна прозрачност при запазване на свързващите интерфейси, но имплементацията на услугите може да еволюира. Разслояването на архитектурния модел на много ОС (UNIX, Ms .Net) и на протоколните стекове е осъществено на следните нива: 1. Базови **услуги** - системните услуги се групират в модули за IO, транзакции, балансирано планиране на процеси, защита на информацията; 2. **Междинен слой -** "ядро" - поддържа проблемно-ориентирана логика - бизнес приложения, числова обработка, информационна обработка, като представя интерфейси към колекции от базовите услуги; 3. **Потребителски интерфейсен слой -** напр. команден еран, графични контролни прозорци, Shell скрипт интерпретатор.
Йерархията с подпрограми е традиционна архитектура, предхождаща ОО, базира се на процедури със споделен достъп до данните (има само частична капсулация). Декомпозицията е по управление, като комплексната функционалност на приложението се разделя на по-малки функционални групи - процедури подпрограми - с цел тяхното споделяне между различни извикващи ги модули. Актуалните данни са параметри на обръщенията към изпълнителнуте функции и могат да се адресират по: 1. указател - подпрограмата може да променя техните стойности на същия адрес; 2. стойност - подпрограмата получава стойностите като константи; 3. име - подпрограмата използва като аргумент локалната стойност за съответното име. Най-често това са локални имплементации на протоколи и други резидентни програми или динамични библиотеки. Главната програма управлява процеса на последователни обръщения към подпрограмите. Подпрограмите формиран нефиксирана, но ациклична слоеста йерархия - фиг. 6.1.
Диаграми на MS архитектура: потоковата диаграма се използва за начално моделиране на изискванията към системата:
*потокова диаграма на OPS (Order Processing) - местата отразяват обработката, а дъгите - преноса на данните (фиг. 6.2).
*Възел 1 - регистрация на заявките; възел 2 - валидиране и отказ (в. 4), или предаване на заявката; възел 3 приема или отказва заявка (в зависимост от изпълнимостта); възел 5 променя стоковата наличност и предава за фактуриране на възел 6; възел 7 обработва правилата за отказ и предава на възел 8 за уведомление (примерно друга оферта).
При анализа се идентифицират: 1. трансформиращите възли - променят формата на входните данни (напр. XML) към вътрешен формат - обикновено възлите с един вход и един изход. 2. транзактивните възли - обработват входящите данни и ги насочват към един или друг изходен поток или нямат изходящи дъги.
От контролната диаграма се извлича **блокова диаграма** на архитектурата, която е съставена от контролни и диспечерски модули (подпрограми), съответстващи респективно на трансформиращите и транзактивните възли на потоковата диаграма - фиг. 6.3.
Master/Slaves е вариант на архитектурата с подпрограми, който е специализиран към поддържане на допълнителни нефункционални изисквания като отказоустойчивост (fault tolerance) и надеждност, както и балансиране за ускорено изпълнение на заявките. Реализира се чрез репликиране на функционалните модули. Задачата му е алтернативно да оцени адекватността на паралелно обработените резултати от S_n - съществуват протоколи за отказоустойчивост, идентифициращи грешните и верни резултати при ограничен брой на изпълнителните реплики, и да извърши разпределяне на заявките, прилагайки принципите за товарен баланс. Фиг. 6.4 представлява блок диаграма и клас диаграма на Master/Slaver архитектура.
Обхват на ПП - архитектури: Подпрограмните архитектури са широко приложими при разделяне на функциите по принципа "отгоре - надолу"; приложими са и при ОО имплементация; проблем може да бъде достъпа до глобалните данни - те са модел на обща памет и затова са по-подходящи при мултипроцесорни машини, и обикновено аргументите на обръщението са указатели, а не стойности.
При **слоестите архитектури** се извършва групиране на различните нива в йерархията във функционално свързани слоеве от пакети класове, библиотелки от подпрограми (включени в заглавните файлове на проекта.). Интерфейсът на слоя се състои от интерфейсите на включените в него компоненти, а изпълняваната от тях функционалност (набора услуги) е протокола на слоя; интерфейсът му към нисколежащите слоеве те определя от техния интерфейс. Обработката се декомпозира на заявки от по-висок слой към непосредствения по-нисък. Възможно е прескачане ("bridge") в йерархията, което е нетехнологично, защото изисква поддържането на повече от един интерфейс към слоеве с услуги. Това се налага при необходимост от минимизиране на целия проект - напр. премахване на криптиращ слой. Протоколите на високите нива изпълняват приложно-ориентирани услуги, а на по-ниските - системно-ориентирани. Типично разслояване (Фиг. 6.5): потребителски интерфейс <-> бизнес слой <-> базови услуги <-> услуги на ядрото. Фиг. 6.6 - клас диаграма на слоеста архитектура с имплементация на общ интерфейс от всички слоеве.
При **компонентно-базираното разслояване** основен подход за капсулирането на услугите в слой е формирането на компонент, който се описва със своя интерфейс - напр .jar файл в JVM. jar - файлът представя всички класове от по-ниските слоеве и включва класовете от слоя, който имплементира. Компонентите на отделните слоеве формират пакета на платформата - Java API. Всеки клас от jar компонента е достъпен за приложенията чрез своя интерфейс - достатъчно е да е включен в променливата на средата classpath.
Модели на разслояване:
OSI: App ↔ Pre ↔ Ses ↔ Tra ↔ Net ↔ DLL ↔ Phy
Web-услуги – 6.10 и л-я 8.: SOAP ↔ XML ↔ HTML ↔ TCP/IP
Unix: shell ↔ core ↔ device drivers
MS .Net: CLR ↔ JIT ↔CTS
Net е технология, която осигурява платформата CLR (Common Language Runtime) за изпълнение на приложения на C#, VB.NET, C++/CLI аналогично на JVM – 6.7. За прозрачност и преносимост, приложенията се компилират до платформено-независим междинен език CIL (Common Intermediate Language). По време на изпълнение CIL кодът („управляван код“) не се интерпретира като при други виртуални машини, а се компилира по начин, известен като JIT (Just In Time) компилация в платформено-зависим машинен код (native code) – за конкретната хардуерна платформа и операционна система. Управлението на паметта, на нишките и процесите, защитата на паметта, верификацията и вътрешната компилация са системните услуги на CLR. CTS (Common Type System) дефинира всички базови типове данни и извършва конверсиите им. Тези типове са споделени между всички .NET езици и са стандартизирани в CLI.
Виртулните машини са слоест модел, който предоставя високо ниво на абстракция – програмен език или интерфейс за приложенията, при който скрива или обвива изпълнителната платформа. ВМ представя основните абстрактни функции на системата, като ги унивесализира без да ги променя – напр. скрива интерфейса към ОС, докато изпълнимите програми (C++) трябва статично да се прекомпилират за всяка ОС, както и за всеки тип процесор; понятието **емулация** означава изпълнение на функциите на дадена система от друга система (с принципно различни функции или организация) – напр. емулация на Unix върху MSDOS/Windows или емулация на PDA и Smart/Mobile Phones от настолен компютър. Примери за такива виртуални машини са:
Unix BM – 6.8, MS .Net BM – 6.7, JVM – 6.9.
Обхват на слоестите архитектури: слоестите архитектури се прилагат за еволюционна развойна дейност, при която нивото на абстракция се повишава – принципа на проектиране е отдолу-нагоре. Всеки слой може да се разглежда като виратуална машина от определено ниво. във високите слоеве се постига значителна прозрачност и преносимост на кода, а в в ниските – възможности за взаимстване на код (reuse) чрез промяна и добавяне на класове при запазен интерфейс на слоя. Подходящи са за компонентни имплементации. В сравнение с MS архитектурите има висок sистемен свръхтовар и по-ниска производителност. Свръхтоварът може да се преодолее с “мостове” през слоевете, но това намалява предимствата и смисъла на обща виртуализация. Слоевете имат тенденция да скриват настъпването на изключения от по-ниско ниво.
Асинхронните архитектури се базират на неявни (implicit) асинхронни обръщения между обслужващите процеси. Асинхронният обмен може да бъде: 1.в реално време (online) – без буфериране – и двата процеса трябва да са активни, но не блокират изчакващо точката на обмен – 6.10; 2.независим (offline) – с опосредяващ обмена процес-буфер на съобщенията; приемащият процес може да не е активен в момента на изпращане на съобщението и обратно. Активният процес генерира съобщения, а пасивните процеси ги получават и евентуално изпълняват реакция: прилагат SW-шаблоните Производител /Консуматор (Producer/Consumer) или Издател/Абонат; управлението е по събите (event driven) , където събитието е издаване на съобщение от издателя и получаване на съобщение от абоната. В независимия вариант процесът-буфер алтернативно може да служи като: 1. централизатор Message Topic на всички издадени съобщения и да ги препраща тематично до абонатите – един-към-много обмен; 2. Резервирана опашка Message Queue за един-към-един обмен.
При **небуферираните асинхронни слоести архитектури** ситемата се декомпозира на две или повече части: 1.генератори на събития (sources); 2.слушатели на събития (event listeners); 3.регистратори на събития, които опосредяват обмена и по-конкретно поддържат асинхронността и неявното (непряко) оповестяване на слушателите. При архитектурния модел на SmallTalk приложенията *n* пасивни графични компонента-слушатели $View_n$ се регистрират в пространство на събития EventSpace за съобщения от даден генератор на събития Model . Фиг 6.11: клас диаграма на архитектурата: 1. класът Event Source осигурява операции за регистриране на слушател и за уведомление за събитие; 2. класът Event Listener осигурява операция за анализ на събитието и генериране на реакция. Небуферираните асинхронни СА са подходящ модел за приложения с GUI и слабо-свързана логика, чиито модули се представят с машина на състоянията и имат недетерминистично поведение (поради което по-сложна настройка и тестване). Налична е значителна поддръжка от междинни компоненти. Елемент на синхронност е началната регистрация. Имат сравнително ниска производителност и голям системен свръхтовар.
При **буферираните асинхронни СА** системата е контекстна (data-centric); слабо свързана (не се чака потвърждение за получаването на съобщенията и обикновено не се получава отговор след обработката) , но с надежден обмен; декомпозира се на 3 части: генератори на съобщения (producers), консуматори на съобщения, услуга за асинхронен буфериран обмен на съобщения – MOM (Message Oriented Middleware). Имат висока скалируемост, надеждност, p2p и CS приложения. Използват се за системна поддръжка (мрежи, телекомуникации), бизнес приложения (бюлетини – новини, метеорология, групи по интереси; транзактивно банкиране и е-търговия). Поддържат се опашки (Message Queuing, MQ) и тематичен обмен (Message Topic, Publish/Subscribe Messaging P&S). Атрибути на съобщенията са: ID, заглавие (header) и тяло. Клиентите на системата обменят съобщения инициативно или пасивно, като адресацията е на базата на идентификатор, получен при началната регистрация на клиента в услугата за обмен.
MOM(Mssage oriented middleware): MS MQ, IBM WebSphere MQ (бивш MQseries), JBossMQ (Java Message Server), Oracle [бивш BEA] WebLogic JMS са системи, които осигуряват преноса на съобщения между различните изчислителни възли и и изчислителни единици и отговарят за предаването на съобщения между тях.
p2p (point-to-point) обмен: Обменът е 1:1 – всяко съобщение има точно 1 получател. Елементи: изпращач на съобщения, получател и асоциирана с получателя опашка, която поддържа асинхронността на обмена. Съобщенията до даден клиент-консуматор се съхраняват в неговата опашка-буфер до извличането им или до изтичането на срока им. Фиг. 6.12: – блокова диаграма на p2p обмен в EJB (Enterprise Java Beans – Java компонентна библиотека за бизнес приложения): получател е MDB (Message Driven Bean); изпращач е клиентски процес; опашката може да се организира чрез JMS API – системно приложение за поддържане на универсален асинхронен обмен.
Pub/Sub (P&S) обмен: тази СА се базира на централизатор (hub), поддържащ асинхронния и непряк обмен на съобщения между издатели и абонати по теми (topics) – тип бюлетин. Инициативата в обмена принадлежи на източника на съобщението – на издателя спрямо бюлетина и на бюлетина спрямо абоната – така се постига максимална асинхронност. Вариант е устойчивия абонамент (durable subscription), при който абонатът получава и съобщенията по дадена тема, издадени преди неговата регистрация в бюлетина. Фиг. 6.13: блок-диаграма на P&S СА – системата се базира също на JMS MDB/EJB, но за разлика от p2p при P&S крайните получатели на дадено съобщение могат да бъдат повече от един – всички регистрирани (и евентуално бъдещите) абонати по темата (или темите), за които е издадено съобщението. При разгърната P&S СА клиентите – издатели и абонати – са отдалечени разпределени процеси без никаква явна връзка помежду си, като абонатите обикновено изпълняват информационни услуги за трети клиенти – напр. сесии със СУБД.
JMS комбинирана (p2p + P&S) СА: фиг. 6.14: клас-диаграма и д-ма на последователността. По отношение на услугата на обмена клиентите (производители и консуматори на съобщения): се регистрират, откриват сесия за изпращане или приемане на съобщения, създават опашка или тема. JMS (и др. MOM) поддържа следните контроли за надеждност и QoS на обмена: обмен с потвърждение от опашката/бюлетина, означаване на съобщението като обмен без загуба, установяване на приоритет на съобщенията, срок на съобщението (expiration).
Обхват на асинхронните СА: Плюсове: подходящи са за слабосвързани системи с устойчив неявен обмен на съобщения, при които обменящите процеси са анонимни – не знаят идентичността на комплементарния процес/и (вкл. неговия интерфейс); има времева и локационна независимост; висока скалируемост и заменимост на компонентите; подходящ за динамично настройваеми разпределени изчисления (при асинхронен алгоритъм); подходящи СА за пакетна обработка; подходящи за интегриране на наследени приложения (legacy systems) в съвременни проекти.
Минуси: независимостта между обменящите клиенти ограничава логиката на приложенията: логиката на клиентите трябва да е независима от получаването (и неполучаването) на конкретни съобщения, не се идентифицира източника и няма пряк обмен с него; усложнена логика на клиентите поради изскването за гъвкавост, т.е. всеки клиент се самоконтролира (контраст с йерархичните и центарлизираните системи); възможност за тясно място (bottleneck) – по време (производителност на опашката/бюлетина) и по пространство (размер на опашката/бюлетина).
Интерактивни софтуерни архитектури: поддържат интензивен потребителски интерфейс; за целта декомпозицията на системата е на 3 функционални модула: **модул за представяне** (изглед) – с потребителски интерфейс – за представяне (в т.ч. графично или мултимедийно) на изходните данни и също намеса на потребителите в обработката (вход за данни и контрол); **модул данни** – поддържане на данните с базова функционалност върху тях; **модул за управление** – системни комуникации, управление на процесите, инициализиране и конфигуриране на модули данни, управление на изгледи. Поддържа се множество (адаптивни) изгледи за даден набор данни; слабо свързана архитектура, която поддържа явни и неявни обръщения към метод – респ. RMI и модел регистрация / уведомление(notification). Съществуват две категории ИСА: PAC (Presentation-Abstraction-Control) и MVC (Model-View-Controller) - аналогията е P-V, A-M и C-C; прилагат различно управление: PAC е с йерархично (разслоено) и разпределно управление, при което системата се формира от набор коопериращи агенти на три нива – базово ниво агенти на общи данни и бизнес логика, ниво на изгледите за локални данни и средно ниво агенти координатори на изгледите; всеки агент интегрира P, A и C компоненти; в MVC агентите са равнопоставени.
MVC: основен модел за сърверни приложения с Web-клиенти за достъп – е-бизнес, е-управление, системи за потребителски профили и т.н.; специализация: промени в контекста (данните) се представят динамично т.е. в реално време при отдалечени клиенти; приложна компонентна платформа за проектиране на MVC е напр. Java Swing. Трите дяла на MVC имат следната специализация: **контролерът** регистрира, подрежда и предава последователността от потребителски заявки; настройва изгледа вкл. динамично и управлява останалите модули ва СА – стартиране, настройка, обмен; **моделът** изпълнява базовите функционлни услуги, като капсулира контекста (непрозрачна обвивка на данните); при СА MVC I той не поддържа пряк интерфейс с присъединените към него изгледи; **изгледът** е динамично настройваемо графично представяне на заявена част от контекста.
MVC I е компактна и базова двуделна имплементация на MVC, при която контролерът и изгледът са интегрирани в един модул C/V. C/V се регистрира и присъединява към даден модел като се абонира за уведомления за промени в контекста, които представя в реално време, и служи като вход/изход на модула за данни – фиг. 6.15. C/V поддържа форми за потребителски вход – текстови полета, радиобутони (за алтернативен или множествен избор на опции) и др. ; при промяна във входните данни ги валидира и генерира заявка към модела с новото съдържание; представя резултата, който се генерира от модела (на базата на заложената в модела функционалност); представя промените в контекста на модела, за които е абониран (без заявка от потребителя) – модел “активен контекст”. MVC I е приложим за по-прости приложения с компактен GUI.
При **MVC II** контролерът и изгледът са самостоятелни, а евентуално и отдалечени процеси. Допълнителна функция на контролера е да инициализира връзката между изгледа и модела и да управлява обмена между тях. контролерът и изгледът се ригистрират в модела и се уведомяват разпределено за промените в контекста. Разделянето позволява самостоятелна имплементация и технологии за V и C - това способства за проектиране на сложна функционалност и също за самостоятелна еволюция на двата модула – по-специално на изгледите, които се поддържат от бързоразвиващите се графични технологии. Фиг. 6.16 - блок-диаграма, клас-диаграма и последователностна диаграма на MVC II: инстанциите на класовете V и C са “сдвоени”, като множество двойки се поддържат от един модел; класът модел агрегира колекция от класове с различни функции върху базата данни.
MVC II с Java: фиг. 6.17 - блок-диаграма на MVC II СА, базирана на Java технологии: **JSP** (Java Server Pages) се използва за V; EJB (Enterprise Java Beans) + JDBC (Java Data Base Connectivity) се прилагат за развитие на M; C може да се имплементира като поризволно сърверно приложение – напр. с Java Servlet технологията (сървлетите са Java сърверни приложения без потребителски интерфейс, които се инициализират от резидентни сърверни програми - напр. Tomcat – подобно на аплетите, които обаче се изпълняват в клиентската част от браузъра); **контролерът** получава потребителска заявка от графичен или текстов интерфейс (1), стартира необходимата инстанция на модела (2), селектира и стартира необходимия изглед (3) – с което управлението се предава към изгледа; **изгледът** получава данни от модела (4) и ги представя графично (5). **Обхват на MVC:** това е базовата архитектура за приложения с интензивен потребителски В/И с динамично представяне на данните и с възможност за самостятелна имплементация на модулите; поддържа се от множество професионални платформи за шаблонно развитие на приложенията; не поддържа агентно-базиран информационнен обмен, характерен за системите с редуциран потребителски интерфейс – автономни и вградени системи, роботи, автонавигатори и др.
PAC: PAC е развитие на MVC, което поддържа агентен обмен на съобщения. Системата се състои от множество специализирани агенти, декомпозирани на трите модула – P, A и C. Декомпозицията на даден агент разделя неговия потребителски интерфейс (P) от функционалността, която поддържа (A) и от модула му за обмен с др. агенти (C) – фиг. 6.18: **Презентационния** модул на агента е опция (съществуват агенти-посредници без потребителски интерфейс); **контролният модул** е задължителен - освен комуникациите с отдалечени атенти, той управлява достъпа до функциите на агента – P и A са слабосвързани процеси без пряк обмен. **Абстрактният модул** капсулира данните и операциите на агента.
PAC-приложение: фиг. 6.19 - примерно PAC-приложение (клас- и последователностна диаграма) за преглед на отдалечен страницииран документ: 1. с 4 бутона – за първа, предишна, следваща и последна страница – поддържани от агентите Æ2 ÷ Æ5 съответно; 2. Æ6 – за графична интерпретация на страниците от документа по съответен стандарт; 3. Æ1 е агента за достъп до документа в БД; 4. C1 приема заявките от Ci (i = 2 ÷ 5), настройва A1 на съответната страница, приема я от него и я предава на заявителя – Æ1 няма нужда от P1; 5. C1 съобщава на Ci за настройки на бутоните от Pi (напр. избледняване на бутони “следваща стр.” и “последна стр.” ако прегледа достигне последната страница) и предава на C6 съдържанието, което се представя от P6; 6. Ai поддържат контекста на съответните агенти – напр. предпочитан изглед на бутон, текущото му състояние; 7. A6 поддържа контекста на представяната страница – напр. декодиращ метод, кеширани страници.
Обхват на PAC: прилагат се за интерактивни системи от коопериращи специализирани информационни агенти; има слабосвързана разпределена система – комуникациите са неблокиращи асинхронни; плюсове: добри възможности за заменимост, еволюция на агентите, ескалиране на системата; поддържа еднакво многонишкови и многопроцесни разпределени приложения. Минуси: значителен свръхтовар особено при групови комуникации; непряк (бавен) обмен между контекста и представянето му; имплементацията на A и P е зависима от тази на C – затруднение при проектирането; усложнени операции за откриване на броя и идентифициране на текущите агенти

7. Въведение в Erlang

C >> ще отбелязваме промпта на интерпретатора на Erlang. След всеки написан израз, в интерпретатора се показва какво се връща от израза(expression-а) като стойност.
Плаващи стойности и основните операции с тях:

Дробно деление	Целочислено деление	Остатък при деление
>>5/3. 1.66667	>>5 div 3. 1	>5 rem 3. 2

Дефиниране на променливи(променливите са с главна буква!)
>> Pi = 3.14159. 3.14159 >> R = 5. 5 >> Pi * R * R. 78.5397
Данните са по същество атомарни (неделими) константи: 1) цели или плаващи числа: 123, -789, 3.14159, 7.8e12, -1.2e-45. 2) атоми – приличат на етикети в някои други езици. **Съставни типове** – комбинации от атомарните типове: 1) колекции(n-торки) 2) списъци
Атомите са неявно-глобални дефиниции (без include – клауза както в C), символите им са с водеща малка буква. Примери: red, december, cat, meters, yards, joe@somehost, a_long_name; При необходимост от водеща главна буква, атоми с имена на запазени оператори и празни места се използват единичите кавички, когато се дефинират атомите: пр.:’Monday’, ’Tuesday’, ’+’, ’*’. Атомите се различават от променливите само по глобалния си обхват. Интерпретират се като текстови (нечислови) константи. **Булевите стойности** в езика не съществуват като самостоятелен тип, а представяват съответните атомарни константи(true, false) пр: >> 1==2. False; >> 1<2. True; >> a>z. false >> less<more. true
Вградена е функцията is_boolean, която проверява дали дадено “нещо” е boolean атом пр.: >>is_boolean(9+6). False >> is_boolean(true). True .**Логически операции в езика:** and – булево и andalso – съкратен(мързеливо) and – връща false ако първия аргумент е false, без да оценява втория or – булево или orelse – съкратен(мерзелив) or – връща true ако първия аргумент е true, без да оценява втория
xor – булев xor(exclusive or) not – булево “не”пр.: >> not((1<3) and (2==2)). False.
Колекциите са редуциран аналог на структурите в C – набор от [разно|едно]типни колекции неименувани стойности напр. атом и плаващо число: >> PersHigh = {Ivan, 1.78}. Съставни (nested) колекции>> Person = {person,{name, Ivan},{height, 1.78},{shoesize, 42},{eyecolour, blue}}. **Адресиране** в колекциите:>> Point = {point, 10, 45}.
{point, 10, 45}. >> {point, X, Y} = Point. {point,10,45} >> X. 10 >> Y. 45. В Ерланг по принцип присвояването и адресирането изглеждат по-тромаво поради принципа за еднократно. Адресиране в сложни колекции: деклариране на структура от променливи, съответстваща на шаблона на стойностите:
>>Person={person,{name,{first,joe},{last,armstrong},{footsize,42}}.
{person,{name,{first,joe},{last,armstrong}},{footsize,42}}
Пример за коментар: >> {_,{_,{_,Who},_},_} = Person. % адресираме first
Списъци. Представляват набор от стойности или коилекции(евентуално с обща семантика, но може и да не са хомогенни) >> ThingsToBuy = [{apples,10},{pears,6},{milk,3}]. [{apples,10},{pears,6},{milk,3}]
Функции вурху списъци. Функциите са разположени в библиотечен модул lists:
>> lists:max([1,2,3]). 3; >> lists:reverse([1,2,3]). [3,2,1]; >> lists:sort([2,1,3]). [1,2,3] >> lists:split(2,[3,4,10,7,9]).
{[3,4],[10,7,9]}; >> lists:sum([3,4,10,7,9]). 33 >> lists:zip([1,2,3],[5,6,7]). [{1,5},{2,6},{3,7}] >>lists:delete(2,[1,2,3,2,4,2]). [1,3,2,4,2]; >> lists:last([1,2,3]). 3; >> lists:member(5,[1,24]). False; >>lists:member(24,[1,24]). True >> lists:nth(2,[3,4,10,7,9]). 4 >> lists:length([1,2,3]). ** exception error: undefined function lists:length/1. >> length([1,2,3]). 3. **Операции върху списъци**: Освен [...|...]-оператора, се прилагат дясно-асоциативните ++ и --: >> [monday, tuesday, Wednesday]. [monday,tuesday,wednesday]
>> [1|[2|[3|[]]]]. [1,2,3]; >> [1,2,3] ++ [4,5,6]. [1,2,3,4,5,6]; >> [1,2,2,3,4,4] -- [2,4]. [1,2,3,4];
>> [1,2,3]--[1,3]--[1,2]. [1,2]; >> ([1,2,3]--[1,3])--[1,2]. []
Сравняване на типове(мачване). Pattern matching - <Pattern>=<Expression>
A = 10 % A := 10; {B, C, D} = {10, foo, bar} % B := 10, C := foo, D := bar; {A, A, B} = {abc, abc, foo} % A := abc, B := foo; {A, A, B} = {abc, def, 123} % грешка, няма присвояване;
[A,B,C] = [1,2,3] % A := 1, B := 2, C := 3;
[A,B,C,D] = [1,2,3] % грешка, няма присвояване; [A,B|C] = [1,2,3,4,5,6,7] % A := 1, B := 2, C := [3,4,5,6,7].
Анонимната променлива “_” е известна като “don’t care” – буфер за произволни стойности но без присвояване; използва се за изравяване на съставните типове.
Низовете представляват списъци от $- префиксирани символи или техните числови ASCII кодове; стойността им се връща в двоен апостроф
>> $A. 65; >> $A + 32. 97; >> $a. 97; >> [65,66,67]. "ABC"; >> [67,$A+32,$A+51]. "Cat".
Празният низ е празен списък. **Атомите** са текстови константи, които само могат да бъдат сравнявани, докато низ списък подлежи на повече операции. **Двоични стойности** се използват за компактно пакетиране на данни – приложимо за системи с ограничена памет (embedded systems):
1.при големи обеми от данни, разполагат се в паметта побайтово (т.е. компактно) вместо структурираното адресно пространство на списъците и колекциите, също така осигуряват бърз В/И – за поточни данни. 2. при супермалки структури – възможност за побитов достъп и пакетиране на няколко суб-байтови стойности в двоични байтове.**Побитово пакетиране в двоична структура:** >> Red = 2. 2;
>> Green = 61. 61; >> Blue = 20. 20; >> Mem = <<Red:5, Green:6, Blue:5>>. % Mem заема 2 байта в паметта;
Функции: Функцията е последователност от едноименни клаузи, която връща стойността на последния израз от първата оценена клауза ,следващите клаузи се игнорират. Реда на клаузите е без синтактично значение, но може да има семантично значение. Не се интерпретират, а се компилират – вписани по няколко функции в компилационни модули. Пример: едноаргумантна функция area, трансформираща колекция в число (лице на фигура) или грешка:

```
area({rectangle, Width, Ht}) -> Width * Ht;
area({circle, Radius}) -> math:pi() * Radius * Radius;
area({square, X}) -> X * X;
area({3angle, A, B, C}) -> P2 = (A + B + C)/2,
math:sqrt(P2*(P2-A)*(P2-B)*(P2-C));
area(Other) -> {error, invalid_object}. % тази клауза тр.
% да е последна, тъй като Other се изравнява винаги успешно
```

Рекурсивни функции: Променливите са локални за клаузата в която са дефинирани

```
factorial(0) -> 1;
factorial(Nmbr) -> Nmbr * factorial(Nmbr-1).
```

Гардът е условие за изпълнение на функционалната клауза, той има булева стойност и е логическо И от [набор] операции за сравнение и/или вградени (но не потребителски) гард-тестове, въведени с ключ when и оценени в недефиниран ред няколка гарда се разделят със запетая
factorial(N) when N > 0, integer(N) -> N * factorial(N - 1); factorial(N) when N == 0 -> 1.
Вградените гард-тестове са true ако: atom(X) % X е атом; constant(X) % X не е списък или колекция;
float(X) % X е дроб; integer(X) % X е цяло число; list(X) % X е списък вкл. и []-оператор и др.
Анонимни функции: Стойността на служебната функция fun с произволен брой аргументи и операции върху тях може да се присвоява на променливи-функции, пример за функция с един аргумент колекция, която връща колекция; колекциите се състоят от атом и [числова] променлива
>> TempConvert = fun({c,C}) -> {f, 32 + C 9/5}; >>({f,F}) -> {c, (F-32)*5/9}; >> end. #Fun<erl_eval.6.56006484>
>> TempConvert({c,100}). {f,212.000}; >> TempConvert({f,212}). {c,100.000};>> TempConvert({c,0}).{f,32.0000}
Приложението на анонимните функции е за създаване на супер- функции – СФ приемат АФ като аргумент и/или връщат АФ като стойност. **Служебният модул lists** експортира функции, които приемат като аргументи стойностите на fun-дефиниции [върху списъци]; напр: lists:map(F, L) прилага функцията F върху елементите на списъка L и връща списък от резултатите, lists:filter(P, L) прилага логическата функция P върху елементите на списъка и др.Пример – анонимна функция за проверка на четността като аргумент на библитечна списъчна функция: >> Even = fun(X) -> (X rem 2) =:= 0 end. % АФ Even
#Fun<erl_eval.6.56006484> ; >> Even(8). True; >> Even(7). False ;>> lists:map(Even, [1,2,3,4,5,6,8]). % АФ е аргумент. [false,true,false,true,false,true,true]; >> lists:filter(Even, [1,2,3,4,5,6,8]). [2,4,6,8].
Вградените функции на Erlang: Намират се в модула erlang и се изпълняват нетитпични за синтаксиса на erlang операции или системни обръщения. Списък BIFs: time(), length([1,2,3,4,5]), size({a,b,c}), atom_to_list(an_atom), list_to_tuple([1,2,3,4]), integer_to_list(2234), tuple_to_list({}).**Функцията apply** изпълнява функция Func от даден модул Mod върху списък от аргументи Args (когато стойностите на 3те аргумента се гененрират динамично!) apply(Mod, Func, Args), Mod и Func са атоми или изрази със стойност атом . Списък BIFs. Пр.: >> apply(lists1,min_max,[[4,1,7,3,9,10]]). {1, 10}. **Case & If:** Алтернатива на функционалните клаузи, добавена за синтактично удобство. Сравнение между клаузите и case код - функцията filter връща списъкът L само с тези негови елементи X, за които P(X) = true**.**
Аритметични операции в Erlang и техните приоритети:

Операция	Описание	Тип на аргументите	Приоритет
+X	+ X	Number	1
-X	- X	Number	1
X*Y	X*Y	Number	2
X/Y	X/Y	Number	2

bnot X	Побитова навигация	Integer	2
X div Y	Целочислено делене	Integer	2
X rem Y	Целочислен остатък	Integer	2
X band Y	Побитово и	Integer	2
X + Y	X + Y	Number	3
X - Y	X - Y	Number	3
X bor Y	Побитово или	Integer	3
X bxor Y	Побитов хог	Integer	3
X bsl Y	Аритметично местене на ляво	Integer	3
Xbsr Y	Аритметично местене на дясно	Integer	3

Модулите са група от функции,които образува компилационен модул, а програмата обикновено е група от модули. **Функциите** са: 1)**глобални** – адресируеми и от други модули чрез префиксиране на обръщението с името на модула: demo:double_(2); 2)**локални** - адресируеми само в рамките на модула, независимо дали вече са дефинирани; 3) може да имат еднакви имена, но да се различават по модул на декларация или даже само по броя входни аргументи (“arity”). Модулите се съхраняват в .erl файлове с името на модула и се декларират с директива module, а функциите се вписват в тях с export. Пример:
-module(demo).
-export([double_/1]).% Function/Arity – глобална
double_(Value) -> mul(Value, 2).
mul(X,Y) -> X*Y. % локална
Компилацията на модули се извършва със следната команда от интерпретатора: c(modul_name) % .erl суфикса се подразбира. Компилацията на модул се запазва в modul_name.beam в същата директория. Функциите от компилираният модул се изпълняват от Björn’s Erlang Abstract Machine – затова .beam.
Обръщение към функциите от външни модули: >>cd("/SPO/uprajneniq/primeri"). /SPO/uprajneniq/primeri
В началото на модула се разполагат неговите атрибути и директиви
–attribute(Value) % –module(ModulName) е задължителен атрибут
–compile(export_all).% експортира всички функции при компилация; ≡ c(Mod,[export_all]).
–import(Module, [Function/Arity,...] % всички външни функции
След компилацията функциите могат да се импортират със следните особености:1) Импортираните функции не се нуждаят от префикс с името на модула; 2)Допускат се произволни едноаргументни потребителски атрибути (освен вградените):– author(Name). -date(Date). 3) Обръщение към вградена функция module_info() за извличане на module_info() дефиниране атрибути в модула и аналогична конзолна команда Mod_name:module_info/0 % безаргументна версия; Mod_name:module_info/1 % аргумента е валиден ключ; m(Module) % конзолна команда за извличане на атрибути. Валидни ключове за module_info(key) са attributes. **Примери с функции върху списъци**: Поелементна функция F върху списъка L: [F(X) || X <- L].резултатът е списък no.: >> L = [1,2,3,4,5]. [1,2,3,4,5]; >> [2*X || X <- L]. [2,4,6,8,10]
Поелементните операции върху списък от колекции се базират на: 1)шаблон, съответстващ на всичките списъчни елементи (!) и 2) операционен конструктор.
Еднократно рекурсивно обхождане на списъка с модифициране на акумулаторите Length и Sum:
average(X) -> average(X, 0, 0).
average([H|T], Length, Sum) -> average(T, Length + 1, Sum + H);
average([], Length, Sum) -> Sum / Length.
Двукратно (с поелементна операция) и еднократно (с case) обхождане на списък, връщащо колекция от два списъка Odds и Evens: odds_and_evens(L) -> Odds = [X || X <- L, (X rem 2) =:= 1],Evens = [X || X <- L, (X rem 2) =:= 0],{Odds, Evens}.
Списъци и примери с права рекурсия: 1)право-рекурсивна сума на елементите на списък:
sum([]) -> 0; sum([Head | Tail]) -> Head + sum(Tail). % т.е. sum([2,3,4] = 2 + sum([3,4]));
2)в erlang рекурсията замества итерациите и може да се наложи изпълнението на значителен брой рекурсии – затова ефективността й може да е от значение;
3)съществуват статистически данни, че в някои случаи приложението на обратна рекурсия (с акумулация) води до по - бързо изпълнение на итерациите; 4)спрямо пряката рекурсия, функциите, прилагаши обратна рекурсия, имат един параметър в повече – акумулатор, който натрупва резултата от последователните итерации:
sum_reverse([],Sum) -> Sum;
sum_reverse([Head|Tail], Sum) -> sum_reverse(Tail, Head+Sum).
i)обръщението към такава функция става с “нулиран” акумулатор!:
>> sum_reverse(List,0).
Акумулаторът само привидно се отклонява от принципа за еднократното присвояване, всъщност натрупването става чрез независимият аргумент- акумулатор на обръщение към рекурсивната функция:
=> sum_reverse([2,3,4],0) => sum_reverse([3,4],2)=> sum_reverse([4],5) => sum_reverse([],9) => 9. Тъй като в съвременните версии на erlang са взети мерки за оптимизиране на правата рекурсия, при необходимост от бърз код (напр. реално време в микротелминал) може да се направи експериментално сравнение на двата стила.
h() - history % последните 20 команди; b() - bindings % стойностите на всички променливи; f() - forget % заличава стойностите на променливите; f(Var) - forget; X. % връща стойност на променлива; e(n) – evaluate % оценява n-тата предходна команда; e(-1) % оценява предходната; apply(Module, Function, Args) >>apply(lists1,min_max,[[4,1,7,3,9,10]]). {1, 10}.
Изключенията възникват при нерешмо сравняване на типове (pattern matching) – функционално обръщение, за което не сработва никоя клауза или обръщение към вградена функция (BIF) с невалиден аргумент – или се декларират явно в кода с обръщение към вградените функции exit(Why), throw(Why) и erlang:error(Why). Те предизвикват прекъсване с връщане към системата и извеждане на код за грешката – освен ако не са обработени програмно с израза try...catch - явната декларация на изключения служи за:
throw(Why) % документирано прекратяване, което потребителят може да обработи
exit(Why) % програмно прекратяване на текущия процес
erlang:error(Why) % нерешима вътрешна грешка
Обработка на изключенията с try...catch има следния синтаксис:
try FuncOrExpressionSequence of
  Pattern1 [when Guard1] -> Expressions1;
  Pattern2 [when Guard2] -> Expressions2;
catch
  ExceptionType: ExPattern1 [when ExGuard1] -> ExExpressions1;
  ExceptionType: ExPattern2 [when ExGuard2] -> ExExpressions2;
  ... % ExceptionType = [throw | exit | error]
after AfterExpressions
end
семантика: При нормално изпълнение на FuncOrExpressionSequence резултатът от него се сранява с Patterni, изпълнява се съответната последователност Expressionsi и резултатът от нея е стойността на блока try…catch. При изключение в FuncOrExpressionSequence, то се сравнява с ExPatterni и се изпълнява съответната последователност ExExpressionsi като резултатът от нея е стойността на блока try…catch. Кодът в AfterExpressions се изпълнява винаги (след FuncOrExpression-Sequence и [Expressionsi | ExExpressionsi]), но неговият резултат не се запазва като стойност на блока try…catch.
Обработка на изключения с catch: Catch е примитив, който конвертира евентуално възникналото изключение в колекция от атрибутите на изключението (и я връща като стойност).
Метапрограмиране с Erlang: Метапр ограмирането е управлението на процесите по време на изпълнение на програмата – т.е. средствата за динамична интерпретация на кода. Напр. може да се дефинира функция apply/3, която стартира извиква дадена функция с идентификация, модул и мощност на списъка аргументи, които не се генерират по време на изпълнение на програмата. Модулът erlang осигурява и редица други BIFs за метапрограмиране като : 1)управление и наблюдение на процесите; 2)разпределяне (mapping); 3)управление на вход/изход; 4)достъп до системни променливи – напр. функциите: date/0 връща колекцията {Year, Month, Day}, time/0 връща колекцията {Hour, Minute, Second}, now/0 връща колекцията {MegaSeconds, Seconds, MicroSeconds} спрямо 1. януари 1970., now/1 връща уникална стойност, дори и когато е извикана повече от веднъж за 1 мкС в даден възел, така че може да се ползва за времева марка по алгоритъма на Лампорт . 5)Достъпът до стандартния вход за всяка програма се задава с BIFs в io-модула, а до произволен файл – във file-модула; 6)Стандартният В/И е достъпен на ниво линия, брой символи, атом или напълно изчислен израз (Erlang term).
Изход: Неформатиран изход на израз е с io:write/1, а с прилагане на форматиращ шаблон – io:format/2. форматните шаблони включват следните ключове: 1) ~c – ASCII код се извежда като символ; 2) ~f – плаваща стойност в 6 символа; 3) ~e – плаваща стойност в е-формат на 6 символа; 4) ~w – стандартен израз; 5)~p – като ~w но за принтер – напр с интерпретация на нов ред.

8. Конкретно програмиране с ERLANG (конкурентни процеси – управление, обмен и синхронизация)
Процеси и планиране: Процесите са самостоятелни виртуални изчислителни машини, генериращи резултат или сервизна функционалност, които се поддържат от суперпроцеса на ОС и могат частично да се управляват чрез вградени в езиците системни обръщения. В erlang (поради наличието на собствена ВМ – ОТР и поради общата концепция в поддържа на конкурентността) управлението на процеси не е вградено, а е езиков компонент, като: бързо и лесно се създават (много голям) брой процеси на ниво програма (т.е. приложение) и erlang прилага 100% модела Обмен на съобщение за синхронизация и комуникация между процесите и 0% модела Общи променливи -> процесите са напълно самостоятелни (и евентуално асинхронни)
*** OTP** (Open Telecom Platform) - OTP is the open source distribution of Erlang and an application server written in Erlang. Contains: Erlang interpreter; Erlang compiler; a protocol for communication between servers (nodes); a Corba Object Request Broker; a distributed database server (Mnesia) and lots of libraries.
Модел на обмена: OTP поддържа мн. ефективен и бърз обмен на съобщенията по модела [D|**G**]**MMP** (Flynn-Johnson’s Distributed or **General Memory Message Passing**). Отказът от обща памет (**SV – shared variables**) – дори в мултипроцесорни архитекрури - е с цел премахване на на проблемите, свързани с общата междупроцесна памет – недетерминираност в състезателния достъп и други, вкл. Блокировка и взаимна блокировка (deadlock); при общите променливи те са преодолими, но опасността от възникването им нараства при по-голям брой процеси/нишки, т.е. при опит за по-фина гранулярност. При асинхронен GMMP няма блокировка, защото **винаги** съобщенията са 1:1 (дори при мултитаскинг), е дефиниран активен и пасивен процес в обмена и няма изчакване за потвърждаване.
Изключения в модела на обмена: OTP поддържа и множество от вградени функции, които позволяват съхраняването на стойности по даден „ключ“ (т.е. променливи с еднократно присвояване) и последващ достъп до тези стойности от други процеси по генериращия ключ – тоест на практика е възможно да се програмира и с общи променливи в erlang (GMSV). Тези BIFs (built-in-function) са групирани в модул, наречен Process Dictionary. Това е компромис с принципа GMMP с цел адаптиране на erlang към стила на повечето програмисти, обаче самите автори на езика строго не препоръчват програмиране с тези BIFs. Практичното приложение на двата модела може да се прецени например с еталонни експерименти за производителността на SV- и MP-приложения.
Конкурентни примитиви: Създаване на процес: пр: Pid = spawn(Fun)%.дефинирана функция като нов процес с Pid. Има неявна йерархия родител-настойник, тъй като само родителят разполага с Pid на наследника, комуникацията е асинхронна, свободна от блокировка. При предаване адресацията е на база Pid, пр.:
Pid ! M % изпраща на съобщение М до Pid без потвърждение, за групово предаване (multicasting) – рекурсивн, пр: Pid1 ! Pid2 ! Pid3 ... ! M % т.к. всяка !-операция връща М. Полученото М се сравнява последователно с Patterni и с опционален Guardi, като при успех се изпълняват Expressionsi. N.B. и при неуспех стойността на М се запазва в наследника Pid (а и в родителя) поради принципа на еднократното присвояване
Spawn: Аргументите на spawn са идентификаторът на функцията със съответния модул и списък с нейните реални аргументи, което може да доведе до синтактични грешки: пр: spawn(m, f, [a]) % коректно; spawn(m, f, a) % некоректно. Функция за списък на идентификаторите на текущите активни процеси: пр.: 1> processes(). [<0.0.0>, <0.2.0>, <0.4.0>, <0.5.0>, <0.7.0>, ...]. Функция за пълен списък на текущите активни процеси пр: 2> i().

Pid Register	Initial Call Current Function	Heap Stack	Reds	Msg
<0.0.0>	Otp_ring:start/2	610	2432	0
Init	Init:loop/1	2		
<0.3.0>	Erlang:apply/2	2584		
Erl_prim_loader	Erl_prim_loader:loop/3	6		
<0.5.0>	Gen_event:init_it/6	377	220	0

process manager: пр.: 2> pman:start(). <0.51.0> (+ GUI)
Системно планиране на процесите: Управлението на процесите в erlang е циклично, но по събитие. Управляващото събитие е изчерпване на лимита операции на процеса ИЛИ нерешима receive-операция (без готово съобщение за никоя от клаузите). Лимитът се задава с максимална брой операции (“reductions” – Reds на предходния слайд), които процесът може да изпълни преди да бъде циклично прекъснат. В някои версии лимитът статично е 2000 редукции, напоследък лимитът е настроен да варива в зависимост от броя на процеси в системата. За въздействие на планиращия процес (scheduler) се ползва BIF-а erlang:bump_reductions(Num)
Леки процеси и нишки: Ерланг-процесите са леки процеси (“lightweight”). Управлението им (създаване, планиране, контекс и обмен) се поддържа – и то много ефективно – от суперпроцеса конзола, а не пряко от ОС. За ефективна конкурентност, конзолата поддържа по една нишка за всеки процесор или ядро в даден възел и на базата на тези ОС-дефинирани конкурентни ерланг-процеси управлява произволен (до MaxProcNum) брой потребителски процеси. [ВМ на Java и C# стартира самостоятелна ОС нишка за всеки нов потребителски процес]. Предимството на единия или другия модел не е предварително ясно, но се доказва експериментално; ерланг е особено ефективен при масивен паралелизъм.
Процесен свръхтовар: * overload е процес, който недиректно регулира CPU usage-а на системата. Измерването му може да стане с еталонна програма за генериране на произволно множество процеси max(N)
Анализ на процесния свръхтовар: Всъщност ерланг конзолата може да поддържа произволен брой процеси – по-голям от предефинирания максимален брой (чрез +P ключ при стартирането на конзолата):
c:\> werl +P 500000
Регистриране на процеси: Освен идентификатори, процесите могат да се регистрират и със символни имена за по-удобно обръщение: пр.: register(AnAtom, Pid) % AnAtom трябва да е уникален; unregister(AnAtom) % злаичава регистрацията на жив процес; whereis(AnAtom) -> Pid | undefined % връща Pid или атома;
registered() -> [AnAtom::atom()|] – връща списъка на регистрираните процеси. Регистрираните процеси са достъпни чрез своя атом, както в обхвата на модула, така и от конзолата.
Итеративен и конкурентен сървер
Итеративен сървер: Итеративните сървери са процеси на обслужване, които изпълняват входен поток от [еднотипни] заявки последователно по реда на постъпване и евентуално [с непрекъсващи (non-preemptive)] приоритети. Алтернатива на итеративен сървер е конкурентен сървер: процес, който стартира нова нишка или извиква друг процес за изпълнение на всяка нова заявка.
Сървърен процес: Итеративен сървърен процес в конзолата на ерланг
1> Pid = spawn(fun My_server:loopSrv/0). <0.36.0>
2> Pid ! {circle, 23}. Area of circle is 1661.90 % Pid връща резултат {circle, 23} % конзолата връща съобщението като резултат от !
3> Pid ! {triangle, 2, 4, 5} is undefined. {triangle, 2, 4, 5}
4> Pid ! {rectangle, 6, 10}. Area of rectangle is 60 {rectangle, 6, 10}
В 1> итеративният сървер се стартира като паралелен процес на процеса ерланг-конзола – и двата процеса генерират резултати. Конзолата в случая е конкурентен сървер, но съвместява и клиентския (интерфейсния) процес – затова няма нужда от друг адрес освен генерирания Pid. Обикновено при клиент-сървер архитектура клиентският процес е самостоятелен отдалечен процес и при заявка към сървера (освен аргумента на функцията на сървера) е хеобходим като аргумент клиентския идентификатор (вкл. URL на клиентската конзола) – за връщане на резултата.
Обмен клиент-сървер: Клиентският Pid е аргумент на заявката към сървера заедно с дункционалния аргумент. По същата причина (за различаване на съобщенията от потенциално различни сървери) в клиента се връща и Pid на сървера освен резултата. Функцията rpc е [част от] клиентският код; тя адресира сървърния Pid с аргумент {Pid, Request} и изчаква съобщението Response, което връща като резултат.
Конкурентен сървер: Стартира самостоятелен обслужващ процес (или нишка) за всяка заявка. За целта интерпретира заявката и изпълнява многократно spawn като BIF, при което spawn влиза в loop на обслужването.
Избирателен обмен: Филтриране на постъпилите съобщения по даден признак – например по идентификатор на изпращащ процес и/или по поредност на съобщението – като останалите получени съобщения остават в кутията за евентуален следващ избор. В примера върху receive е дефинирана функцията decode_digit/1 и полученото съобщение ще се извлече от пощенската кутия на процеса Pid2 (и интерпретирана) само при съвпадение на дункционалния аргумент и първия елемент на колекцията-съобщение: (Фиг. 8.Y)
Срочен обмен: Дефиницията на receive може да се разшири с after клауза, чрез която се определя срок за изпълнение на обмена. Последователност на операциите:1) при достигане на процеса до receive се стартира обратен таймер от Time; 2)извлича се първото съобщение от кутията и се проверява последователно по шаблоните; при успех с Patterni се изпълняват Expressionsi и блока receive приключва, като съобщенията от буфер за чакащи се прехвърлят в кутията за преглед от бъдещо receive (където се сортират по ред на постъпване), таймерът се нулира, процесът продължава след receive,ако не съвпадне първото съобщение, то се прехвърля в буфера, а процедурата по търсене на съвпадение се повтаря със следващите съобщения от кутията, ако не се намери съвпадение, процесът, който изпълнява текущия receive се отлага, докато в кутията не постъпи ново съобщение; вече прегледаните съобщения не се връщат от буфера в кутията за преглед, ако таймерът се нулира се изпълняват TimeoutExpressions и блока receive приключва по същия начин, т.е. процесът не се отлага.
Групов обмен: Тъй като моделът на обмен между процесите се базира на MP(MapReduce), т.е. на бинарна операция, при групов достъп до данни (на повече от 2процеса) се налага обмен на Pid с цел всеки от процесите да има достъп до останалите процеси в групата. В примера е показан p2p модел на групов обмен: (Фиг. 8.Y)
Състезателен достъп и блокировка: Състезателният достъп – race condition – е проблем, характерен за GV(SM)-моделите – конкурентно блокиране на достъпа до общите променливи от един от процесите в групата: 1)race condition възниква при невъзможност да се определи статично кой от процесите пръв ще успее да изпълни критичната зона; 2)deadlock е взаимно блокиране на процесите при достигане до общи критични зони – особено при приоритетни процеси: високоприоритетен процес прекъсва нископриоритетен, който вече е заключил обща променлива. Принципно ерланг отстранява и двата разпространени в конкурентното програмиране проблема чрез MP-модела си принцип за еднократно присвояване (на практика всяка променлива е от тип CREW- Concurrent Read Exclusive Write): няма общи променливи, обменът е асинхронен и няма приоритети.
Състезателен достъп в ерланг: Пример за състезателен достъп – 2 процеса стартират едновременно
db_server:
start() ->
  case whereis(db_server) of
    undefined ->
      Pid = spawn(db_server, init, []), % тук P1 прекъсва
      register(db_server, Pid), % тук P1 се възстановява
      {ok, Pid};
    Pid when is_pid(Pid) ->
      {error, already_started}
end.
Процес P1 пръв изпълнява start/0 и whereis(db_server) връща undefined, поради което той създава сървера, но може по случайност да прекъсне поради изтичане на Reds; стартира се P2. P2 също създава сървера – тъй като той още не е регистриран – регистрира го и прекъсва също по Reds. P1 се възстановява, прави опит да регистрира „своя“ сървер, но прекъсва с грешка, че вече има регистриран процес с това име – вместо да получи колекцията {error, already_started}. Новите версии на ерланг-конзолата отчитат подобни ситуации но без 100% гаранция.
Приоритетни процеси: Въвеждането (по-скоро ползването) на процесен приоритет е срещу всички концепции в ерланг. Приоритетите обаче се считат за единствено надежно средство за изпълнение на RT-процеси в многопроцесна система. Ползва се BIF-а: пр.: process_flag(priority, Priority) % Priority = [high|normal|low]. Твърди се, че самата конзола в по-голямата си част се изпълнява в нормален приоритет, което: засилва възможността на RT-изпълнение на високоприоритетните процеси и опасно възможността за блокировка
Емулация на SV с MP: Достъпът до общите променливи, ресурси или услуги от конкурентни процеси (PRAM-[ER|CR]EW]) е чрез mutex semaphore (mutual exclusion). Семафорът е протокол на достъпа с операциите signal и wait. При опит за достъп до ресурса P1 изпълнява заключване чрез ресурсния семафор, викайки mutex:wait() и получава OK от свободния семафор; след достъпа ресурсът се освобождава от P1 с mutex:signal() както освобождаването се потвърждава с OK от семафора. Ако P2 опита да изпълни wait докато семафорът е заключен от P1, няма да получи OK и се отлага докато P1 освободи семафора – тогава семафорът връща OK на P1, а и на P2 в отговор на неговия wait (диаграми на машина на състоянието и последователностна диаграма: 8.25) **(Фиг. 8.Y)**? . В следващия пример взаимно рекурсивните функции free и busy отразяват устойчивите състояния на семафора, а обмена на съобщения съответства на събитията – т.е. преходите между състоянията. stop/0 прекратява семафора само ако той е free. Функцията terminate е необходима за да прекрати всички процеси, изпратили непотвърдени заявки wait след стартиране на stop/0. Семафорният протокол може да се оптимизира (и усложни).
* EREW – Exclusive Read Exclusive Write – само един процесор може да чете / пише в даден момент

9. Разпределено програмиране с Erlang (RPC и транспортни съединения)

Възли е всяка именувана ерланг-ВМ. При разпределената обработка, процесите се изпълняват във 2 или повече възела. Няколко ерланг-възела могат да се изпълняват от една физическа машина („хост").

Обмен в интранет. в интернет обменът се базира на SSL/TCP/IP –съединения (sockets). В интранет обаче обменът се базира директно на комуникационни примитиви. С тях се изгражда подходящ комуникационен модел- пример за това RPC. Синтактично обменът е прозрачен по отношение на локалността на процесите, но адресирането се разширява с името на възела: {a_process,host1,@EServ}
!My_message. Така редът на приемане запазва и реда на предаване. За да се върне на процеса From съобщение с името на локалния възел се използва функцията whe_re: whe_re(From) → From ! node(). Където From е pid.
Примерно стартиране на функцията whe_re от модул dist като процес в отдалечен възел host2@EServ с аргумент pid на родителя, където 2-та възела host1 и host2 са на общ сървъв EServ
(host1@EServ)1>spawn('host2@EServ',dist,whe_re,[self()]).
<4556.32.0>
(host1@EServ)2>flush(). – чете всички съобщения
Shell got 'host2@EServ' – възела на наследника
OK
За **идентификация** на възел (име) се използва 2-ката възел/хост и тя трябва да бъде уникална. Имената на възлите могат да бъдат дълги и къси
C:\>erl –sname a_name : дава a_name@host_name, късите са ОК в интранет
C:\>erl – name a_name : дава a_name@{IP адреса | URL}, дългите са OS sensitive!
В модула за мрежово програмиране net_kernel се съдържат основните функции за разпределена обработка
1> net_kernel:start([my_node, shortnames]).
{ok,<0.33.0>}
(my_node@VG-110-fmi)2> erlang:is_alive().
True
(my_node@VG-110-fmi)3> node().
'my_node@VG-110-fmi'
(my_node@VG-110-fmi)4> net_kernel:stop().
Ok
5> erlang:is_alive().
False
6>

При защитен обмен се ползва secret cookie т.е. споделен защитен ключ (парола) между обменящите възли. По този начин обменът е възможен само между възли с еднакъв ключ, а всеки възел има по един дефиниран ключ – атом в даден момент.
C:\er; -sname a_name – setcookie blah. Възелът може да стартира и без дефиниран ключ и съответно ще получи служебна стойност на ключа от файла erlang.cookie. Ако този файл/стойност не са дефинирани, то те се създават служебно в областта на текущия потребител. По този начин се допуска обмен м/у процесите на един потребител по подразбиране. Би било добре ако разпределеното приложение предефинира специален ключ във всички свои процеси.

Cookies са разрешаващи обмена ключове. Установяването им винаги е на ниво възел и става при стартиране на конзолата **(erl - setcookie SFEWRG34AFDSGAFG35235 – name a_node),** запис в контролния файл $HOME/.erlang.cookie (**SFEWRG34AFDSGAFG35235**), със конзолна команда – във възел с установено име !(1>erlang:set_cookie(node(),'**SFEWRG34AFDSGAFG35235**') и с проверка на текущия ключ (1>erlang:get_cookie(). %nocookie)

Контролна и отдалечена конзола. За преход към контролен режим на конзолата ^G(и help)
User switch command
→h
c [nn] – connect to job
i [nn] – interrupt job
k [nn] – kill job
j - list all jobs
s [shell] – start local shell
r [node [shell]] – start remote shell
q – quit erlang
? | h – this message
→
За стартиране и прекратяване на отдалечена конзола се използва
→ r 'node@machide2.example.com'
→ j
1 {shell, start,[]}
2*{node2@mashine2.example.com,shell,start,[]}
→ k 2 % прекратяване на отдалечената конзола
→ j
→ 1 {shell,start,[]} %проверяваме че не е активна.

Мрежа топология, скрити възли. Обменът е 1:1. за група от повече процеси се изгражда логическа топология. Например: пръстен, всеки- към – всеки (изисква n(n-1)/2 tcp/ip канала т.е. 6 за 4 възела обаче 28 на 8 възела). За намаляване на комуникационния сврът товар се използват скрити възли, т.е. възли които не се включват в комуникатиращатат група по подразбиране. За да се достъпят скритите възли се използват експлицитни директиви

RPC е класическия модел за процедурно-ориентирано разпределено програмиране (който е подобен на RMI в Java): процесът стартира отдалеченото изпълнение и на дадена процедура с локални параметри. В ерланг RPC се емулира и то по-ефективно отколкото в останалите платформи за разпределена обработка (и от MP):
remote_call(Message, Node) → {factorialServer, Node} ! {self(), Message},
receive {ok,Result} → result
after 1000 → {error, timeout} % срок 1 сек.
end.
В изпълняващия функцията factorial сървър factorialServer може да се приложи програмния шаблон:
server() → register(factorialServer,self()),
factorialLoop().
factorialLoop() →
receive
{Pid,N}→ Pid ! {ok, factorial(N)}
End,
factorialLoop().
Наличен е и модул вградени функции, поддържащи RPC: rpc:call(Node, Module, Function, Arguments). Изпълнява Function в Node при положение че Module е деклариран в изпълнимия път на Node и викащия възел има еднакъв ключ (cookie) с Node. Резултатът е върнатата стойност на Function или {badrpc,Reason}
Общата функция може да модифицира обръщението да бъде: синхронно,асинхронно, блокиращо, косвено, еднопосочно или групово (със всяка от предишките характеристики).

Синхронизация. RPC е средство за програмиране на клиент-сървър архитектури. Разпределените архитектури изискват внимателно обмислена система за синхронизация между процесите. В ерланг възможностите за сингронизация са: събитие/обмен със срок, свързани процеси или наблюдение на възел. *Синхронизацият със срок*: в предния пример чрез зададения срочен МР се избягва блокиране на клиента при блокиране на сървера или на комуникациите. В такъв случай обаче трябва да се вземат мерки за избягване на погрешна клиентска интерпретация при закъснял сървеен отговор. Например, чрез поредност на заявките или чрез изчистване на буфера за входящи съобщения преди следващотот обръщение към този сървер – за премахване на закъснели игнорирани съобщения от предходни заявки.

Свързани процеси. При свързаните процеси прекратяването на наследника предизвиква прекратяване на родителя. Използва се spawn_link/4 вместо spown/4:
setup() → % в родителя/клиента
process_flag(trap_exit,true),
spown_link('srv_node@fmi',myrpc,server,[]).
При свързаните процеси родителят се прекратява след получаването на EXIT сигнал ако наследникът се прекрати или EXIT сигнал с основание noconnection ако конзолата установи разпадане на комуникационния канал между 2-та процеса

Водещ процес и драйвер на обмена. За група процеси, например родител и наследници, е дефиниран входящ процес, който изпълнява стандартния IO за всички процеси от групата независимо от кой възел резидират. Лидерството се наследява при създаване на нови наследници или RPC:
Group_leader() % връща Пид на текущия leader
Group_leader(LeaderPid,Pid) % включва Pid в групата на LeaderPid.
Родителския процес epmd поддържа карта с виртуалните портове на всички ерланг-процеси за дадена машина без оглед на дефинираните възли. Самия epmd слуша порт 4369, като разпределя получените съобщения по локалните процеси.

Наблюдение на възли. Наблюдение на активността на сървеен възел се базира на BIF-a monitor_node(Node,Flag) → true. Ако Flag е true се стартира наблюдението на възела Node, а ако е false – то се прекратява. При активен Node се връща колекцията {nodedown, Node}:
Remode_call(Message, Node) → monitor_node(Node,true),
{factorialServer, Node}! {self(),Message},
receive
{ok,Res}→ monitor_node(Node,false), Res;
{nodedown,Node}→{error,node_down}
End.

Обмен със съдединения. Обменът в Интернет се базира на суперслоя за общи комуникации [TCP|UDP]/IP, който изпълнява обмена през sockets (съединения на процеси) на последователност от наредени или ненаредени съобщения от порт до порт (не възел или машина като при IP) със или без потвърждение/препредаване на грешните съобщения. Транспортните съединения разширяват IP-адреса с локален интерфейсен порт (4-цифрен),който се асоциира с един локален процес при откриването на съединението. Обменът със съединения не гарантира пренос през защитни стени. Използват се модулите get_udp и get_tcp. Откриване на съединение с опции:
gen_[udp|tcp]:open(Port)
gen_[udp|tcp]:open(Port, OptionList)

Опции на съединения:
list – предава съобщенията като спъсък от цели числа
binary – предава съобщенията като списък от доични.
{header,size} – отделя първите Size байта от binary-съобщение в списъка header, който да се интерпретира като заглавна/протоколна част от съобщението.
{id, ip_address()} – за машини с няколко IP-адреса(това са маршрутизатори) определя на кой IP-адрес да се отвори съединението
Inet6 – открива съединение с IPv6-адресация
{active, false} – съединението се открива в пасивен режим, при който може да се получават съобщения само с BIF-овете (gen_[udp|tcp]:recv(Socket,Length[,Timeout]) # срокът е в милисекунди)
{active,once} - след първото изпратено съобщение съединението минава в пасивен режим. Той е подходящ за конкурентен сержерен/реактивен процес, който приема заявки под формата на съобщения и стартира нова нишка за обслужването на прекалено много сержерни нишки при наплив от клиентски заявки. Родителският серверен процес е известен като listener, а обслужващите сътнетните клиенти са acceptor-и(например Tomcat+сервлети). След първата заявка обменът е между клиента и сервлета. Tcp се прилага при многокпратен обмен на дълга последователност от наредени съобщения между 2 процеса – например клиент(инициатор) и сервер(преактивен процес). Атрибут на съобщенията е поредният им номер.

UDP съединения. Използва се модула gen_udp. Откриване на UDP съединения в локалните възел 1 и възел 2 на портове 1234(1) и 1235(2) съответно: възел 2 изпраща съобщението(3) в двочен формат и съобщението(4) на порт 1234 в локалната им машина. Възел 1 прочита своя входящ буфер на отворения си порт(5). Закриват съединенията (6)
{ok,socket}= gen_udp:open(1234,[binary,{header,2}])
{ok,socket}=gen_udp:open(1235) {ok,#port<0.203>}
gen_udp:send(socket,{127,0,0,1},1234,[0,10|<<"Hi">>])
gen_udp:send(Socket,{127.0.0.1},1234,"Zdr")
flush. Shell got
{udp,#Port<0.576>,{127,0,0,1},1235,[0,10]<<"hi">>}
Shell got {udp,#Port<0.576>,{127,0,0,1},1235,"Zdr"}
gen_udp:close(Socket).

В TCP-съединенията данните са дълъг поток от байтове, които се структурират като последователност то номерирани кадри. А в ерланг TCP-съобщението (заявката или отговора) се предхожда от N=1-,2- или 4- байтове дължина на самото съобщение в байтове. Конкретната дължина се указва с аргумента {packet,N} във функциите gen_tcp:connect и gen_tcp:listen. С еднакви дължини(N) трябва да работят и клиентското и сървърното приложение. Подреждането на кадрите и фрагментацията се извършват прозрачно от платформата. Често съобщението от клиента към сървъра съдържа сериализиран(marshalling) бинарен ерланг-израз или друга сериализирана структура данни. Най-простата възможност за сериализацая в клиента и десериализация в сервера са функциите: term_to_binary и binary_to_term.

Конкурентният TCP-сървер е в състояние да обслужва поток от асинхронни (т.е. от множество клиенти) заявки като: всеки клиент може да изпраща последователност от заявки. За целта конкурентният сървър стартира нова нишка за индивидуално съединение с всеки нов клиент с gen_tcp:listen()+gen_tcp:accept().

Активно приемане на съобщения е неблокиращ обмен. Сървърът приема всички заявки. Изключение се получава при наводняване. Отсъства възможност за контрол върху процеса на приемане. **Пасивното обменяне на съобщения** е блокиращ обмен. Сървеът приема по една заявка само когато изпълни gen_tcp:recv в своя цикъл.Междувременно клиентът е блокиран. **Хибритното обменяне на съобщения** е обмен при който сървърът приема първата заявка неблокиращо (активно) а следващите им заявки са в пасивен обмен. Допуска се слушането на множество съединения от сървера. Практически е невъзможно да се стигне до наводняване.

10. Паралелно програмиране с ERLANG (Приложения за симетричен мултипроцесоринг)

Мултипроцесорно програмиране: За линейността (скалируемостта) и ефективността се изискава: 1) декомпозиция на приложението на подходящо множество процеси n>=p (по възможност n>>p); 2) избягване на страничните ефекти от конкурентния достъп до данните и от синхронизацията: взаимна блокировка (deadlocks), състезателен достъп (race condition) и т.н.; 3)в ERLANG страничните ефекти са почти елиминирани поради отсъствието на общи променливи и многонишково присвояване; 4) избягване на тесните места от последователната част на програмата; 5) при МП има къси съобщения и по-рядко обмен м/у процесите.

ERLANG и мултипроцесорите: Чертите на ERLANG, които подкрепят мултипроцесоринга са: конкурентност без критични области, асинхронен МП, прозрачно разпределение по ашини, аядра и виртуални възли, отказоустойчивост (чрез репликиране) и без mutex-и и транзактивна памет (освен ets/dets). ВМ се стартира в многоядрен режим със следните флагове: c:>erl –smp +S N, където –smp стартира ВМ в многоядрен режим – Symmetric Multiprocessing; +S N стартира ERLANG с N ВМ – всяка ВМ е нишка със самостоятелно планиране – но нишките имат общ процесен контекст. Нормално е N=>p, където р е брой ядра. N има стойност по подразбиране. Освен паралелни ВМ е желателно да се паралелизират и отделните приложения – по управлени, по данни и т.н.

Паралелизъм с криптични области: Общите данни м/у групата нишки на паралелното приложение намалява ефективността заради поддържането на консистентността м/у кешовете на ядрата. Подари това е трудно да се постигне ефективно работеща многонишкова паралелна програма на повече от 2-4 UMA-ядра. Всеки процес има собствен защитен контекст – за обмен се ползват променливите-ключалки – с произтичащите от това забавяния и странични ефекти. Процесите осигуряват истинска защита на паметта, на цената на значителен системен сврътовар за планиращото ОС-ядро, което управлява достъпа до паметта и движението на процесорния контекст по слоевете на виртуалната памет. Затова обработката в ERLANG-машината се нарича „процес" – като самите ERLANG -„процеси" не са дори нишки от гледна точка на ОС, но ERLANG -машината ги управлява като комуникиращи последователни процеси, а ключалките са заменени от „съобщения" – за лесен пренос м/у мултикомпютри и мултипроцесори – или дори за съчетаване на двете архитектури.

Паралелизъм без критични области: Функциите на ERLANGнямат общи данни – освен предадените чрез аргументите на обръщението – т.е. подобно на RPC-MP. Именно това позволява миграцията на всеки „процес" на ново ядро или процесор или даже на нов възел – т.е. с отдело адресно пространство. Този модел на комуникации м/у процесите обаче е ефективен когато свързваните процеси нямат „много" голям общ контекст. Миграцията на голям общ контекст поражда неефективност от комуникационния сврътовар.

Паралелизиране с хеш: Прилага се за паралелен (p2p) достъп до общ ресурс в хеш-таблици –distributed hash tables. Има разпределение на данните (като е в разпределителна система). Ключа определя в кой „възел" ще се съхраняват асоциираните с него данни. Обичайно възлите се сгранизират в логически пръстен по свои n-разредни идентификатори и всеки нов компонент се асоциира с този възел, който е с най-близко ID до неговия ключ. Постига се квази-балансирано разпределяне по възли на контекста – по памет и по време за достъп за запис и писане.

Общи данни: ETS/DETS: [Disk] Erlang Term Storage са вградени системни модули за ускорен достъп до огромни по размер записи в/у диск или направо в ОП. Тези данни представляват подходящо структурирана за бърз достъп таблици с двойките ключ-стойност. Достъпът се влияе много слабо от размера на таблиците. Тези данни не са специализиран SV механизъм, но са много полезна алтернатива на МР при паралелните процеси, понеже позволяват ефективен конкурентен достъп при значителен размер общ контекст. Частично достъпът се ускорява поради това, че не е сериен както при „съобщенията", макар че „съобщенията" м/у 2 паралелни процеса се разполагат в общата памет на възела, достъпът не ползва друг адрес освен началото на съобщението. При ETS данните съществуват в 1 копие, а при съобщенията се дублират – в областите на изпращащия и получаващия процес – което може да се разполагат в общата памет на възела. Данните позволяват множествен достъп, което е важно предимство при локално- и глобално-синхронните алгоритми, които изискват групови комуникации и то по възможност с паралелна имплементация.

D/ETS операции: 1)създаване на ETS: ets:new; 2)отваряне на DETS: dets: open_file; 3) включване на двойка или списък двойки ключ-стойност: insert(Tablename, X); 4) извличане на стойност по ключ: lookup(Tablename, Key) – може да се върнат в списък повече от 1 стойност за някой типове таблици, при неоткрит ключ резултатът е празен списък; 5)прекратяване на ETS: ets: delete(TableId)-прекратяването е изтриване на данните с освобождаване на паметта, ERLANG не прави изчистване на неадресираните таблици, но ets се изтрива когато терминира създалия процес, а dets се затваря; 6)затваряне на DETS: dets:close(TableId).

Типове D/ETS: Има няколко типа таблици, те са: 1) set и ordered_set: точно 1 обект на ключ без или със подреждане на обектите (напр.:{a,1} {b,2} {c,3}). Подреждането на обектите води до различна скорост на операциите. 2) bag и duplicate_bag: повече от 1 обект на ключ – с едно или повече копия на всеки от обектите (напр.: {a,1} {a,2} {b,2} и {a,1} {a,1} {a,2} {b,3}). Съществува максимален брой таблици, поддържан от 1 ERLANG-възел.

Паралелизиране на списък: 1) последователно-рекурсивна (поелементна) обработка на списък с lists:map: map(_,[])->[]; map(F,[H|T])->[F(H|map(F,T)]. 2)паралелна версия на map-pmap: pmap(F,L)->S = self();
Ref = erlang: make_ref(), %make_ref връща нов адрес;
Pids=map(fun(I)->spawn(fun()->do_f(S,Ref,F,I) end)end,L),gather(Pids, Ref);
do_f(Parent, Ref, F, I) -> Parent !{self(), Ref,(catch F(H))};
gather([Pid|T], Ref)->receive {Pid, Ref, Ret ->[Ret\gather(T,Ref)] end; gather([],_)->[].
3) паралелната pmap: обработка като map, но за всеки елемент на списъка създава нов процес с do_f и го маркира с Ref; 4) обработващите процеси могат да завършат в произволен ред, затова при събирането на резултатите gather избирателно ги сортира по Ref; 6) (catch F(H)) се използва вместо директно F(H) за да се прихванат изключенията в процесите-наследници, които да терминират родителя pmap.

Гранулираност: Паралелизирането на списъци може да подведе, че за ефективна паралелна програма е достатъчно да подменим map с pmap. За ефективност допълнително трябва да се подбере подходящата гранулираност. При pmap паралелизма P=l е максимален (l = length(L) е мощността на списъка), съответно гранулираността е най-фина – това е другият край на скалата спрямо последователната map. Трябва да се анализират междинните варианти в зависимост от времевата сложност T_F на F(L). Ефективното приложение на pmap може да има само при $T_F >> T_{SPAWN}$

11. Операционни системи – управление на процеси и нишки, миграция на код. Имена, идентификатори и адреси; йерархия и откриване. Премахване на неадресирани компоненти.

11.1. Управление на процеси и нишки 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 – например операциите са: 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 (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. 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.

Паралелизъм на ниво израз: Това ниво е свързано с езикови спецификации (примитиви за паралелно изпълнение на инструкции). Напр. примитивът Parbegin/Parend задава блок от изрази за паралелно изпълнение, по време на което главният процес блокира. В UNIX паралелизмът на израз се осъществява 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

Паралелно програмиране в UNIX : UNIX е най-разпространената ОС за паралелни системи. В нея процесите се управляват чрез системни заявки (calls). За планиране и контрол се използват системния таймер – функциите timer-init() и timer-get() или семафори. Междупроцесния обмен се осъществява чрез алокиране на общи променливи чрез 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.

Многопроцесно приложение в UNIX: За вътрешна идентификация на процесите често се прилага и индексиране. Например функция Mkps() за създаване на *n* процеса-наследници със стойност на ppid 0 в процеса-родител и от 1 до *n* в наследниците.

Паралелно програмиране в UNIX:

```
ppid = Mkps(procs); /* creation of argv[1] number of processes*/
switch (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. Всяка вече декларирана променлива може да бъде обявена за общ достъп (и алокирана в общ сегмент от паметта) със системната заявка Share(). Резултатът е, че освен деклариращия процес, всички негови наследници (създадени след нейната декларация) имат достъп до съответната променлива.

Времево планиране на процесите: Времевото планиране е частен случай на планирането по събитие, при който събитието е изтичане на таймер. За целта се ползва се системния часовник с импулси на всяка микросекунда. Прави се заявка към системата timer-init(),която стартира (нулира) локален брояч за процеса, а заявката timer-get() връща текущата му стойност в микросекунди.

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 lockname_create и lockname_init, където lockname е множество от следните типове: LOCK, BARRIER, SEMAPHORE и EVENT. Следва кратко описание на различните типове данни: 1)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. 2)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, а flag задава режима на изчакване. flag = SPIN_BLOCK блокира с циклично изчакване, а flag = PROCESS_BLOCK блокира при достъп до данни. 3)EVENT е тип данни, съставен от двойката (event, flag), където event задава събитие, което трябва да се изпълни преди процеса да продължи (възможно еповече от един процес да чака това събитие),а flag задава режима на изчакване като при BARRIER. 4)SEMAPHORE е тип данни, асоцииран с двойката атрибути (count, flag), където count задава броя процеси, които имат право на достъп до променливата преди заключването й,а flag задава режима на изчакване като при BARRIER. С този тип е асоциирана операцията semaphore_set за count.

Особености на 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 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 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 наричани нишки (threads), са средство за постигане на по-фина грануларност респективно по-оптимално планиране. При нишките има по-слаба конкурентност. Нишковия контекст се състои примерно от CPU-контекста и текущ статус, така че защитата на нишковите данни в рамките на процеса зависи от кодирането на многонишковото приложение. Това води до по-сложно програмиране. Многонишково програмиране се прилага и при унипроцесорни приложения. Многонишковата програма за унипроцесор е преносима и за паралелна обарботка. Многонишковите програми са по-удобни за настройка.

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). При нишките има по-нисък системен свръхтовар, т.к няма операции върху паметта - при създаване/закриване само се заделя и освобождава стека и при превключване - само се замянят стойностите в ЦПУ регистрите. Недостатък е,че 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, така се преодолява тоталното блокиране, но свръхтовара е съпоставим с процесния.

Леки процеси:LWP (lightweight process) са хибриден подход. Леките процеси се изпълняват като обикновени процеси. Един процес може да включва няколко LWP. При 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 имплицитно предават изпълнението им на LWP – фиг. 10.1 (10.16). 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 потребителски 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 асоциираната 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, т.е. програмистите работят с нишки, не с LWP. Има преносимост за унипроцесорно и паралелно програмиране(във втория случай леките процеси на едно приложение се изпълняват на различни процесори)

Многонишкови клиентски процеси:Те обикновено постигат маскиране на комуникационните и
синхронизационни закъснения на някои нишки чрез изпълнение на други. Например 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).

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. Например 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.

11.2. Миграция на код

Миграция на код: Миграцията на код се среща под формата на:1) миграция на процеси – например за балансиране на локалния изчислителен товар; 2) мигриране на програми за отдалечено изпълнение (при сървера) – напр. зареждане в сървера на програма за локална обработка на данни и връщане само на резултата; 3)миграция при клиента – напр. зареждане в клиента на програма за попълване параметрите на заявка и връщането й 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 има процес свързване (binding) т.е. настройка на адресните аргумент (данни). Това може да стане по следните варианти: 1) свързване по идентификатор – напр. при мигриране на данни, които са адреси на файлове с URL идентификация (понеже идентификатора е униресален); 2) свързване по стойност – напр. адресиране на стандартна библиотека в С и Java (действителния им идентификатор е локален); 3)свързване по тип – напр. адресиране на локални устройства (принтери, монитори)

Модели за миграция на код: Видовете модели са:1) ниска (weak) мобилност – само на сегментите код и данни. Изпълнението стартира отначало. Например Java аплетите. Има изисквания за преместваемост на кода.2)висока (strong) мобилност – ниска мобилност + сегмента на статуса. 3) по инициатива на изпращащия процес. Например изпращане на програма за изпълнение от изчислителен сървер (изпращащия процес е клиент. В този случай е необходима идентификация на клиента) или изпращане на процес за балансиране на товара при групово обслужване (изпращащия процес е сървер). 4)по инициатива на приемащия процес. Например Java аплети (прием. п-с е клиент) или отново за балансиране но при инициатива на приемащ сървер

Миграция на код в хетерогенна среда : В хетерогенна среда:1) при ниска мобилност (само на код и данни) е необходима прекомпилация на програмата за различни машини/ОС; 2) при висока мобилност (код, данни и статус) се осъществява с поддържане на машиннонезависим миграционен стек в определени точки на програмата, (в които и само в които може да се извърши миграцията) . 3)в процедурните езици (С) това е след изпълнението на текуща функция/метод и преди стартирането на следващ/а, за да не се налага пренос на стойностите на процесорните регистри, които са машиннозависими. 4)при интерпретирани езици. Например при скриптовите езици виртуалната машина директно интерпретира програмния код (Tcl) или универсален междинен код, генериран от компилатор (Java)

11.3. Имена, адреси и идентификатори;йерархия и откриване

Имената са символни низове за идентификация на компоненти – ресурси (възли, устройства вкл. вторични памети, файлове) и обслужвани компоненти (процеси, потребители, съобщения, документи, мрежови съединения и др.). Именуваните компоненти подлежат на управление или промяна посредством съответни точки за достъп – адреси. В РС са широко застъпени **динамичните адреси,** а 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. Например 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, т.е.: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 чрез граф – т.н. граф на имената. Има два вида пътища на имената в графа – абсолютни (от корена) и относителни. Графът на имената обикновено е дърво (само с едно входящо ребро за всеки възел – връх или листо) или е ацикличен. Във върховете са разположени имената на компонентите и обикновено дървото има само един корен.
Решаване на имената (name resolution) е извличането на идентификатор на компонента при зададено име (и път). Псевдоним (alias) е допълнително име на компонент. Това е възможно, когато графът на имената допуска повече от един път до компонента – например в UNIX или 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. Нивата са:1) 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); 2)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 (примери в DNS sun.com, uni-sofia.bg, fmi.uni-sifia.bg, acm.org); 3)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. Характеризира се с 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. Тя е 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. Абсолютният път се отчита от корена и се означава с “.”, която може да се пропусне. Област (domain) е поддърво в DNS, а 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. Има 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.

Итеративно решаване на адресите: При итеративното решаване на адресите пълното име (с път) се предава на сървера на имената в корена (пример ftp://is.fmi.uni-sofia.bg/t3/rITa1.pdf). 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. 10.2 (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. Стъпката 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. 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.

11.4. Премахване на неадресирани компоненти

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 основна роля играе 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 и могат да маскират тази системна функция от клиентския и сърверния процес. Съществува граф на указателите с множество на корените, които не се премахват дори и когато няма указатели към тях – напр. потребители, системни услуги. 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: Статуса на обекта(компонента) се асоциира с брояч на указателите. Брояча може да се инкрементира и декрементира. Обект с нулев брояч подлежи на премахване. Броячът на указателите се поддържа обикновено от скелетон-стъба на обектния сървер. При РС този подход (приложен без модификации) поражда проблеми поради комуникационни закъснения и загуби. 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 асоосиира 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.
Допълнително предимство на идемпотентността е, че заявките могат да се изпращат няколкократно 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 е ако 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 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). При унипроцесорите проследяването се прави по метода mark-and-sweep: 1)фаза на маркирането на достижимите от корените компоненти; 2)фаза на изчистването, при която системата открива в паметта компоненти, нефигуриращи в маркирания списък, които се изтриват. Един вариант е компонентите с открит указател към тях, но преди да е извършено проследяване на техните указатели, се маркират междинно като “сиви” (традиционно “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 процес Пi стартира собствен колектор, който оцветява прокси- и скелетон-стъбовете, както и самите обекти с Б, Ч и С в следните стъпки:
1)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.
2)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) 3)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 4)прокси-стъбовете, получили това обратно съобщение се оцветяват Ч 5)колекторите продължават рекурсивно до завършване на оцветяването т.е. до оцветяване с Б и Ч (накрая няма С-компонети няма)
6)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 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.

12. Синхронизация и системно време

Синхронизация на системното време е необходима при комуникация на процесите, при подреждане на разпределени събития(право на достъп, бюлетин, транзакции) и при използване на системното време като аргумент. В РС програмните компоненти може да са разположени на компютри с разлика в системните времена. Тази десинхронизация може да бъде причинена от разлика в тактовата честота на осцилаторите и при различни системно настройки. **Системното време** се отчита от таймер, който използва кристален осцилатор. Асоциирани с всеки кристален осцилатор са регистър брояч и регистър за броя импулси на осцилатора за определен интервал. Всяка осцилация на кристала намалява стоинostта на регистъра брояч с едно, когато стоинostта стане 0 се генерира прекъсване и стоинostта на регистъра брояч се приравява на стойността на регистъра за броя импулси. Всяко такова прекъсване се нарича clock tick. При всеки clock tick софтуерното време(системният часовник) се увеличава с едно. Нека стоинostта на този системен часовник означим с *C.* За глобална координация се използва UTC(Universal Coordinated Time), което се разпространява чрез късовълнови радиостанции от националните институти по стандартизация и геостационарни сателити. Нека UTC времето е *t*, тогава стоинostта на системния часовник на машина *p* е $C_p(t)$. В идеалния случай $C_p(t)= t$, т.е. $dC/dt=1$ за всяко *p* и *t.* Реалните осцилатори в масовите компютри работят с относителна грешка $\rho \approx 10^{-5}$ т.е. $1-\rho \le dC/dt \le 1+\rho$, ρ е максимално отклонение с възможно избързване или изоставане . За да се гарантира, че два часовника се отклоняват с повече от δ, то те трябва да се ресинхронизират поне на всеки δ/2ρ секунди.
Ще разгедаме следните алгоритми за **синхронизиране на системното време**:
1.времеви сървър, синхронизиран по UTC или усредняваш системното време на възлите; 2.разпределени системи за ресинхронизация от тип p2p. Разглеждаме **алгоритъма на Christian**, който използва пасивен сървър, синхронизиран с UTC. През период не повече от δ/2ρ секунди възлите пращат заявка за системното време на сървъра. синхронизиран с UTC. При този подход има проблем в закъснението, което се генерира от цикъла заявка-обслужване-отговор. Този проблем се решава като към полученото време от сървъра се добавя половината от времето за отговор измерено от машината, пращаща заявка към сървъра. Друг проблем е, че е възможно времето, подавано то сървъра да е по-малко от времето на машината, пращаща заявката, т.е. машината да трябва да върне своето време назад. **Не се** позволява коригиране на локалното време назад. Тези промени се внасят постепенно при всяка следваща заявка. **Алгоритъмът на Berkley** използва активен сървър(демон), който предиодично преверява локалното системно време на машините и изравнява времето им към усредненото системно време. Този алгоритъм се ползва в мрежи, при които няма връзка до UTC предавател. Известни са и **p2p(децентрализирани) алгоритми за синхронизация на системното време**. p2p синхронизацията се базира на периодично изпращане на локалното системно време от всеки възел до всички останали. Времето, през което възлите пращат локалното си системно време се означава с *R*(период на ресинхронизация). Всеки възел изчаква време *S* получаването на съобщения от други възли. След като се получат всички съобщение се изчислява новото системно време, като се усредняват всички стойности като евентуално се игнорират *m* на брой екстремно стойността. Всеки възел изпраща съобщение със своето локално системно време в началото на периода за ресинхронизация(за начален момент на периоадите се избира синхронизирано време T_0). Съществува и протокол за мрежово време (Network Time Protocol, NTP; Mills,1992), който осигурява синхронизация в Интернет с точност до 50 мсек.

Синхронизация за подреждане на събития се прилага, когато не е важно съответствието между машинното и физическото време и когато не е необхосима синхронизация на машинното време между възлите, а само еднакво подреждане на отдалечени събития. Тази синхронизация се базира на релацията за предходност (happens-before relation) между събитията *a* и *b*: $a \to b$ (*a* предхожда *b*). Тази релация е транзитивна. Две отдалечени събития са в релация на конкурентност, когато не може да се определи реда, в който са се случили. С *C(a)* означаваме логическото време на събитието *a*, тогава $a \to b \Leftrightarrow C(a) < C(b)$ е изпълнено. Времето *C* се движи единствено напред, не са позволени корекции, които връщат времето назад. Ще разгледаме **алгоритъма на Lompart за синхронизация с времеви марки(timestamps).** Същността на този алгоритъм е пращането на синхронизиращи съобщение, в който се намира локалното логическо време на процеса изпращач. Ако локалното логическо време на процеса получател е по-малко от времето в съобщението(времевата марка), то локалното време на получателя става равно на времето от съощението + 1 При събития с еднакво време *C* тези събития се дистанцират логически с 1 такт(единицата за системно време). Допълнителна прецизност на логическото време(за уникални времеви марки) се постига като към логическото време се добави идентификатора на процеса(или негова произодна) като дробна част.

Протоколът за тотално подреждане прилага логическа синхронизация с времеви марки за еднакво подреждане на събитията (получаване на съобщения) при групово предаване(multicasting) - напр. при коригиране на записите в репликирана на база данни. При групово предаване на съобщения с времеви марки изпращащия процес, като член на групата получава свойте съобщения и то в реда на изпращането им и без загуби. Всеки приемащ процес записва получените съобщения в локален буфер по реда на времевите марки и потвърждава приемането до прицесите в групата. Тези потвърждения също се маркират. Същевременно се прилага алгоритъма на Лампорт за положителни корекции на локалното логическо време. Всички съобщения(дори потвържденията) са групови, независими дали са предназначени за всички процеси в групата. Локалните
буфери са опашки (FCFS) от които съобщенията се предават към съответните локални приложения, като се изтриват от буфера (както и техните потвърждения). Резултатът е, че всички локални буфери са с еднакво подреждане на съобщенията и потока от съобщения към всяко локално приложение е идентичен (N.B.: еднаквото подреждане обаче не гарантира запазване на реда на възникване на събитията в реално
астрономическо време ⇒ алгоритъма на Лампорт е приложим за събития, между които
няма причинно-следствена връзка – causality).

Протоколът за съхранено подреждане позволява тотално подреждане на събития при запазване на реда им в реално време – напр. при публикуване на дискусионни и новинарски бюлетини, където е важна не само идентична подредба, но и запазване на причинно-следствената връзка – т.е. съхранено подреждане (causally ordering). Прилага се векторна маркировка(vector timestamp). Всеки процес Pi поддържа свой вектор от броячи Vi, чиито елементи отразяват броя събития, настъпили в процесите със съответен индекс – Vi[j] = брой настъпили събития в Pj; Vi[i] = брой събития в Pi. За целта когато Pi изпраща съобщението m, към него добавя (т.нар. piggybacking) и текущата стойност на своят вектор Vi като векторна марка vt. По този начин получаващият съобщението m процес Pj е информиран за броя събития, възникнали във всички процеси преди Pi да изпрати m – т.е. общия брой събития, от които изпращането на m може (потенциално) да е следствие. При получаването на m Pj прави корекциите Vj[k] = max{Vj[k], vt[k]} и Vj[i]++, при което Pj вече разполага с броя събития-съобщения, които предхождат (евентуално като причина) m (и съответно – ако има такива – може да ги изчака).

Представяне на глобалния статус

Глобалния статус* се състои от *локалния статус на всеки процес и ***съобщенията в транзит*** (напуснали локалния изходен буфер на изпращащия процес, но недоставени в локалния входящ буфер на приемащия процес/и). Локалният статус на процесите е контекстно-зависим – при разпределена БД той може да включва само записите в БД без междинните резултати на обработка; при mark-and-sweep разчистване може да се състои само от маркировката на скелетоните, прокси и обектите от адресното пространство на съответния процес. Глобалния статус на РС се използва най-често за детекция на мъртва точка(deadlock) или край на разпределената обработка (и в двата случая изпълнението на всички локални процеси е преустановено и няма съобщения в транзит; интерпретацията е предмет на анализ). При РС е важна изискването за *свързаност* (*консистентност*) на глобалния статус – т. нар. ***заснемане на РС*** (***distributed snapshot***) – напр. ако процес *P* е получил съобщение от *Q*, заснемането трябва да съдържа и запис, че *Q* е изпратил това съобщение. Консистентността на заснемането се представя с разрез (cut).

Алгоритъм за заснемане на глобален статус

РС се разглежда като набор процеси, свързани с преки симплексни канали (еднопосочни, за разлика от дуплексните и полу-дуплексните) от тип точка-точка (напр. TCP връзки). Алгоритъмът се инициира от произволен процес *P* с регистриране на локалния си статус и изпращане на маркер-заявка за заснемане на глобалния статус по всичките си изходящи канали. Процесът *Q* получава заявката по свой входящ канал *C* след което: [заснемане на процес:] регистрира своя локален статус и размножава заявката по своите изходящи канали; *Q* е наследник, а изпращащия заявката процес е предшественик **ИЛИ** [заснемане на канал:] ако вече е получил заявката (по друг свой входящ канал) и е регистрирал локалния си статус, той регистрира статуса на канала *C* – т.е. съобщенията, които е получил по този канал в интервала от регистриране на локалния статус до получаване на последния маркер по канала *C. Краят на заснемането за Q е когато получи маркер по всеки свой входящ канал и изпълни* горната стъпка; заснетият от него локален статус се изпраща на *P* (възможни варианти с цел рекурсивно описание на процеса). *P* разполага с глобалния статус на системата когато получи локалните статуси на своите наследници (и рекурсивно – на техните наследници).Няколко заснемания могат да бъдат инициирани така че да протичат едновременно – за целта маркерите съдържат идентификатор на инициатора (който се използва и за изпращане на локалния статус).

Критичните зони за взаимно изключване

Алоритмите за взаимно изключване се изповат в конкурентното програмиране за да се предотврати едновременнен достъп до общ ресурс от части от компютърен код, които наричаме критични зони.
В унипроцесорите критичните зони за взаимно изключване на достъпа до споделени ресурси се управлява с механизмите на ключалки-семафори и монитори. В РС тези подходи се имплементират от централизирани алгоритми за управление на достъпа, но се прилагат също и разпределени и резервационни алгоритми.

Централизирано взаимно изключване:

Базира се на излъчен процес - координатор, към който се отправят заявките за достъп до критична зона. Заявките се потвърждават по реда на постъпването, като процесите с непотвърдени заявки изчакват. След освобождаване на критичната зона един от чакащие (блокирани) заявители получава потвърждение (и достъп). Положителните страни на този алгоритъм са :1)Ограничен служебен обмен – необходими са само три съобщения за всеки път, когато критичния район се използва – заявка, потвърждаване и освобождаване. 2)Поради това, че заявките се потвърждават по реда на постъпване никой процес не чака безкрайно.Отрицателна страна на този алгоритъм е, че не може да се различи изчакване от болкирал координатор защото и в двата случая не се връща съобщение.

Разпределено взаимно изключване (Ricart, Argawala – 1981)

Базира се на тотално подреждане на събитията, тоест за всеки две съобщения между процесите се знае кое е изпаратено първо. Това се реализира най-често като се пазят времеви марки . За реализиране на алгоритъма са необходими също така надеждни (потвърдени) групови комуникации. Заявителят изпраща съобщение с името на критичната зона, своя идентификационен номер и локалното време на всички останали процеси. Всеки получател извършва алтернативно следното:1)връща ОК съобщение ако не е или не чака достъп в тази критична зона; 2)ако е в критичната зона, не отговаря, а буферира локално заявката; 3)ако е изпратил собствена заявка за същата критична зона, сравнява двете времеви марки и ако има по-късна (по-голяма) марка, изпраща ОК на заявителя, в противен случай не отговаря, а буферира локално отдалечената заявка
Заявителят изчаква ОК от всички останли процеси и заема критичната зона. След напускане на критичната зона, процесът изпраща ОК на всички заявители от локалната си опашка за тази зона и ги изтрива от нея. Минусите на този алгоритъм са, че ако един процес болкира всички процеси, които искат да влязат, в която и да е критична зона остават в състояние на изчакване. Освен това са нужни 2*(n-1) съобщения всеки път, когато процес иска да използва критичен район.

Резервирано взаимно изключване Token Ring Algorithm

Базира се на логическо подреждане на процесите в пръстен. Когато пръстенът се инизиализира стартиращият процес получава съобщение- token. Служебното съобщение се предава последователно между процесите, давайки право на текущия процес, в който е token-а, достъп до
критичната зона. След излизане от нея, съобщението-token се предава към

следващия процес в пръстена. Получаването на token дава права на еднократен достъп в една от критичните зони.
Плюс е, че последователното преминаване през всички процеси осигурява, че никой няма да чака безкрайно. За сметка на това при загубен token възстановяването е трудно. То е контекстнозависимо, тъй като е базирано на интервали от време.
Сравнение между централизираните, разпределените и резервационните алгоритми за взаимно изключване:Първият е с най-малко съобщения за влизане в район, втория е с фиксиран, но голям брой, а при третия съобщението-token може да се предава безкрайно без някой да иска да влезе в критичен район, но може и при всяко предаване текущият процес да влиза. И при трите се наблюдава проблем, когато процес болкира. За тези случаи се реализира допълнителна логика.
Транзакции: Транзакциите са механизъм за синхронизация на съвместната работа на устройствата в системата (първоначално при унипроцесорите), на взаимодействащи процеси и др.. Те притежават свойството аомарност – тоест няколко действия се третират като едно. Функционират на ***принципа "всичко-или-нищо"***. Или се изпълняват докрай, или процесите се връщат в състоянието преди началото на изпълнение на транзакцията (примери: обслужване с банкомат, електронна търговия, он-лайн резервации) Синхронизацията с транзакции се базира на специални *примитиви*, които се поддържат от ОС или се интерпретират като езиково разширение – т.е. обръщения към системата, библиотечни процедури или езикови изрази (те са специализирани, но в тялото на транзакцията може да присъстват и изрази с общо предназанчение). Наборът транзакционни примитиви е контекстноориентиран, но за синхронизация на обслужването винаги включва : 1)begin_transaction – начало на транскацията;2)end_transaction – край на транзкцията, прави се опит да се запаметят промените;3)abort_transaction – прекратява се заявката и се връщат в старите стойности;4)евентулно read и write – четене и писане във файл или в друг обект
Свойства на транзакциите (ACID), блокови транзакции
Свойства на транзакциите (ACID):1) Атомарност (Atomic) – т.е. прозрачност – резултата от транзакцията е или като от еднократна моментална операция или операция, изобщо отсъства вс едно не е правен опит да се изпълни ("all-or-nothing") – напр. Транзактно добавяне на байтове към файл. Ако друг процес достъпи файла по време на трансакцията, той е в началния си вид (без междинни състояния) 2)Логичност (Consistent) – съхраняване на системните константи. Например при банковия трансфер трябва да се запази общата сума пари преди и след транзакцията, въпреки че по време на изпълнение на самата транзакция принципа може да се наруши. Другите процеси обаче нямат достъп до манипулираната информация, така че нарушението е прозрачно.3)изолираност (Isolated | serializable) – конкурентните (едновременни) транзакции се изпълняват като последователни действия съгласно определени принципи на подреждане4)устойчивост (Durable) – след изпълнението на транзакцията резултатите от нея не могат да се отменят. Тези свойства се обозначават с абривиатурата ACID- или flat- (т.е. блокови). Тоест транзакциите не допускат съхраняване и достъп до междинни резултати, което не винаги е желателно, напр. при резервацията на серия полети.
Вложени транзакции
Вложени (nested) транзакции – представляват йерархичен дървовиден набор от субтранзакции, първата от които инициира няколко от следващото ниво и т.н. – в съответствие с логическото и каузално (причинно-следствено) разделение на цялата супертранзакция". Всяка от субтранзакциите е логически независима от изпълнението на останалите (примера с последователните полети)
Целта е да се постигне ускорено изпълнение при паралелно изпълнение от няколко сървера или за да се упрости програмирането, но може да се ползват и за съхраняваняване на междинни резултати.
Наборът субтранзакции се счита за изпълнен, само ако главната субтранзакция е изпълнена, а ако не е – заличават се и резултатите на успешно изпълнените дъщерни субтранзакции (което може да породи проблем особено при изпълнение в РС). За тази цел на всяка субтранзакция се дава частно копие на всички обекти от системата,което тя променя, както е необходимо. Ако се отмени родителската субтранзакция копието изчезва, ако се приеме то заменя моментното състояние на системата, коетородителската транзакция притежава. Изпълнението на вложените транзакции е рекурсивно: когато главната субтранзакция е изпълнена, за изпълнени се считат и другите завършили субтранзакции по йерархията. Резултатите от неизпълнените субтранзакции се заличават.
Разпределени транзакции: При тях декомпозицията на супертранзакцията в субтранзакции не следва логическото разделение, а се определя от структурата на разпределения контекст. Например при разпределна база данни, върху всеки от дяловете на която оперира отделна субтранзакция или при междубанков трансфер със субтранзакции върху различни бази данни . Блокова транзакция (като контраст с разпределената) е начисляване на лихва по сметка(това е само в една база данни).
Имплементация на транзакциите: С резервирано работно пространство или с дневник (log-файл)
Резервираното работно пространство изисква при стартирането на транзакцията целият контекст, заедно с входно-изходните файлове да се разположат в резервирано (private) работно пространство. Операциите (промените от работното пространство) не се регистрират във файловата система до приключването на транзакцията. За оптимизиране, в работното пространство се копират само съответните блокове от файловете, отварями за писане – както и системния индекс на съответния файл. Обработката се извършва върху копието на блоковете и индекса и чак след приключване на транзакцията, индекса и блоковете се коригират и във файловата система. При метода с log-файл всяка промяна от записите на транзакцията се извършва направо върху блоковете на файловата система, но предварително се регистрира в log-файл с индекс на блока, старо и ново съдържание (writeahead log). В случай че транзакцията бъде отменена, регистрационният (log-) файл се използва за възстановяване в обратен ред на записите (LIFO) – този процес се нарича "rollback" Тези методи са приложими и за разпределените транзакции, тъй като субтранзакциите оперират локално върху даден контекст.
Конкурентно изпълнение на транзакциите: Конкурентното (едновременно) изпълнение на няколко транзакции изисква контрол на достъпа до техния контекст – например файлове и БД-записи – така че резултата да е консистентен т.е. такъв като при последователното им изпълнение.
За целта управлението на транзакциите се разслоява йерархично на 3 нива: 1) мениджър транзакции МТ – транслира примитивите на отделните транзакции в заявки за следващото диспеческо ниво (напр. с идентификация на транзакцията и [отдалечен] адрес на данните + управляваща информация); 2) Диспечер Д – планира реда и момента за извършване на отделните операции от различните транзакции съгласно планиращ алгоритъм (по методите с ключалки и времеви марки); 3) мениджър данни МД – изпълнява четене и запис в устойчивите структури данни, което е прозрачно за планирането на транзакциите. Конкурентно изпълнение в РС:1) Във всеки възел се стартира двойка от процесите диспечер и МД, а за всяка транзакция – отделен МТ; 2)МТ изпраща генерираните заявки към съответния диспечер; 3)Диспечерът може да изпрати планираните от него заявки и към отдалечени МД.
Серийно планиране на конкурентни транзакции: Серийното планиране запазва резултата от конкурентните транзакции такъв, какъвто би бил при последователното им изпълнение. Коректното планиране разрешава конфликтните операции. Конфликтни операции са тези, които две (или повече) конкуретни транзакции извършват върху общи данни и поне една от тези операции е запис: четене-запис конфликт; запис-запис конфликт. Конфликтът се разрешава чрез заключване на данните или чрез подреждане с времеви марки като се прилагат се два планиращи подхода:1) ***песимистичен подход***: операциите се синхронизират ***преди*** изпълнението им т.е. проверяват се за конфликт и ако да – се подреждат преди да бъдат изпълнени. 2)***оптимистичен подход***: операциите се синхронизират ***след*** изпълнението им. Изпълняват се целите транзакции и ако накрая се установи че е имало конфликтни операции, поне една от транзакциите се отменя (абортира).
Песимистично планиране с двуфазно заключване: Тъй като транзакциите са конкурентни, заявките за заключване подлежат на потвърждение (от диспечера в зависимост от изискванията на безконфликтното серийно планиране)При двуфазното заключване (two-phase locking, 2PL) заключването се разделя на две фази: 1)нарастване (growing phase): процесите на транзакциите заявяват заключване на съответните данни (чрез заявка от съответен мениджър транзакции до диспечер);. Заключване е необходимо и при четене за да не се проемнятдните от друга транзакция.2)свиване (shrinking phase): процесите на транзакциите заявяват отключване на съответните данни чрез заявка от съответен мениджър данни до диспечер. Важат следните правила за диспечеризация на конкурентните заявки:1) При заявка за операция, диспечерът проверява конфликтността с вече потвърдените заявки и потвърждава заключването или отлага заявката както и изпълнението на заявяващата транзакция (песимистично планиране);2)Диспечера освобождава заключване само след като получи потвърждение от мениджър данни, че операцията е завършила; 3)след освобождаване на заключване по заявка на даден мениджър транзакции (и респективно транзакция), диспечерът не допуска нова заявка от същата транзакция – независимо дали е за същия или друг обект. Нови заключвания се допускат преди да е освободено първото от тях, противното е програмна грешка, която отменя самата транзакция.
Варианти на 2PL
Строго (strict) 2PL, при което всички заключвания на транзакцията се освобождават след приключване на последното от тях (дори и когато съответната транзакция завършва с отмяна). Така се избягва възможността от каскадни отмени на транзакции, която възниква, ако са били обработени резултати от отменени впоследствие транзакции (достъпни са само резултати на вече изпълнени транзакции)Блокировка в мъртва точка (deadlock) при strict 2PL настъпва ако две транзакции заявят едновременно две заключвания но в обратен ред. За избягване на мъртва точка се прилага:1) служебно подреждане на заявките; 2)времеинтервал за откриване на мъртва точка - когато заключването продължи в рамките на този интервал е настъпил deadlock. 3)граф на процесите и заключванията за откриване на цикли. Видове 2PL:1) ***централизирано*** 2PL, при което заявките се обработват от централизиран диспечер, а достъпът на мениджър транзакции до мениджър данни е разпределен. Вариант на централизираното 2PL е когато няколко диспечера си разпределят контрола за достъп до данните (primary 2PL);2)***разпределено*** 2PL: всеки диспечер планира достъпа само до локалните данни, но ако данните са репликирани, съответния диспечер размножава заявката до възлите с реплики.
Песимистично планиране с времеви марки:При този метод се маркират както заявките, така и данните. Заявките се макират с времева марка *s* за началото на съответната транзакция *T,* като се прилага алгоритъма на Лампорт за уникалност на марките т. е. *s*(*T*). Обектите данни *x* се маркират с марки за четене и запис – съотв. *sw*(*x*) и *sr*(*x*) – съответстващи на транзакционните марки *s*(*Tm*) и *s*(*Tn*) на процесите, които последни са извършили съответните операции.
При конфликт на две заявки се потвърждава тази с по-малка марка (по-ранно стартиране):
При заявка read(*T, x*): *s*(*T*) < *sw*(*x*) → *T* се отменя (абортира) – *x* е променян след старта на *T*
При заявка read(*T, x*): *s*(*T*) > *sw*(*x*) → заявката на *T* се потвърждава, като *sr*(*x*) = max{*s*(*T*), *sr*(*x*)}
При заявка write(*T, x*): *s*(*T*) < *sr*(*x*) →*T* се отменя (абортира) – *x* е прочетен след старта на *T*
При заявка write(*T, x*): *s*(*T*) > *sr*(*x*) → заявката на *T* се потвърждава, като *sw*(*x*) = max{*s*(*T*), *sw*(*x*)}
Планирането с времеви марки води по-често до отмяна на транзакции от това със заключване, защото отменя транзакции, които при заключването само биха били отложени. Същевремнно при времевото маркиране не възниква мъртва точка (поради уникалността и маркиравката на данните).
Оптимистично планиране с времеви марки:Конкурентните транзакции се изпълняват докрай без заключване и сравняване на времеви марки, като същевременно се регистрират всички обекти данни, върху които е изпълнено четене или запис. В края на транзакцията се проверява дали нейните операции са консистентни на операциите на останалите конкурентни транзакци и при откриване на промяна в даден обект след стартирането на тази транзакция, тя се отменя (аналогия с песимистичното времево планиране). Това планиране се имплементира с резервирано работно пространство за всяка транзакция, чието съдържание се записва във файловата система само при успешно изпълнение на транзакцията.Особености на оптимистичното планиране: 1)висок паралелизъм – няма отлагане и мъртви точки; 2)при отмяна на транзакция, тя се рестартира отначало; 3)при високо натоварване на РС (р >80% производителността е по-лоша от тази на песимистичното планиране; 4)рядко се прилага за РС и понеже се възприема като по-сложно за имплементация.