## 8. Конкуретно 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 поддръжат от суперпроцеса на ОС и могат частично да се управляват чрез вградени в езиците системни обръщения. В ерланг (поради наличието на собствена ВМ – OTP и поради общата концепция в подкрепа 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 ерланг 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 междурпроцесна 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 грануларност. 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 функии, които позволяват съхраняването на стойности по даден „ключ“ (т.е. променливи с еднократно присвояване) и последващ достъп до тези стойности от други процеси по генерирания ключ – тоест на практика е възможно да се програмира и с общи променливи в ерланг (GMSV). Тези BIFs (built-in-function) са групирани в модул, наречен Process Dictionary. Това е компромис с принципа GMMP с цел адаптиране на ерланг към стила на повече 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 % изпращане на съобщение M до Pid без потвърждение, за групово предаване (multicasting) – *рекурсивн, пр:* Pid1 ! Pid2 ! Pid3 … ! M % т.к. всяка !-операция връща M. Полученото M се сравнява последователно с Pattern*i* и с опционален Guard*i*, като при успех се изпълняват Expressions*i.* N.B. и при неуспех стойността на M 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:strart/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 ерланг 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 максималния 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 еrlang: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 произвоелн 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.

## Сърверен процес: Итеративен сърверен 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 сърверния 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 дункионалния 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 Pattern*i* се изпълняват Expressions*i* 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 – конкурентко блокиране на достъпа до общите променливи от един от процесите в групата: 1)race condition възниква при невъзможност да се определи статично кой от прецесите пръв ще успее да изпълни критичната зона; 2)deadlock е взаимно блокиране на процесите при достигане до общи критични зони – особено при приоритетни процеси: високприоритетен 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 изпълянва заключване чрез ресурсния семафор, викайки mutex:wait() и получава OK от свободния семафор; след достъпа ресурсът се освобождава от P1 с mutex:signal() като 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 OК 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