МГТУ
им. Н.Э. Баумана
ТЕХНОПАРК
Mail.Ru Group

Углубленное программирование на языке C / C++

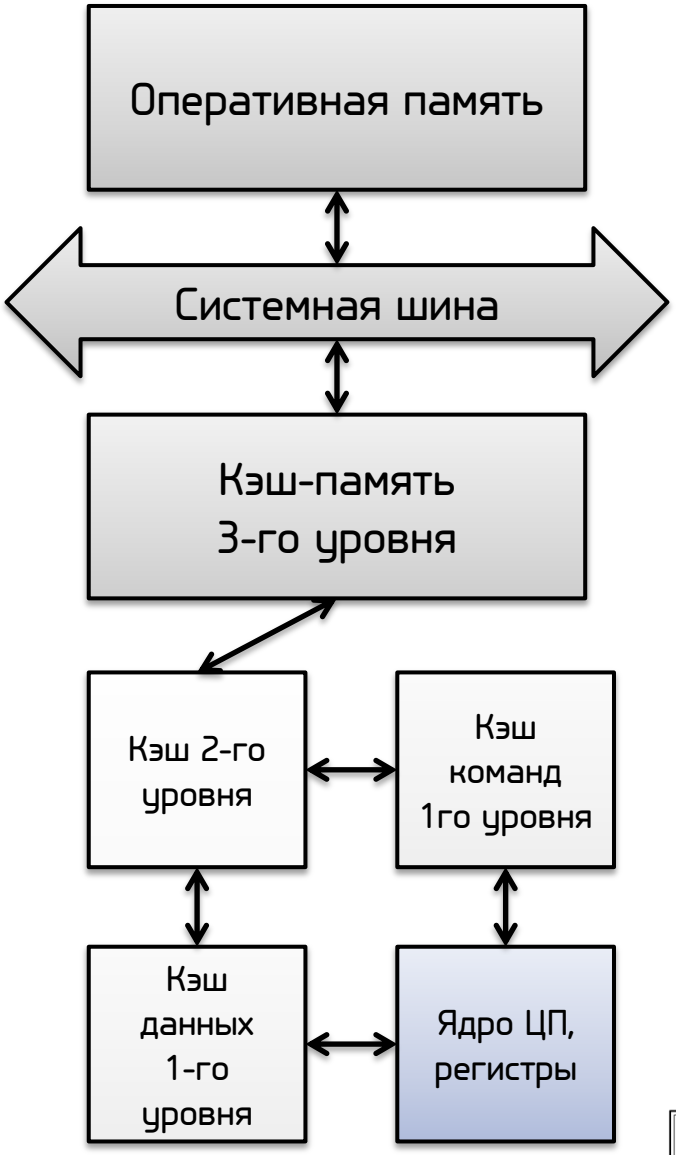
Алексей Петров

Лекция №2. Организация и использование СОЗУ. Основы многопоточного программирования. Вопросы качества структурного кода

1. Оптимизация работы с кэш-памятью ЦП ЭВМ.
2. Анти-шаблоны структурного программирования, их поиск и устранение.
3. Взаимодействие приложения с ОС семейства UNIX.
4. Многопоточное программирование с использованием потоков POSIX.
5. Вопросы производительности и безопасности структурного исходного кода.
6. Постановка индивидуальных задач к практикуму №2.

Кэш-память в архитектуре современных ЭВМ

- **Проблема** — отставание системной шины [и модулей оперативной памяти (DRAM)] от ядра ЦП по внутренней частоте; простой ЦП.
- **Решение** — включение в архитектуру небольших модулей сверхоперативной памяти (SRAM), полностью контролируемой ЦП.
- **Условия эффективности** — локальность данных и кода в пространстве-времени.



При подготовке сл. 3 – 4, 6 – 11 и Прил. 2 использованы материалы доклада на конференции *DEV Labs C++* 2013.

Кэш-память в архитектуре современных ЭВМ: что делать?

- Обеспечивать **локальность данных и команд** в пространстве и времени:
  - совместно хранить совместно используемые данные или команды;
  - не нарушать эмпирические правила написания эффективного кода.
- Обеспечивать **эффективность загрузки** общей (L2, L3) и раздельной кэш-памяти данных (L1d) и команд (L1i):
  - полагаться на оптимизирующие возможности компилятора;
  - помогать компилятору в процессе написания кода.
- Знать **основы организации** аппаратного обеспечения.
- Экспериментировать!

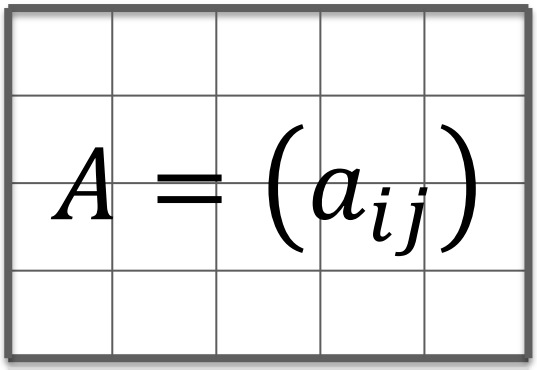
Эффективный обход двумерных массивов

- Простейшим способом повышения эффективности работы с двумерным массивом является **отказ от его обхода по столбцам в пользу обхода по строкам:**
  - для массивов, объем которых превышает размер (выделенной процессу) кэш-памяти данных самого верхнего уровня (напр., L2d), время инициализации по строкам приблизительно втрое меньше времени инициализации по столбцам вне зависимости от того, ведется ли запись в кэш-память или в оперативную память в обход нее (У. Дреппер, 2007).
- Дальнейшая оптимизация может быть связана с анализом и переработкой решаемой задачи в целях снижения частоты кэш-промахов или использования векторных инструкций процессора (SIMD — Single Instruction, Multiple Data).

Эффективный обход двумерных массивов: постановка задачи

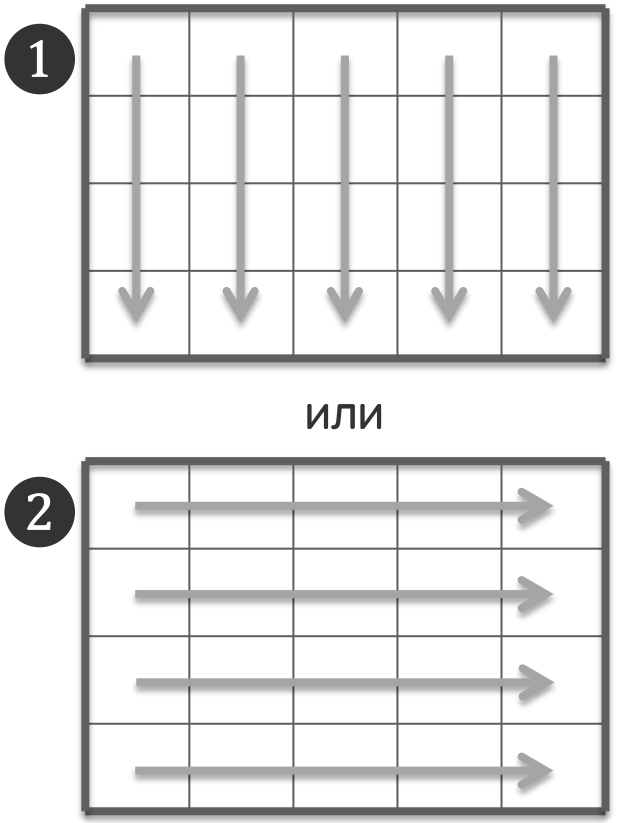
- **Задача.** Рассчитать сумму столбцов заданной целочисленной матрицы. Оптимизировать найденное решение с точки зрения загрузки кэш-памяти данных.

Дано:



Найти:

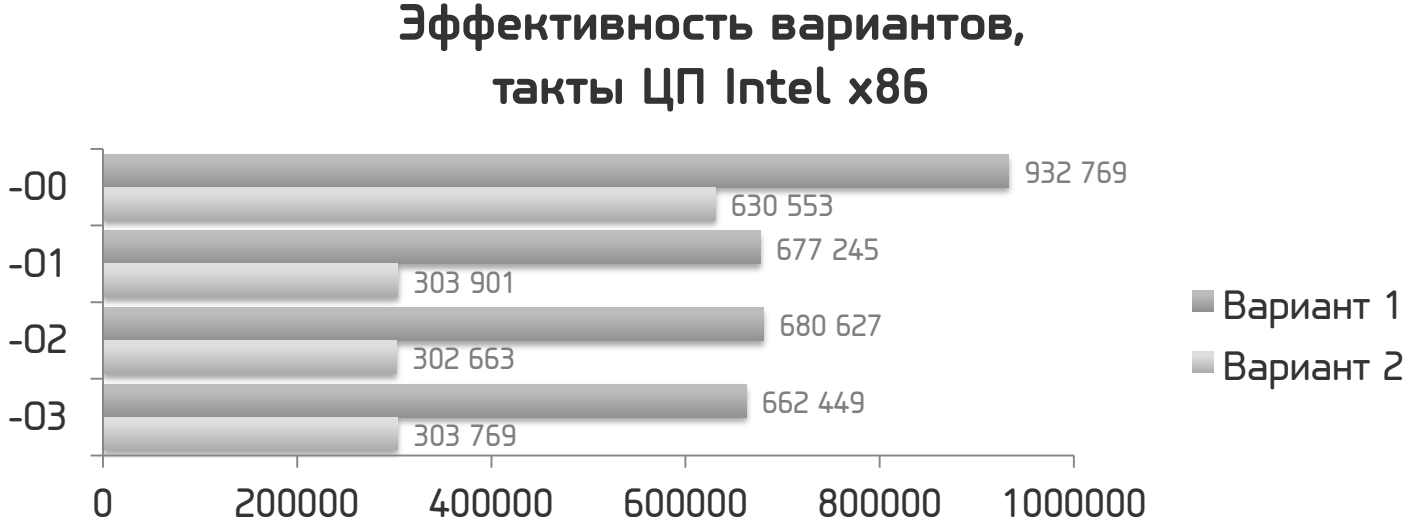
$B = (b_j)$

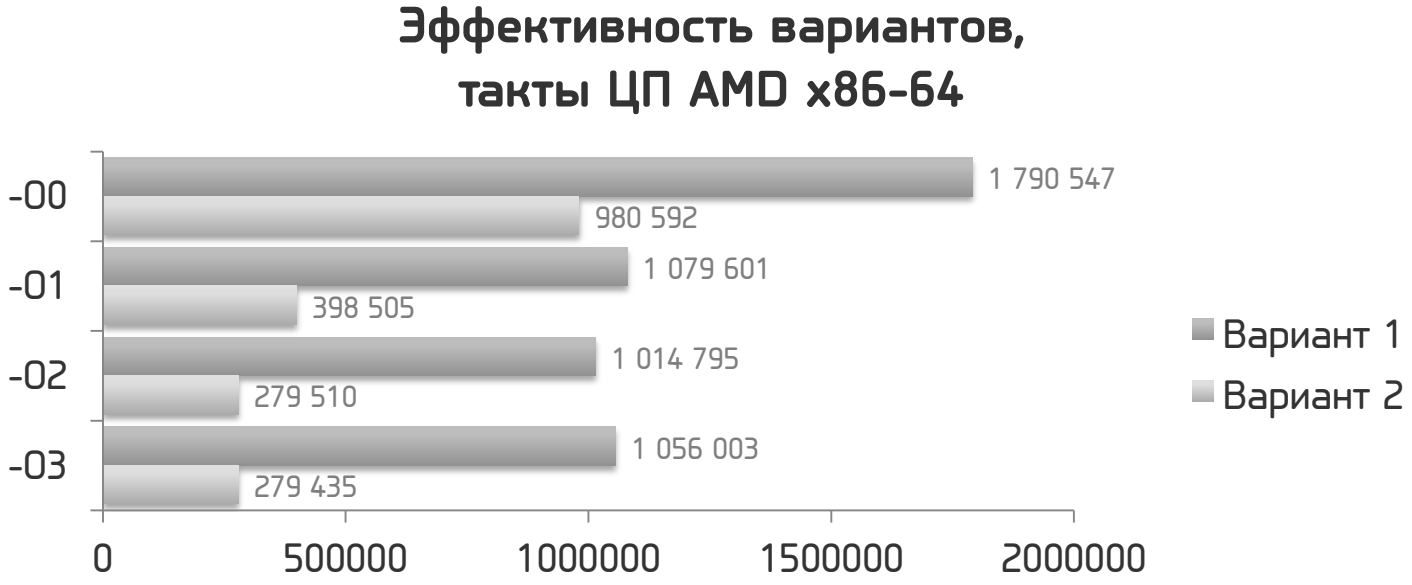


Эффективный обход двумерных массивов: порядок анализа вариантов

- **Размерность задачи:**
  - **набор данных:** 2^8 x 2^8 (2^{16} элементов, 2^{18} байт);
  - **количество тестов:** 100;
  - **выбираемое время:** минимальное.
- **Инфраструктура тестирования:**
  - **x86:** Intel® Core™ i5 460M, 2533 МГц, L1d: 2 x 32 Кб;
  - **x86-64:** AMD® E-450, 1650 МГц, L1d: 2 x 32 Кб / ядро;
  - **ОС:** Ubuntu Linux 12.04 LTS; **компилятор:** GCC 4.8.x.
- **Порядок обеспечения независимости тестов —** 2-фазная программная инвалидация памяти L1d:
  - последовательная запись массива (10^6 элементов, ≈ 2^{22} байт);
  - рандомизированная модификация элементов.

Эффективный обход двумерных массивов:сравнительная эффективность





Эффективный обход двумерных массивов: результаты оптимизации

- Время решения задачи за счет оптимизации обхода данных (без применения SSE-расширений) **снижается в 1,8 – 2,4 раза**:
  - для ЦП **Intel x86**: от 1,5 до 2,2 раза;
  - для ЦП **AMD x86-64**: от 1,8 до 3,8 раза.
- При компиляции с флагами -O0, -O1 результат характеризуется **высокой повторяемостью** на ЦП с выбранной архитектурой (Intel x86 / AMD x86-64):
  - относительный рост эффективности колеблется в пределах **20 – 25%%**.
- Применение векторных SIMD-инструкций из наборов команд SSE, SSE2, SSE3 позволяет улучшить полученный результат еще на **15 – 20%%.**

Почему оптимизация работы с кэш-памятью того стоит?

- Несложная трансформация вычислительноемких фрагментов кода позволяет добиться **серьезного роста** скорости выполнения:
  - разбиение на квадраты (square blocking) и пр.
- **Причина — высокая «стоимость» кэш-промахов:**
  - на рис. — оценки для одного из ЦП Intel.

Что еще можно оптимизировать?

МГТУ им. Н.Э. Баумана ТЕХНОПАРК Mail.Ru Group

- **Предсказание переходов:**
  - устранение ветвлений;
  - развертывание (линеаризация) циклов;
  - встраивание функций (методов) и др.
- **Критические секции:**
  - устранение цепочек зависимости для внеочередного исполнения инструкций;
  - использование поразрядных операций, INC (++), DEC (--), векторизация (SIMD) и т.д.
- **Обращение к памяти:**
  - выполнение потоковых операций;
  - выравнивание и упаковка данных и пр.

Оптимизация загрузки кэш-памяти команд: асимметрия условий (1 / 2)

- Если условие в заголовке оператора ветвления часто оказывается **ложным**, выполнение кода становится **нелинейным**. Осуществляемая ЦП предвыборка инструкций ведет к тому, что неиспользуемый код загрязняет кэш-память команд L1i и вызывает проблемы с предсказанием переходов.
- **Ложные предсказания переходов делают условные выражения крайне неэффективными.**
- Асимметричные (статистически смещенные в истинную или ложную сторону) условные выражения становятся причиной ложного предсказания переходов и «пузырей» (периодов ожидания ресурсов) в конвейере инструкций ЦП.
- **Реже исполняемый код должен быть вытеснен с основного вычислительного пути.**

Оптимизация загрузки кэш-памяти команд: асимметрия условий (2 / 2)

- Первый и наиболее очевидный способ повышения эффективности загрузки кэш-памяти L1i — **явная реорганизация блоков**. Так, если условие $P(x)$ чаще оказывается ложным, оператор вида:
  - if(P(x)) statementA; else statementB;

  должен быть преобразован к виду:
  - if(!(P(x))) statementB; else statementA;
- Второй способ решения той же проблемы основан на применении **средств, предоставляемых GCC**:
  - перекомпиляция с учетом результатов профилирования кода;
  - использование функции __builtin_expect.

Функция __builtin_expect (GCC)

- **Функция __builtin_expect** — одна из целого ряда встроенных в GCC функций, предназначенных для целей оптимизации:
  - long __builtin_expect(long *exp*, long *c*);
  - снабжает компилятор информацией, связанной с предсказанием переходов,
  - сообщает компилятору, что наиболее вероятным значением выражения *exp* является *c*, и возвращает *exp*.
- При использовании __builtin_expect с логическими операциями имеет смысл ввести дополнительные макроопределения вида:
  - #define unlikely(expr) __builtin_expect(!!(expr), 0)
  - #define likely(expr) __builtin_expect(!!(expr), 1)

Функция __builtin_expect (GCC): пример


```
#define unlikely(expr) __builtin_expect(!!(expr), 0)
#define likely(expr) __builtin_expect(!!(expr), 1)

int a;

srand(time(NULL));
a = rand() % 10;

if(unlikely(a > 8)) // условие ложно в 80% случаев
    foo();
else
    bar();
```

Оптимизация загрузки кэш-памяти команд: встраивание функций

- **Эффективность** встраивания функций объясняется способностью компилятора одновременно оптимизировать бóльшие кодовые фрагменты. Порождаемый же при этом машинный код способен лучше задействовать конвейерную архитектуру микропроцессора.
- **Обратной стороной** встраивания является увеличение объема кода и бóльшая нагрузка на кэш-память команд всех уровней (L1i, L2i, …), которая может привести к общему снижению производительности.
- Функции, вызываемые однократно, подлежат **обязательному встраиванию**.
- Функции, многократно вызываемые из разных точек программы, **не должны встраиваться** независимо от размера.

GCC-атрибуты always_inline и noinline: пример


```
// пример 1: принудительное встраивание
/* inline */ __attribute__((always_inline)) void foo()
{
    // ...
}

// пример 2: принудительный запрет встраивания
__attribute__((noinline)) void bar()
{
    // ...
}
```

Утилита pfunct. Формат DWARF

- Утилита pfunct позволяет анализировать объектный код на уровне функций, в том числе определять:
  - количество безусловных переходов на метку (goto), параметров и размер функций;
  - **количество функций, определенные как рекомендуемые к встраиванию (inline), но не встроенных компилятором, и наоборот**.
- Работа pfunct (как и описанной ранее утилиты pahole) строится на использовании расположенной в ELF-файле (Executable and Linkage Format) отладочной информации, хранящейся в нем по стандарту DWARF, который среди прочих компиляторов использует GCC:
  - отладочная информация хранится в разделе debug_info в виде иерархии тегов и значений , представляющих переменные, параметры функций и пр. См. также утилиты readelf (binutils) и eu-readelf (elfutils).

Антишаблоны структурного программирования (1 / 2)

МГТУ им. Н.Э. Баумана ТЕХНОПАРК Mail.Ru Group

- **«Загадочный» код** (cryptic code) — выбор малоинформативных, часто однобуквенных идентификаторов.
- **«Жесткий» код** (hard code) — запись конфигурационных параметров как строковых, логических и числовых литералов, затрудняющих настройку и сопровождение системы.
- **Спагетти-код** (spaghetti code) — несоблюдение правил выравнивания, расстановки декоративных пробельных символов, а также превышение порога сложности одной процедуры (функции).
- **Магические числа** (magic numbers) — неготовность определять как символические константы все числовые литералы за исключением, может быть, 0, 1 и -1.

Антишаблоны структурного программирования (2 / 2)

МГТУ им. Н.Э. Баумана ТЕХНОПАРК Mail.Ru Group

- **Применение функций как процедур** (functions as procedures) — неготовность анализировать возвращаемый результат системных и пользовательских функций.
- **«Божественные» функции** (God functions) — функции, берущие на себя ввод данных, вычисления и вывод результатов или иные задачи, каждая из который следует оформить самостоятельно.
- **Неиспользование переносимых типов** — size_t, ptrdiff_t и др.
- **«Утечки» памяти** (memory leaks) и внезапное **завершение процесса** вместо аварийного выхода из функции.
- Использование ветвлений с **условиями, статистически смещенными** не к истинному, а **к ложному результату**.
- **Недостижимый код** (unreachable code).

Системные аспекты выделения и освобождения памяти

- Взаимодействие программы с ОС в плане работы с памятью состоит **в выделении и освобождении участков оперативной памяти разной природы и различной длины**, понимание механизмов которого:
  - упрощает создание и использование структур данных произвольной длины;
  - дает возможность избегать «утечек» оперативной памяти;
  - позволяет разрабатывать высокопроизводительный код.
- **В частности, необходимо знать о существовании 4-частной структуры памяти данных:**
  - область данных, сегмент BSS и куча (входят в сегмент данных);
  - программный стек (не входит в сегмент данных).

Область данных и сегмент BSS

- **Область данных** (data area) — используется для переменных со статической продолжительностью хранения, которые явно получили значение при инициализации:
  - делится на область констант (read-only area) и область чтения-записи (read-write area);
  - инициализируется при загрузке программы, но до входа в функцию main на основании образа соответствующих переменных в объектном файле.
- **Сегмент BSS** (BSS segment, .bss) — предназначен для переменных со статической продолжительностью хранения, не получивших значение при инициализации (инициализированы нулевыми битами):
  - располагается «выше» области данных (занимает более старшие адреса);
  - ОС по требованию загрузчика, но до входа в функцию main способна эффективно заполнять его нулями в блочном режиме (zero-fill-on-demand).

Куча и программный стек

- **Куча** (heap) — контролируется программистом посредством вызова, в том числе, функций malloc / free:
  - располагается «выше» сегмента BSS;
  - является общей для всех разделяемых библиотек и динамически загружаемых объектов (DLO, dynamically loaded objects) процесса.
- **Программный стек** (stack) — содержит значения, передаваемые функции при ее вызове (stack frame), и автоматические переменные:
  - следует дисциплине LIFO;
  - растет «вниз» (противоположно куче);
  - обычно занимает самые верхние (максимальные) адреса виртуального адресного пространства.

Функция malloc

МГТУ им. Н.Э. Баумана ТЕХНОПАРК Mail.Ru Group

- **Внутренняя работа** POSIX-совместимых функций malloc / free **зависит от их реализации** (в частности, дисциплины выделения памяти):
  - одним из самых известных распределителей памяти в ОС семейства UNIX (BSD 4.3 и др.) является распределитель Мак-Кьюсика — Карелса (McKusick–Karels allocator).
- malloc **выделяет для нужд процесса непрерывный фрагмент** оперативной памяти, складывающийся из блока запрошенного объема (адрес которого возвращается как результат функции) и следующего перед ним блока служебной информации, имеющего длину в несколько байт и содержащего, в том числе:
  - размер выделенного блока;
  - ссылку на следующий блок памяти в связанном списке блоков.

Функция free (1 / 2)

МГТУ им. Н.Э. Баумана ТЕХНОПАРК Mail.Ru Group

- free **помечает ранее выделенный фрагмент памяти как свободный**, при этом использует значение своего единственного параметра для доступа к расположенному в начале выделенного фрагмента блоку служебной (дополнительной) информации, поэтому при передаче любого другого адреса поведение free не определено.
- **Контракт программиста с функциями malloc / free состоит в его обязанности передать free тот же адрес, который был получен от malloc.** Вызов free с некорректным в этом смысле параметром способен привести к повреждению логической карты памяти и краху программы.

Функция free (2 / 2)

- В ходе своей работы функция free:
  - идентифицирует выделенный блок памяти;
  - возвращает его в список свободных блоков;
  - предпринимает попытку слияния смежных свободных блоков для снижения фрагментации и повышения вероятности успешного выделения в будущем фрагмента требуемого размера.
- Такая логика работы пары malloc / free избавляет от необходимости передачи длины освобождаемого блока как самостоятельного параметра.

Стандарт POSIX

- **POSIX** (Portable Operating System Interface for UNIX) — набор стандартов, разработанных IEEE и Open Group для обеспечения совместимости ОС через унификацию их интерфейсов с прикладными программами (API), а также переносимость самих прикладных программ на уровне исходного кода на языке C.
- Стандарт IEEE 1003 содержит четыре основных компонента:
  - «Основные определения» (том XBD);
  - «Системные интерфейсы» (том XSH) — в числе прочего описывает работу с сигналами, потоками исполнения (threads), потоками В/В (I/O streams), сокетами и пр., а также содержит информацию обо всех POSIX-совместимых системных подпрограммах и функциях;
  - «Оболочка и утилиты» (том XCU);
  - «Обоснование» (том XRAT).
- POSIX-совместимыми ОС являются IBM AIX, OpenSolaris, QNX и др.

Потоки POSIX или процессы UNIX?

- **Потоки POSIX** (POSIX threads, Pthreads) впервые введены стандартом POSIX 1003.1c-1995 и, с теоретической точки зрения, являются «легковесными» процессами ОС UNIX, которые в настоящее время поддерживаются во всех ОС семейства (Linux, AIX, HP-UX и пр.), а также в системах Microsoft Windows.
  - Поток — единственная **разновидность программного контекста**, включающая в себя необходимые для исполнения кода аппаратные «переменные состояния»: регистры, программный счетчик, указатель на стек и т.д. При этом потоки компактнее, быстрее и более адаптивны, чем традиционный процесс.
- В модели с потоками процесс можно воспринимать как **данные** (адресное пространство, дескрипторы файлов и т.п.) **вкупе с** одним или несколькими **потоками**, разделяющими его адресное пространство.

Асинхронное программирование как парадигма

МГТУ им. Н.Э. Баумана ТЕХНОПАРК Mail.Ru Group

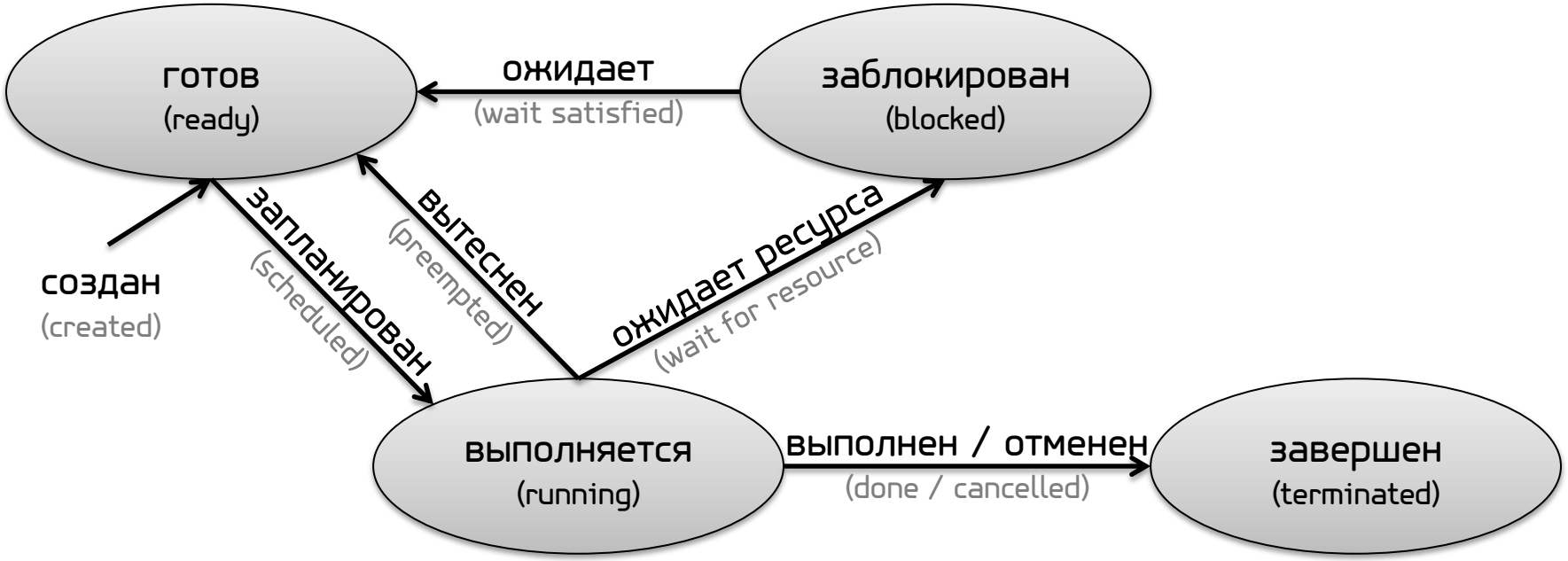
- Техника асинхронного программирования многопоточных приложений **отличается от разработки** (синхронных) однопоточных систем, что позволяет говорить о смене парадигмы разработки как таковой.
- **Любые две операции являются «асинхронными», если в отсутствие явно выраженной зависимости они могут быть выполнены независимо друг от друга** (одновременно, с произвольным чередованием).
- Асинхронные приложения **по-разному выполняются** на системах с одним и несколькими доступными программисту вычислительными узлами.
- Асинхронное программирование ставит вопросы **потоковой безопасности и реентерабельности** программного кода.

Элементы многопоточного программирования

- Поддержка многопоточного программирования со стороны ОС предполагает три важнейших аспекта: контекст исполнения, механизмы диспетчеризации и синхронизации.
- С архитектурных позиций, поддержка потоков POSIX заключается:
  - в предоставлении программисту ряда специфических переносимых типов: pthread_t, pthread_attr_t, pthread_mutex_t, pthread_cond_t и др.;
  - в предоставлении набора функций создания, отсоединения, завершения потоков, блокировки мьютексов и т.д.;
  - в механизме проверки наличия ошибок без использования errno.
- Каждый поток POSIX имеет свою начальную функцию, аналогичную функции main процесса:
  - void *my_thread_main(void *arg) { /* ... */ }

Создание и жизненный цикл потока

- Среди потоков процесса особняком стоит «исходный поток», создаваемый при создании процесса. Дополнительные потоки создаются явным обращением к pthread_create, получением POSIX-сигнала и т.д.



Примечание: Синхронизации возврата потока-создателя из pthread_create и планирования нового потока в рассматриваемой модели не предусмотрено.

Запуск исходного и дополнительных потоков

- Выполнение потока начинается с входа в его начальную функцию, вызываемую с единственным параметром типа void*.
  - Значение параметра начальной функции потока передается ей через pthread_create и может быть равно NULL.
- Функция main де-факто является начальной функцией исходного потока и в большинстве случаев вызывается средствами прилинкованного файла crt0.o, который инициализирует процесс и передает управление главной функции:
  - параметром main является массив аргументов (argc, argv), а не значение типа void*; тип результата main — int, а не void*;
  - возврат из main в исходном потоке немедленно приводит к завершению процесса как такового;
  - для продолжения выполнения процесса после завершения main необходимо использовать pthread_exit, а не return.

Выполнение и блокировка потока

МГТУ им. Н.Э. Баумана ТЕХНОПАРК Mail.Ru Group

- В общем случае выполнение потока может быть **приостановлено**:
  - если поток нуждается в недоступном ему ресурсе, то он блокируется;
  - если поток снимается с исполнения (напр., по таймеру), то он вытесняется.
- В связи с этим большая часть жизненного цикла потока связана с переходом **между тремя состояниями**:
  - **готов** — поток создан и не заблокирован, а потому пригоден для выполнения (ожидает выделения процессора);
  - **выполняется** — поток готов, и ему выделен процессор для выполнения;
  - **заблокирован** — поток ожидает условную переменную либо пытается захватить запертый мьютекс или выполнить операцию ввода-вывода, которую нельзя немедленно завершить, и т.д.

Завершение потока

- Стандартными **способами завершения** потоков являются:
  - штатный возврат из начальной функции (return);
  - штатный вызов pthread_exit завершающимся потоком;
  - вызов pthread_cancel другим потоком (результат: PTHREAD_CANCELLED).
- Если завершающийся поток был «отсоединен» (detached), он сразу уничтожается. Иначе поток остается в состоянии «завершен» и доступен для объединения с другими потоками. В ряде источников такой поток носит название «зомби».
  - Завершенный процесс сохраняет в памяти свой идентификатор и значение результата, переданное return или pthread_exit.
  - Поток-«зомби» способен удерживать (почти) все ресурсы, которые он использовал при своем выполнении.
- Во избежание возникновения потоков-«зомби» потоки, не предполагающие объединения, должны всегда отсоединяться.

Уничтожение потока

МГТУ им. Н.Э. Баумана ТЕХНОПАРК Mail.Ru Group

- Поток уничтожается по окончании выполнения:
  - если он был отсоединен самим собой или другим потоком по время своего выполнения;
  - если он был создан отсоединенным (PTHREAD_CREATE_DETACHED).
- Поток уничтожается после пребывания в состоянии «завершен»:
  - после отсоединения (pthread_detach);
  - после объединения (pthread_join).
- Уничтожение потока высвобождает ресурсы системы или процесса, которые не были освобождены при переходе потока в состояние «завершен», в том числе:
  - место хранения значения результата;
  - стек, память с содержимым регистров ЦП и др.

Критические секции и инварианты

МГТУ им. Н.Э. Баумана ТЕХНОПАРК Mail.Ru Group

- **Инвариант** — постулированное в коде предположение, в большинстве случаев — о связи между данными (наборами переменных) или их состоянии. Формулировка инварианта как логического выражения позволяет смотреть на него как на **предикат**.
  - Инварианты **могут нарушаться** при исполнении **изолированных** частей кода, по окончании которых **должны быть восстановлены** со 100% гарантией.
- **Критическая секция** — участок кода, который производит логически связанные манипуляции с разделяемыми данными и влияет на общее состояние системы.
  - Если два потока обращаются к разным разделяемым данным, оснований для возникновений ошибок нет. Поэтому говорят о критических секциях «по переменной x» или «по файлу f».

Взаимные исключения и ситуация гонок

- Организация работы потоков, при которой два (и более) из них не могут одновременно пребывать в критических секциях по одним данным, носит название **взаимного исключения**.
- При одновременном доступе нескольких процессов к разделяемым данным могут возникать **проблемы, связанные с очередностью действий.**
- Ситуация, в которой результат зависит от последовательности событий в независимых потоках (или процессах), называется **гонками (состязанием)**.
  - Нежелательные эффекты в случае гонок возможны, в том числе, только при чтении данных.
- В ОС UNIX обеспечение взаимных исключений строится на кратковременном запрете прерываний и возлагается на ядро ОС. Блокировать процесс (поток) может только ОС.

Мьютексы (1 / 2)

МГТУ им. Н.Э. Баумана ТЕХНОПАРК Mail.Ru Group

- В общем случае под **мьютексом** (mutex) понимается объект с двумя состояниями (открыт/заперт) и двумя атомарными операциями:
  - **операция «открыть»** всегда проходит успешно и незамедлительно возвращает управление, переводя мьютекс в состояние «открыт»;
  - **операция «закрыть»** (неблокирующий вариант) может быть реализована как булева функция, незамедлительно возвращающая «истину» для открытого мьютекса (который при этом ей закрывается) или «ложь» для закрытого мьютекса;
  - **операция «закрыть»** (блокирующий вариант) может быть реализована как процедура, которая закрывает открытый мьютекс (и незамедлительно возвращает управление) или блокирует поток до момента отпирания закрытого мьютекса, после чего закрывает его и возвращает управление.

Мьютексы (2 / 2)

- В стандарте Pthreads мьютексы, задача которых — сохранить целостность асинхронной системы, реализованы как **переменные в пользовательском потоке**, ядро ОС поддерживает лишь операции над ними.
- Мьютексы могут создаваться как статически, так и динамически. После инициализации мьютекс всегда открыт:
  - pthread_mutex_t my_mutex = PTHREAD_MUTEX_INITIALIZER;
- Потоки POSIX поддерживают как блокирующий, так и неблокирующий вариант закрытия мьютексов.
- Неиспользуемый мьютекс может быть уничтожен. На момент уничтожения мьютекс должен быть открыт.
- Обобщением мьютекса является **семафор Дейкстры**, реализованный в Pthreads как объект типа sem_t.

Условные переменные

- Механизм **условных переменных** (condition variable) позволяет организовать нетривиальный протокол взаимодействия потоков, блокируя их выполнение до наступления заданного события (выполнения предиката).
- Использование условных переменных предполагает:
  - создание и уничтожение переменных;
  - ожидание выполнения условия (с возможностью указать абсолютное астрономическое время снятия блокировки);
  - передачу сигналов, в том числе в широковещательном режиме.
- Переносимым типом условной переменной в Pthreads является pthread_cond_t.

Практикум №2

МГТУ им. Н.Э. Баумана ТЕХНОПАРК Mail.Ru Group

Постановка задачи

- Решить индивидуальные задачи №№3, 4 и 5 в соответствии с формальными требованиями.
- Для этого в блоге дисциплины:
  - узнать постановку задач.

Спасибо за внимание

Алексей Петров

Приложение

Прил. 1.
Задача об умножении матриц (1 / 2)

МГТУ им. Н.Э. Баумана ТЕХНОПАРК Mail.Ru Group

- Классическим примером задачи, требующей неэффективного, с точки зрения архитектуры ЭВМ, обхода массива по столбцам, является задача об умножении матриц: $(AB)_{ij} = \sum_{k=0}^{N-1} a_{ik} b_{kj}$ — имеющая следующее очевидное решение:

```
for(i = 0; i < N; i++)
    for(j = 0; j < N; j++)
        for(k = 0; k < N; k++)
            res[i][j] +=
                mul1[i][k] * mul2[k][j];
```

Прил. 1.
Задача об умножении матриц (2 / 2)

- Предварительное транспонирование второй матрицы повышает эффективность решения в 4 раза (с 16,77 млн. до 3,92 млн. циклов ЦП Intel Core 2 с внутренней частотой 2666МГц).
- Математически: $(AB)_{ij} = \sum_{k=0}^{N-1} a_{ik} b_{jk}^{T}$.
- На языке C:

```
double tmp[N][N];
for(i = 0; i < N; i++)
    for(j = 0; j < N; j++)
        tmp[i][j] = mul2[j][i];
for(i = 0; i < N; i++)
    for(j = 0; j < N; j++)
        for(k = 0; k < N; k++)
            res[i][j] += mul1[i][k] * tmp[j][k];
```

Прил. 2. Эффективный обход двумерных массивов: эффективность оптимизации