Василенко Анатолий, 321 группа

**Какие есть Анти LINPACK способы оценки производительности (по каким критериям осуществляют градации) (Анти LINPACK тест производительности). Какой есть универсальный тест оценки производительности сейчас.**

Перечислим основные тесты проверки производительности (кроме LINPAC):

1. HPCG benchmark – приходит на смену LINPACK
2. Whetstone
3. Dhrystone
4. Perfect
5. SPEC (сейчас сильно развивается, может быть использована вместо LINPACK)
	1. SPEC\_89
	2. SPEC\_92
	3. SPEC\_SDM
	4. SPEC\_SFS
6. AIM
7. EDN
8. Livermore Loops (Lawrence Livermore Loops)
9. Решето Эратосфена
10. LAPACK (является развитием библиотеки LINPACK, т.к. в вопросе спрашивалось «анти», то рассматривать не будем)
11. TPC-A – TPC-E (эти тесты предусмотрены для проверки производительности баз данных, и потому здесь рассматриваться не будут)

Теперь охарактеризуем эти системы оценки:

**Whetstone**

Комплект тестов Whetstone состоит из нескольких модулей, имитирующих программную нагрузку в наиболее типичных режимах исполнения вычислительных задач (целочисленная арифметика, арифметика с плавающей точкой, операторы типа IF, вызовы функций и т.д.). Каждый модуль выполняется многократно, от 12 до 899 раз, а производительность рассчитывается как отношение числа Whetstone-инстpукций к суммарному времени выполнения всех модулей пакета. Этот результат представляется в KWIPS (Kilo Whetstone Instructions Per Second) или в MWIPS (Mega Whetstone Instructions Per Second).

Интеpпpетация pезультатов:

* Пакет Whetstone ориентирован на оценку производительности обработки чисел с плавающей точкой.
* Большое число обращений к библиотеке математических функций, заложенное в тесты Whetstone, требует особой осторожности при сравнении результатов, полученных для разных компьютеров: фирмы-изготовители имеют возможность оптимизировать оценку Whetstone, внося изменения в библиотеку. Во всяком случае, не следует забывать, что тесты Whetstone дают надежные ориентиры только в отношении задач с большой интенсивностью использования стандартных математических функций.
* Поскольку тестовые модули Whetstone представлены очень компактным исполнительным кодом (весь пакет Whetstone в C-версии занимает около 2 Кбайт кода), для современных процессоров они не позволяют оценить эффективность механизма динамической подкачки команд в кэш инструкций.
* Особенностью рассматриваемых тестов является почти полное отсутствие локальных переменных. В результате оценки Whetstone в значительной степени зависят от эффективности функционирования ресурсов компьютера, обеспечивающих доступ к оперативной памяти и буфеpизацию данных в пpоцессоpе (включая количество регистров, емкость кэш-памяти данных и механизм ее замещения), а также от качества реализованных в компиляторе оптимизирующих алгоритмов размещения глобальных переменных в регистрах. Однако это же обстоятельство делает тесты Whetstone практически нечувствительными к средствам повышения эффективности работы с локальными переменными (например, динамическое переключение регистровых окон MORS в процессорах SPARC почти не сказывается на величине Whetstone-оценки).
* Единственная ''официальная'' версия тестов Whetstone - это Pascal-версия (Pascal Evaluation Suite), зарегистрированная Британским обществом стандартов (British Standards Institution - Quality Assurance, BSI-QAS).

**Dhrystone**

Dhrystone по своей структуре напоминает Whetstone. В состав Dhrystone включено 12 модулей, представляющих различные типовые режимы обработки. Тесты Dhrystone предназначены для оценки производительности особого рода - относящейся к функционированию конкретных видов системного и прикладного ПО (операционные системы, компиляторы, редакторы и т. д.).

Естественно, это наложило заметный отпечаток на структуру данных и исполнительного кода: в тестах Dhrystone отсутствует обработка чисел с плавающей точкой, зато преобладают операции над другими типами данных (символы, строки, логические переменные, указатели и т. п.). Кроме того, по сравнению с тестами Whetstone уменьшено количество циклических конструкций, используются более простые вычислительные выражения, но возросло число операторов IF и вызовов процедур.

В 1984 г. Пакет появился на языке Ada, позднее появились его версии на языках Pascal и Cи. Тестовые процедуры Dhrystone объединены в один измерительный цикл, который содержит 101 оператор в Pascal- версии или 103 оператора в C-версии. Этот глобальный цикл принят за единицу работы (один Dhrystone), а производительность измеряется в количестве измерительных циклов, выполненных за секунду (Dhrystone s/s). Пpавда, в последнее вpемя пpи публикациях оценок Dhrystone стали пpименяться и дpугие единицы измеpения - MIPS VAX (см. подробнее ниже, где описываются SPEC).

Автор тестов Dhrystone настаивает на соблюдении трех основных правил измерения и сравнения производительности с помощью этого пакета:

* Должна использоваться версия не ниже 2.1, поскольку более ранняя редакция (1.1) может дать не вполне корректные оценки производительности. В ней результаты вычислений не печатаются и не используются, так что компилятор может распознать часть исходных операторов теста на языке высокого уровня как ''мертвый код'' и подавить для них генерацию исполнительного кода, что не допустимо.
* Модули пакета должны быть откомпилированы раздельно - объединение процедур (in-lining) не допускается. Впрочем, при компиляции C-версии Dhrystone разрешается включать библиотечные функции в исполнительный модуль теста.
* При сравнении компьютеров необходимо использовать результаты, полученные для одной и той же версии Dhrystone . Должен совпадать не только номер редакции (скажем, Dhrystone 2.1), но и язык (Pascal, Ada или C). Это требование связано с тем, что версии на языках Pascal и Ada дают оценки производительности примерно на 10% выше, чем C-версия. Основная причина этого кроется в организации обработки строк в этих языках.
* Исполнительный код Dhrystone содержит множество команд условного перехода, в том числе внутpи циклических конструкций. Поэтому тесты Dhrystone дают достаточно высокую нагрузку на кэш инструкций и позволяют выявить такие принципиальные особенности аппаратуры, как эффективность предсказания переходов и замещения кэша инструкций.
* Пакет Dhrystone оpиентиpован на обpаботку локальных пеpеменных, что резко снижает нагрузку на тракт процессор – память.

**Perfect** (PERFormance Evaluation for Cost-effective Transformations)

В первоначальной концепции контрольных тестов Perfect измерению производительности отводилась второстепенная роль: результаты тестирования должны были демонстрировать увеличение скорости обработки после оптимизации программ. Основное внимание авторы пакета сосредоточили на методике определения величины и сложности усилий, предпринимаемых для минимизации времени выполнения приложений. Однако идея не распространилась и Perfect стал рядовым тестом для проверки производительности.

Тесты Perfect выполняются дважды - до и после оптимизации. В остальном тестовый пакет Perfect ничем особо не примечателен: это комплект из 13 прикладных Fortran-программ, представляющих четыре типа вычислительных задач - аэро- и гидродинамики, моделирования химических и физических процессов, инженерного проектирования, а также обработки сигналов.

Первоначально пакет Perfect предназначался для оценки производительности суперкомпьютеров, однако с развитием микропроцессорных систем область применения этих контрольных тестов значительно расширилась, и сегодня их часто используют для хаpактеpистики быстpодействия Unix-серверов и рабочих станций.

Пакет Perfect ориентирован на измерение производительности при выполнении научных вычислительных программ, содержащих большое количество арифметических операций над числами с плавающей точкой удвоенной точности. Кроме того, тестовые программы, за редким исключением, имеют очень высокий уровень параллелизма: доля векторизуемых операций (до оптимизации) в среднем составляет 65%.

Пакет Perfect содержит большие программы (от 500 до 19000 строк на языке Fortran), обрабатывающие достаточно крупные массивы данных (в среднем 8 Мбайт на каждую программу; для одной из программ пакета эта величина превышает 16 Мбайт). Еще одно, уникальное свойство тестов Perfect - значительный объем ввода/вывода, достигающий почти 500 Мбайт. Таким обpазом, результаты, полученные с помощью контрольных тестов Perfect, характеризуют системную производительность компьютера (процессор + оперативная память + средства ввода/вывода).

**SPEC**

Изначально идею начали реализовывать Apollo Computers, Hewlett- Packard, Mips Computer Systems и Sun Microsystems. Они образовали под эгидой еженедельника ''Electronic Engineering Times'' консорциум по оценке производительности систем (Systems Performance Evaluation Cooperative, SPEC). Позднее к ним присоединились AT&T, Bull, Control Data, Compaq, Data General, Digital Equipment, Dupont, Fujitsu, IBM, Intel, Intergraph, Motorola, NCR, Siemens Nixdorf, Silicon Graphics, Solbourne, Stardent и Unisys. Для публикации своих сообщений консорциум выпускает регулярную газету ''SPEC Newsletter''.

Новорожденная организация провозгласила своей целью подбор, стандартизацию и распространение тестовых программ для измерения производительности быстродействующих компьютеров.

Была введена концептуально новая шкала единиц измерения характеристик системы, нормированных на параметры компьютера DEC VAX 11/780.

Сегодня благодаря усилиям консорциума SPEC широко используются наборы тестов SPEC\_89 и SPEC\_92, постепенно вытесняющие из обращения пакеты первого поколения - Whetstone, Dhrystone , Linpack, Livermoore Loops и др.

Менее известны последние продукты SPEC - тесты SPEC\_SFS и SPEC\_SDM.

**SPEC\_89**

Сегодня уже считается устаревшим. Однако существуют по крайней мере два обстоятельства, которые поддерживают интерес к тестам SPEC\_89: во-первых, методология более современных тестов SPEC\_92 во многом повторяет подходы, реализованные в пакете SPEC\_89, а во-вторых, производители компьютеров продолжают приводить характеристики SPEC\_89 наравне с оценками SPEC\_92. Пакет SPEC\_89 включает два тестовых набора - CINT89, состоящий из четырех программ целочисленной обработки, и CFP89, объединяющий шесть программ со значительным объемом операций над числами с плавающей точкой двойной точности. Все десять модулей представляют собой широкий спектр достаточно сложных решаемых задач на языках C и Fortran.

Методика SPEC\_89 предполагает формирование десяти дифференциальных оценок SPECratio, каждая из которых определяется в условных единицах по результатам измерения времени исполнения одного тестового модуля из наборов CINT89 и CFP89: вpемя выполнения модуля на тестиpуемом компьютеpе SPECratio = вpемя выполнения модуля на VAX 11/780.

Интегральной характеристикой производительности компьютера служит показатель SPECmark, являющийся средним геометрическим всех десяти частных оценок SPECratio. Некоторое время назад к параметру SPECmark добавились еще две оценки - SPECint89 и SPECfp89, раздельно характеризующие быстродействие компьютера при обработке целочисленных данных и вещественных чисел.

**SPEC\_92**

В январе 1992 г. была опубликована новая редакция комплектов тестов SPEC CINT92 и CFP92 под общим названием SPEC\_92. По сравнению с предыдущей версией в обоих тестовых наборах SPEC\_92 расширен состав программ: целочисленные тесты дополнены двумя, а тесты плавающей точки - восемью программами. Таким образом, число модулей в CINT92 достигло шести, а в CFP92 – 14.

Методика расчета основных характеристик производительности в SPEC\_92 почти не изменилась. Но одно качественное новшество в SPEC\_92 все-таки введено: тестируемый компьютер выполняет задание, состоящее из множества копий одной программы, а показателем производительности многопроцессорной обработки служит количество копий, завершенных за определенный интервал времени.

Таким образом, SPECrate\_int92 и SPECrate\_fp92 оценивают среднюю скорость выполнения задач в многопроцессорном режиме работы системы. Кроме того, эти показатели позволяют получить представление о возможностях компилятора по организации параллельного мультизадачного кода, а также операционной системы - по эффективному динамическому распределению ресурсов системы (в частности, процессоров) между выполняемыми параллельными программами. Это делает оценки SPECrate\_int92 и SPECrate\_fp92 особенно представительными для SMP-систем коллективного пользования, работающих в пакетном режиме.

**SPEC\_SDM**

Название этой группы тестов происходит от термина System Development/Multitasking, определяющего область их применения - измерение производительности системы при использовании инструментальных средств разработки приложений в многопользовательской сpеде Unix. Действующая pедакция SPEC\_SDM включает два самостоятельных теста, основанных на системных Unix-процедурах с высоким содержанием команд типа find, mkdir, cc и т.д.

**SPEC\_SFS**

Само название этих тестов - SPEC\_SFS (SPEC System-level File Server) - определяет их целевую функцию, а именно измерение производительности сетевого файл-сервера NFS. В каком-то смысле тесты SPEC\_SFS представляют собой серьезную альтернативу тестам TPC, поскольку выгодно отличаются от них низкой стоимостью пpоведения испытаний, а также простотой и наглядностью представления результатов. Действующая версия SPEC\_SFS Release 1.0 содержит один синтетический тест 097.LADDIS, который позволяет измерить две основные характеристики сервера - время ответа на запрос станции/клиента и пропускную способность ввода/вывода.

Любопытная особенность тестов SPEC\_SFS состоит в возможности получения не одиночного, ''точечного'' результата для фиксированной нагрузки, а целого спектра значений, характеризующих работу системы при ее возрастании. Результаты испытаний пpедставляются таблицей, состоящей из двух колонок: ''Среднее время реакции сервера NFS'' (Average NFS Response Time), выраженное в миллисекундах, и ''Пропускная способность сервера NFS'' (NFS Throughput), измеряемая в количестве операций ввода/вывода, выполненных за секунду (SPECnfs\_A93 Ops/Sec). Затем эти данные офоpмляются в виде графика, на котором представлены среднее время реакции и пропускная способность системы и который позволяет сравнить характеристики различных компьютеров.

**AIM**

Контpольно-оценочные тесты AIM System Benchmark pазpаботаны компанией AIM Technology, специализиpующейся на экспеpтизе Unix-систем с использованием собственных методик и сpедств измеpения пpоизводительности, что является одновpеменно и достоинством и недостатком: независимость от пpоизводителей гаpантиpует объективность pезультатов, но закpытый хаpактеp методологии испытаний не позволяет контpольным тестам AIM получить статус общепpизнанного стандаpта. Самые често используемые спецификации:

* индекс пpоизводительности (Performance Index, или Performance Rating) - показатель быстpодействия пpоцессоpа, пpедставляемый в условных единицах. Действующая веpсия тестов AIM System Benchmark (Suite III) использует смесь опеpаций шести типов (по теpминологии AIM - моделей): обмен с опеpативной памятью (20%), плавающая аpифметика (10%), целочисленная аpифметика (20%), межпpоцессоpное взаимодействие (10%), вызовы функций (20%) и ввод/вывод на диск (20%);
* максимальная нагpузка (Maximum User Load), опpеделяющая максимальное число заданий, котоpое может быть выполнено системой в полной конфигуpации (пpи наибольшем числе пользователей, использующих pежим On-Line) за одну минуту;
* показатель пpопускной способности (Throughput Graph), пpедставляющий число заданий, выполняемых компьютеpом в минуту (jobs/m) пpи интенсивной нагpузке на подсистему ввода/вывода.

**EDN**

EDN - Hабор контрольных тестов оценки производительности при обработке символьных и битовых строк. Hаиболее часто применяются пять модулей:

* E - поиск строки символов;
* F - операции проверки и установки состояния битов;
* вставка символов в связанный текст;
* быстрая сортировка;
* преобразование матрицы битов.

**Livermore Loops (Lawrence Livermore Loops)**

В конце 70-х годов в Ливерморской национальной физической лаборатории Лоуренса были разработаны тесты для оценки производительности суперкомпьютеров CRAY. Автор этих контрольных тестов доктор Макмагон (F. H. McMahon), выбрал из Fortran- программ, выполняющих вычислительные задачи ядерной физики, 18 наиболее представительных циклических процедур обработки чисел с плавающей точкой. Позднее количество циклов было доведено до 24. Ориентация Ливерморских циклов на обработку вещественных чисел определяет представление результатов в MFLOPS, причем для каждого тестового цикла формируется своя оценка производительности. Как следствие, в целях анализа быстpодействия компьютера из 24 частных результатов тестирования может быть выбрано любое подмножество, наиболее точно отвечающее задачам пользователя. Кроме того, Ливерморские циклы, как и контрольный пакет Linpack, допускают проведение измерений производительности при различном уровне параллелизма операций; обычно каждый тест выполняется три раза при различном числе ''оборотов'' цикла.

**Pешето Эратосфена**

Контрольный тест ''Решето Эратосфена'' (Sieve of Eratosthenes) был весьма популярен в конце 80-х годов и применялся для оценки производительности ПК и хаpактеpистики системного ПО машин малого класса. Подавляющую часть программной нагрузки составляют выражения присвоения констант (33%) и управления циклами (50%), зато на долю вычислительных операторов приходится всего 5% кода. Для современных компьютеров, даже персональных, тест Prime не позволяет получить сколько-нибудь представительной оценки производительности, поэтому ориентироваться на результаты, полученные с его помощью, не рекомендуется.

//=====

Очень распространённым тестом оценки производительности является LINPACK, однако его постепенно начинают вытеснять другие системы, такие как SPEC, которые являются более универсальными.

В последний буквально год была изобретена ещё одна система, которая потенциально может выступить в роли универсальной системы измерения производительности – HPCG Brenchmark.

Cистема HPCG обладает следующими особенностями:

1. Позволяет покрыть основные вычислительные операции и операции обмена данными между различными элементами вычислительной системы.
2. Не большой размер кода, позволяющий покрыть большинство основных математических вычислений.
3. Обладает похожестью на реальные вычислительные задачи, т.е. улучшение результатов для этого теста будет неизбежно приводить к улучшению результатов и при выполнении реальных задач.
4. Так же проводится тестирование эффективности работы с локальной памятью в вычислительной системе.
5. Идентифицирует и измеряет различия в побитовые различия идентичных вычислений. (Проблема различий идентичных вычислений основана на том, что операции для чисел с плавающей запятой не являются вполне ассоциативными в вычислительной системе, где количество разрядов ограничено)

//=============Я в начале не на тот вопрос начал отвечать, и поэтому описывал систему LINPAC, до конца оно не доделано, однако и выбрасывать жалко, поэтому я оставил ===================================

LINPACK была разработана в 1970-х. Активно использует интерфейс [BLAS](http://ru.wikipedia.org/wiki/Basic_Linear_Algebra_Subprograms) для обработки матриц и векторов. Linpack представляет собой набор функций линейной алгебры, часто используемых в вычислительных Fortran-программах (так же существует реализация библиотеки на си). Программы пакета выполняют обработку двумерных матриц, размер которых является основным параметром тестирования (чаще всего применяются матрицы 100х100 или 1000х1000): чем больше элементов в матрице, тем выше параллелизм операций при тестировании производительности. Особую значимость этот параметр имеет для компьютеров с векторной архитектурой. Дело в том, что практически все современные компьютеры широко используют средства параллельной обработки (конвейеризованная и/или суперскалярная арифметика, VLIW-архитектура процессора, MPP-организация системы и т. д.), поэтому оценка производительности при разной глубине программного параллелизма весьма показательна для любой современной системы.

Также под названием LINPACK часто понимают [тесты производительности LINPACK](http://ru.wikipedia.org/w/index.php?title=%D0%A2%D0%B5%D1%81%D1%82%D1%8B_%D0%BF%D1%80%D0%BE%D0%B8%D0%B7%D0%B2%D0%BE%D0%B4%D0%B8%D1%82%D0%B5%D0%BB%D1%8C%D0%BD%D0%BE%D1%81%D1%82%D0%B8_LINPACK&action=edit&redlink=1) (решение плотной СЛАУ методом LU-декомпозиции). Изначально тест был опубликован в приложении "B" к документации библиотеки и предназначался для грубой экстраполяции времени работы библиотеки.

Различные варианты тестов:

1. linpack100 (матрица 100 на 100; 1977 год)
2. linpack1000 (матрица увеличена до 1000 элементов в каждом измерении, 1986 год)
3. linpack parallel (1000 элементов, параллельная обработка)
4. [HPL](http://ru.wikipedia.org/w/index.php?title=High-performance_Linpack&action=edit&redlink=1) (High-performance Linpack, произвольные размеры, первые версии выпущены в 1991-1993 годах) — популярный [тест производительности](http://ru.wikipedia.org/wiki/%D0%A2%D0%B5%D1%81%D1%82_%D0%BF%D1%80%D0%BE%D0%B8%D0%B7%D0%B2%D0%BE%D0%B4%D0%B8%D1%82%D0%B5%D0%BB%D1%8C%D0%BD%D0%BE%D1%81%D1%82%D0%B8), предназначенный для оценки производительности параллельных вычислительных систем и созданный на базе некоторых функций из библиотеки LINPACK.

Самым распространённым мировым рейтингом является TOP500. По правилам в TOP500 используется HPL на больших размерах, в качестве формата элементов возможно использование только [64-разрядного представления дробных чисел IEEE](http://ru.wikipedia.org/wiki/Double_precision), не допускается применение [метода Штрассена](http://ru.wikipedia.org/wiki/%D0%90%D0%BB%D0%B3%D0%BE%D1%80%D0%B8%D1%82%D0%BC_%D0%A8%D1%82%D1%80%D0%B0%D1%81%D1%81%D0%B5%D0%BD%D0%B0) или более коротких форматов плавающих чисел.

LAPACK была создана в основном для работы с векторными суперкомпьютерами с разделяемой памятью.

На 17 ноября 2013 года в TOP500 лидирует китайский суперкомпьютер Tianhe-2, Rmax = 33.863, Rpeak = 54.902.

Так же существуют TOP-50 СНГ и TOP-50 China, которые тоже основаны на тестах LINPAC.

**Виды параллельных вычислительных процессов в ЭВМ. И их синхронизация (что-такое треды, нити, программы, …)**

Многопоточность — свойство платформы (например, [операционной системы](http://ru.wikipedia.org/wiki/%D0%9E%D0%BF%D0%B5%D1%80%D0%B0%D1%86%D0%B8%D0%BE%D0%BD%D0%BD%D0%B0%D1%8F_%D1%81%D0%B8%D1%81%D1%82%D0%B5%D0%BC%D0%B0), [виртуальной машины](http://ru.wikipedia.org/wiki/%D0%92%D0%B8%D1%80%D1%82%D1%83%D0%B0%D0%BB%D1%8C%D0%BD%D0%B0%D1%8F_%D0%BC%D0%B0%D1%88%D0%B8%D0%BD%D0%B0) и т. д.) или [приложения](http://ru.wikipedia.org/wiki/%D0%9F%D1%80%D0%B8%D0%BA%D0%BB%D0%B0%D0%B4%D0%BD%D0%BE%D0%B5_%D0%BF%D1%80%D0%BE%D0%B3%D1%80%D0%B0%D0%BC%D0%BC%D0%BD%D0%BE%D0%B5_%D0%BE%D0%B1%D0%B5%D1%81%D0%BF%D0%B5%D1%87%D0%B5%D0%BD%D0%B8%D0%B5), состоящее в том, что [процесс](http://ru.wikipedia.org/wiki/%D0%9F%D1%80%D0%BE%D1%86%D0%B5%D1%81%D1%81_%28%D0%B8%D0%BD%D1%84%D0%BE%D1%80%D0%BC%D0%B0%D1%82%D0%B8%D0%BA%D0%B0%29), порождённый в операционной системе, может состоять из нескольких потоков, выполняющихся «[параллельно](http://ru.wikipedia.org/wiki/%D0%9F%D0%B0%D1%80%D0%B0%D0%BB%D0%BB%D0%B5%D0%BB%D1%8C%D0%BD%D1%8B%D0%B5_%D0%B2%D1%8B%D1%87%D0%B8%D1%81%D0%BB%D0%B5%D0%BD%D0%B8%D1%8F)», то есть без предписанного порядка во [времени](http://ru.wikipedia.org/wiki/%D0%92%D1%80%D0%B5%D0%BC%D1%8F). При выполнении некоторых задач такое разделение может достичь более эффективного использования [ресурсов вычислительной машины](http://ru.wikipedia.org/wiki/%D0%92%D1%8B%D1%87%D0%B8%D1%81%D0%BB%D0%B8%D1%82%D0%B5%D0%BB%D1%8C%D0%BD%D1%8B%D0%B5_%D1%80%D0%B5%D1%81%D1%83%D1%80%D1%81%D1%8B).

Существуют два принципиально разных понятия: процесс (программа – хотя это менее корректно, потому что процесс – это нечто «удушевлённое» в операционной система, для процесса есть системные таблицы, и он выполняется, я программа – это строго говоря – просто код, текст) и поток (или ещё называют - нить)

Процесс – это поставленная на выполнение программа. Поток – это наименьшая единица обработки, исполнение которой может быть [назначено](http://ru.wikipedia.org/wiki/%D0%94%D0%B8%D1%81%D0%BF%D0%B5%D1%82%D1%87%D0%B5%D1%80_%D0%BE%D0%BF%D0%B5%D1%80%D0%B0%D1%86%D0%B8%D0%BE%D0%BD%D0%BD%D0%BE%D0%B9_%D1%81%D0%B8%D1%81%D1%82%D0%B5%D0%BC%D1%8B) [ядром операционной системы](http://ru.wikipedia.org/wiki/%D0%AF%D0%B4%D1%80%D0%BE_%D0%BE%D0%BF%D0%B5%D1%80%D0%B0%D1%86%D0%B8%D0%BE%D0%BD%D0%BD%D0%BE%D0%B9_%D1%81%D0%B8%D1%81%D1%82%D0%B5%D0%BC%D1%8B). В одном процессе может быть несколько потоков. Одно из главных отличий – это то, что несколько потоков могут существовать в рамках одного и того же процесса и совместно использовать его ресурсы, такие как [память](http://ru.wikipedia.org/wiki/%D0%A0%D0%B0%D0%B7%D0%B4%D0%B5%D0%BB%D1%8F%D0%B5%D0%BC%D0%B0%D1%8F_%D0%BF%D0%B0%D0%BC%D1%8F%D1%82%D1%8C), тогда как процессы не разделяют этих ресурсов. В следствие этого отличаются способы взаимодействия между процессами и потоками.

Основные отличия потоков и процессов:

* процессы, как правило, независимы, тогда как потоки выполнения существуют как составные элементы процессов
* процессы несут значительно больше информации о состоянии, тогда как несколько потоков выполнения внутри процесса совместно используют информацию о состоянии, а также память и другие [вычислительные ресурсы](http://ru.wikipedia.org/wiki/%D0%92%D1%8B%D1%87%D0%B8%D1%81%D0%BB%D0%B8%D1%82%D0%B5%D0%BB%D1%8C%D0%BD%D1%8B%D0%B5_%D1%80%D0%B5%D1%81%D1%83%D1%80%D1%81%D1%8B)
* процессы имеют отдельные адресные пространства, тогда как потоки выполнения совместно используют их адресное пространство
* процессы взаимодействуют только через предоставляемые системой механизмы [связей между процессами](http://ru.wikipedia.org/wiki/%D0%9C%D0%B5%D0%B6%D0%BF%D1%80%D0%BE%D1%86%D0%B5%D1%81%D1%81%D0%BD%D0%BE%D0%B5_%D0%B2%D0%B7%D0%B0%D0%B8%D0%BC%D0%BE%D0%B4%D0%B5%D0%B9%D1%81%D1%82%D0%B2%D0%B8%D0%B5)
* [переключение контекста](http://ru.wikipedia.org/wiki/%D0%9F%D0%B5%D1%80%D0%B5%D0%BA%D0%BB%D1%8E%D1%87%D0%B5%D0%BD%D0%B8%D0%B5_%D0%BA%D0%BE%D0%BD%D1%82%D0%B5%D0%BA%D1%81%D1%82%D0%B0) между потоками выполнения в одном процессе, как правило, быстрее, чем переключение контекста между процессами.

В таких системах WindowsNT и OS/2 разница между переключением контекста процессов и потоков – очень велика, в других операционных системах, она обычно немного меньше, хотя всё равно достаточно ощутима.

Для синхронизации процессов могут использоваться такие средства, как (хотя какие-то распространены больше, например, сигналы, сокеты, MPI, а какие-то меньше – например обычные файлы):

1. Обычные файлы
2. Сигналы
3. Сокеты – для взаимодействия по сети
4. Канал (pipes) – для взаимодействия родственных процессов
5. Именованный канал – для взаимодействия любых процессов
6. Семафоры (это составная часть IPC – Inter Process Communication)
7. Разделяемая память (составная часть IPC)
8. Обмен сообщениями (в частности MPI)
9. Проецируемый в память файл (в последнее время используется мало)
10. Очередь сообщений (составная часть IPC)

## Проблемы при синхронизации процессов впринципе такие же, что и проблемы при синхронизации потоков, поэтому я рассмотрю их немного позже.

## Способы синхронизации потоков:

В многопоточной среде часто возникают проблемы, связанные с использованием параллельно исполняемыми потоками одних и тех же данных или устройств. Для решения подобных проблем используются такие методы взаимодействия потоков, как взаимоисключения (мьютексы), семафоры, критические секции события … Рассмотрим их подробнее:

* Взаимоисключения (mutex, [мьютекс](http://ru.wikipedia.org/wiki/%D0%9C%D1%8C%D1%8E%D1%82%D0%B5%D0%BA%D1%81)) — это объект синхронизации, который устанавливается в особое сигнальное состояние, когда не занят каким-либо потоком. Только один поток владеет этим объектом в любой момент времени, — одновременный доступ к общему ресурсу исключается. После всех необходимых действий mutex освобождается, предоставляя другим потокам доступ к общему ресурсу. Объект может поддерживать рекурсивный захват второй раз тем же потоком, увеличивая счетчик, не блокируя поток, и требуя потом многократного освобождения. Тем не менее есть и такие реализации, которые не поддерживают такое и приводят к [взаимной блокировке](http://ru.wikipedia.org/wiki/%D0%92%D0%B7%D0%B0%D0%B8%D0%BC%D0%BD%D0%B0%D1%8F_%D0%B1%D0%BB%D0%BE%D0%BA%D0%B8%D1%80%D0%BE%D0%B2%D0%BA%D0%B0) потока при попытке рекурсивного захвата. Например FAST\_MUTEX в ядре Windows. (mutex – основной способ синхронизации в FreeBSD)
* [Семафоры](http://ru.wikipedia.org/wiki/%D0%A1%D0%B5%D0%BC%D0%B0%D1%84%D0%BE%D1%80_%28%D0%B8%D0%BD%D1%84%D0%BE%D1%80%D0%BC%D0%B0%D1%82%D0%B8%D0%BA%D0%B0%29) представляют собой доступные ресурсы, которые могут быть приобретены несколькими потоками в одно и то же время, пока пул ресурсов не опустеет. Тогда дополнительные потоки должны ждать, пока требуемое количество ресурсов не будет снова доступно. Семафоры очень эффективны, поскольку они позволяют одновременный доступ к ресурсам. Семафор есть логическое расширение mutex — семафор со счетчиком 1 эквивалентен mutex, но счетчик может быть и более 1.
* События – это объект, хранящий в себе 1 бит информации («просигнализирован или нет»), над которым определены операции «просигнализировать», «сбросить в непросигнализированное состояние» и «ожидать». Ожидание на просигнализированном событии есть отсутствие операции с немедленным продолжением исполнения потока. Ожидание на непросигнализированном событии приводит к приостановке исполнения потока до тех пор, пока другой поток (или же вторая фаза [обработчика прерывания](http://ru.wikipedia.org/wiki/%D0%9E%D0%B1%D1%80%D0%B0%D0%B1%D0%BE%D1%82%D1%87%D0%B8%D0%BA_%D0%BF%D1%80%D0%B5%D1%80%D1%8B%D0%B2%D0%B0%D0%BD%D0%B8%D1%8F) в ядре ОС) не просигнализирует событие. Возможно ожидание нескольких событий в режимах «любого» или «всех». Возможно также создания события, автоматически сбрасываемого в непросигнализированное состояние после пробуждения первого же — и единственного — ожидающего потока (такой объект используется как основа для реализации объекта «критическая секция»). Активно используются в MS Windows, как в режиме пользователя, так и в режиме ядра. Аналогичный объект имеется и в ядре Linux под названием kwait\_queue.
* [Критические секции](http://ru.wikipedia.org/wiki/%D0%9A%D1%80%D0%B8%D1%82%D0%B8%D1%87%D0%B5%D1%81%D0%BA%D0%B0%D1%8F_%D1%81%D0%B5%D0%BA%D1%86%D0%B8%D1%8F) обеспечивают синхронизацию подобно mutex за исключением того, что объекты, представляющие критические секции, доступны в пределах одного процесса. События, mutex и семафоры также можно использовать в однопроцессном приложении, однако реализации критических секций в некоторых ОС (например, Windows NT) обеспечивают более быстрый и более эффективный механизм взаимно-исключающей синхронизации — операции «получить» и «освободить» на критической секции оптимизированы для случая единственного потока (отсутствия конкуренции) с целью избежать любых ведущих в ядро ОС системных вызовов. Подобно mutex объект, представляющий критическую секцию, может использоваться только одним потоком в данный момент времени, что делает их крайне полезными при разграничении доступа к общим ресурсам.
* Условные переменные (condvars). Сходны с событиями, но не являются объектами, занимающими память — используется только адрес переменной, понятие «содержимое переменной» не существует, в качестве условной переменной может использоваться адрес произвольного объекта. В отличие от событий, установка условной переменной в просигнализированное состояние не влечет за собой никаких последствий в случае, если на данный момент нет потоков, ожидающих на переменной. Установка события в аналогичном случае влечет за собой запоминание состояния «просигнализировано» внутри самого события, после чего следующие потоки, желающие ожидать события, продолжают исполнение немедленно без остановки. Для полноценного использования такого объекта необходима также операция «освободить mutex и ожидать условную переменную атомарно». Активно используются в [UNIX](http://ru.wikipedia.org/wiki/UNIX)-подобных ОС.
* Порт завершения ввода-вывода (IO completion port, IOCP). Реализованный в ядре ОС и доступный через системные вызовы объект «очередь» с операциями «поместить структуру в хвост очереди» и «взять следующую структуру с головы очереди» — последний вызов приостанавливает исполнение потока в случае, если очередь пуста, и до тех пор, пока другой поток не осуществит вызов «поместить». Самой важной особенностью является то, что структуры в него могут помещаться не только явным системным вызовом из режима пользователя, но и неявно внутри ядра ОС как результат завершения асинхронной операции ввода-вывода на одном из дескрипторов файлов. Для достижения такого эффекта необходимо использовать системный вызов «связать дескриптор файла с IOCP». В этом случае помещенная в очередь структура содержит в себе код ошибки операции ввода-вывода, а также, для случая успеха этой операции — число реально введенных или выведенных байт. Реализация порта завершения также ограничивает число потоков, исполняющихся на одном процессоре/ядре после получения структуры из очереди. Таким образом работают pipes в операционной системе linux.
* ERESOURCE. Mutex, поддерживающий рекурсивный захват, с семантикой разделяемого или эксклюзивного захвата. Семантика: объект может быть либо свободен, либо захвачен произвольным числом потоков разделяемым образом, либо захвачен всего одним потоком эксклюзивным образом. Любые попытки осуществить захваты, нарушающее это правило, приводят к блокировке потока до тех пор, пока объект не освободится так, чтобы сделать захват разрешенным. Также есть операции вида TryToAcquire — никогда не блокирует поток, либо захватывает, либо (если нужна блокировка) возвращает FALSE, ничего не делая. Используется в ядре Windows, особенно в файловых системах — так, например, любому кем-то открытому дисковому файлу соответствует структура FCB, в которой есть 2 таких объекта для синхронизации доступа к размеру файла. Один из них — paging IO resource — захватывается эксклюзивно только в пути обрезания файла, и гарантирует, что в момент обрезания на файле нет активного ввода-вывода от кэша и от отображения в память.
* [Rundown protection](http://ru.wikipedia.org/wiki/Rundown_protection). Полудокументированный (вызовы присутствуют в файлах-заголовках, но отсутствуют в документации) объект в ядре Windows. Счетчик с операциями «увеличить», «уменьшить» и «ждать». Ожидание блокирует поток до тех пор, пока операции уменьшения не уменьшат счетчик до нуля. Кроме того, операция увеличения может отказать, и наличие активного в данный момент времени ожидания заставляет отказывать все операции увеличения.

Для наглядности проблемы синхронизации процессов, рассмотрим классический пример: «Задача обедающих философов»

Постановка задачи: Пять безмолвных философов сидят вокруг круглого стола, перед каждым философом стоит тарелка спагетти. Вилки лежат на столе между каждой парой ближайших философов. Каждый философ может либо есть, либо размышлять. Спагетти – не заканчивается. Тем не менее, философ может есть только тогда, когда держит две вилки — взятую справа и слева. Каждый философ может взять ближайшую вилку (если она доступна), или положить — если он уже держит её. Взятие каждой вилки и возвращение её на стол являются раздельными действиями, которые должны выполняться одно за другим. Суть проблемы заключается в том, чтобы разработать модель поведения (параллельный алгоритм), при котором ни один из философов не будет голодать, то есть будет вечно чередовать приём пищи и размышления.

Два основных типа проблем, с которыми мы можем столкнуться при синхронизации процессов:

1. Dead Lock – взаимная блокировка: Например все философы взяли по левой вилке, и каждый ждёт, пока освободится правая вилка, однако она уже никогда не освободится.
2. Проблема не одновременности проверки и присвоения, т.е. могло случиться, что сначала один философ проверил наличие вилки и собрался взять её, но ещё не взял, потом другой философ проверил наличие и собрался взять, но не взял, потом первый философ взял вилку, и потом второй тоже взял вилку, хотя её уже нет, однако он этого не знает, ведь проверка была раньше и прошла успешно. Для решения этой задачи не всегда необходимо использовать способы синхронизации, описанные выше, потому что существуют специальные атомарные директивы процессора, которые сравнивают и в случае успеха, сразу же присваивают новое значение.

Пример неправильного алгоритма, способного привести к deadlock: Например, можно чтобы каждый философ выполнял следующий алгоритм:

* Размышлять, пока не освободится левая вилка. Когда вилка освободится — взять её.
* Размышлять, пока не освободится правая вилка. Когда вилка освободится — взять её.
* Есть
* Положить левую вилку
* Положить правую вилку
* Повторить алгоритм сначала

Этот алгоритм как раз способен породить deadlock, как я уже описывал выше

Существует несколько правильных решений этой задачи, рассмотрим так называемое «Решение на основе монитора». Оно основано на том, что философы могут есть, если ни один из их соседей не ест. В отсутствии блокировок, связанных с вилками, философы должны обеспечивать то, что начало принятия пищи не основывается на старой информации о состоянии соседей. Например: Если философ B видит, что A не ест в данный момент времени, а потом поворачивается и смотрит на C, A мог начать есть, пока философ B смотрит на C. Используя одну взаимоисключающую блокировку, можно избежать этой проблемы. Эта блокировка не связана с вилками, но она связана с решением процедур, которые могут изменить состояние философов. Это обеспечивается монитором.

Алгоритм монитора реализует схему «проверить, взять и положить» и разделяет взаимоисключающую блокировку. Заметим, что философы, желающие есть, не будут иметь вилок. Если монитор разрешает философу, желающему есть, действовать, то философ снова завладевает первой вилкой, прежде чем взять уже свободную вторую.

Стоит заметить, что этот алгоритм монитора не решает проблемы голодания. Например, философ B может бесконечно ждать своей очереди, если у философов A и C периоды приёма пищи всё время пересекаются. Чтобы гарантировать также, что ни один философ не будет голодать, можно отслеживать, сколько раз голодный философ не ел, когда его соседи положили вилки на стол. Если количество раз превысит некий предел, такой философ перейдёт в состояние Голодания и алгоритм монитора форсирует процедуру завладения вилками, выполняя условие недопущения голодания ни одного из соседей.

**Спецкомментарии к средствам параллельного программирования. Спецкомментарии – средство параллельного программирования (в алгоритмических языках) (нужно взять все, но глубоко разобрать какой-то очень подробно)**

Спецкомментарии – это своего рода концепция уточнения и дополнения исходного языка программирования, которые сохраняют корректность программы с точки зрения эталонного стандарта языка и однако в то же время расширяют его возможности. Обычно спецкомментарии могут распространяться в любом месте программы. Некоторые виды спецкомментариев позволяют работать с уже существующей программой (напр. для получения документации или для верификации программы), а некоторые позволяют менять работу программы, при учёте конкретной среды выполнения программы (если их поддерживает компилятор) (напр. добавление в текст макроопределений, дополнения текста возможностями библиотек, изменение рабты уже существующей программы).

Примерами продуктов, работающих с программами на основе спецкомментариев, но не меняющих их поведение, являются системы автоматического документирования Doxygen, Javadoc, PHPdoc.

Сначала были созданы спецкомментарии, которые позволяли трансляторам генерировать более эффективный объектный код, сообщая транслятору информацию, которую нельзя выяснить автоматически. Для мультипроцессорных кластеров была предложена совокупность спецкомментариев, включение которых в тексты программ с последовательного языка позволяют трансляторам получать параллельный объектный код (примером таких спецкомментариев может служить OpenMP, или директивы фирмы CONVEX для кластера SPP1000).

Подобные спецкомментарии созданы для учёта особенностей вычислительной системы, на которой будет работать программный продукт (они могут учитывать работу кеш-памяти, архитектуру системы – NUMA, ccNUMA …, решение проблемы миграции процессов(нитей/тредов) между ядрами, даже вероятность определённого перехода в условном операторе)

Однако конечно же распараллеливание вручную позволяет добиться большего успеха, по сравнению с способами автоматическими распараллеливания.

Так например для систем MPP, чтобы программировать методом передачи сообщений были созданы специальные языки Фортран GNS, и Си GNS. Эти языки были получены из стандартных добавлением средств описания и идентификации параллельных процессов.

Самыми распространёнными языками, основанными на спецкомментариях считаются Фортран HPF (High performance Fortran), Фортран DVM и система OpenMP (open multiprocessing).

Хотя существует очень много различных параллельных расширений и \_диалектов\_ как для языка Фортран, так и для языка Си, например:

1. Фортран: Fortran-DVM, Cray MPP Fortran, F--, Fortran90/95, FortranD95, FortranM, Fx, HPF, Opus, ViennaFortran
2. Си: C-DVM, A++/P++, CC++, Charm/Charm++, Cilk, HPC, HPC++,C/OpenMP, Maisie, Mentat, mpC, MPC++, Parsec, pC++, sC++, uC++

Все системы основанные на спецкомментариях работают примерно одинаково, т.е. по одной схеме. Рассмотрим подробно систему OpenMP (в основном применительно к языку СИ):

OpenMP (Open Multi-Processing) это набор директив компилятора, библиотечных процедур и переменных окружения, которые предназначены для программирования многопоточных приложений на многопроцессорных системах с единой памятью на языках C, C++ и Fortran.

Межпроцессное взаимодействие в большинстве случаев реализовано с помощью MPI – Message Passing Interface.

Разработку спецификации OpenMP ведут несколько крупных производителей вычислительной техники и программного обеспечения, чья работа регулируется некоммерческой организацией, называемой OpenMP Architecture Review Board (ARB). (В основном система OpenMP – это систем спецкомментариев системы SPP1000)

OpenMP реализует параллельные вычисления с помощью многопоточности, в которой «главный» (master) поток создает набор подчиненных (slave) потоков и задача распределяется между ними. Предполагается, что потоки выполняются параллельно на машине с несколькими процессорами (соотношение количества процессоров и количества потоков потоков не регламентируется).

Задачи, выполняемые потоками параллельно, также как и данные, требуемые для выполнения этих задач, описываются с помощью специальных директив препроцессора соответствующего языка — pragma. Например, участок кода на языке Си, который должен исполняться несколькими потоками, каждый из которых имеет свою копию переменной N, предваряется следующей директивой:

**#pragma omp parallel private(x)**

Количество создаваемых потоков может регулироваться как самой программой при помощи вызова библиотечных процедур, так и извне, при помощи переменных окружения.

Ключевыми элементами OpenMP являются

* конструкции для создания потоков (директива **parallel**),
* конструкции распределения работы между потоками (директивы **for** и **section**),
* конструкции для управления работой с данными (выражения **shared** и **private**),
* конструкции для синхронизации потоков (директивы **critical**, **atomic** и **barrier**),
* процедуры библиотеки поддержки времени выполнения (например, **omp\_get\_thread\_num**),
* переменные окружения (например, **OMP\_NUM\_THREADS**).

Существует возможность скомпилировать код с игнорированием директив OpenMP, и тогда код будет выполняться последовательно.

Другая система: DVM - ориентирована на параллелизм данных. В этой модели данные распределяются по вычислительным узлам, а программа преобразуется в параллельную программу, с использованием модели передачи сообщений, либо с общей памятью. Это освобождает программиста от распределения данных между различными вычислительными узлами. Хотя безусловно эта система применима не для всех решаемых задач, а преимущественно для тех, в которых параллельно над различными данными нужно производить вычисления. Фортран HPF – как раз является одним из первых примеров подобного языка программирования, где было сделано всё для полной автоматизации задачи распараллеливания, однако на некоторых задачах он выдавал плохие показатели и в результате появился Фортран DVM – который давал некоторую настраиваемость, т.е. гибкость, и потому был более успешен для большего числа задач.

В результате спецкомментарии для распараллеливания процессов – это очень удобно и иногда это возможно даже и оптимальнее, чем сделал бы программист. Однако так или иначе, но они работают по некоторым заранее заложенным алгоритмам и по большому счёту не могут адаптироваться к решаемой задачу, а потому часто распараллеливанием вручную можно добиться более качественных результатов.

(Т.е. в данном случае – это палка о двух концах – с одной стороны удобство, с другой – скорость, тут как с ассемблером – программы на нём – самые быстрые, однако писать их крайне не удобно, и не смотря на различные оптимизации в компиляторах и прочие ухищрения, добиться той же скорости чо и при написании вручную, получается плохо)