Углубленное программирование на языке C / C++

Лекция № 2

Алексей Петров

Лекция №2. Организация и использование СОЗУ. Основы многопоточного программирования. Вопросы качества 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 ИЗ к практикуму №2.**

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 90% 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()
{
    // ...
}
```

Антишаблоны структурного программирования (1 / 2)

- **«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)

- **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).

Антишаблоны в примерах: использование continue (1 / 2)


```
// не рекомендуется
// см. SonarQube C Quality Profile [Sonar Way]
// MINOR: ‘continue’ should not be used
int main(int argc, char *argv[]) {
    int i;
    for(i = 0; i < 10; i++) {
        if(i == 5) {
            continue;          /* Non-Compliant */
        }
        printf(“i = %d\n”, i);
    }
    return -1;
}
```

Антишаблоны в примерах: использование continue (2 / 2)


```
// рекомендуется
// см. SonarQube C Quality Profile [Sonar Way]
// MINOR: ‘continue’ should not be used
int main(int argc, char *argv[]) {
    int i;
    for(i = 0; i < 10; i++) {
        if(i != 5) {          /* Compliant */
            printf(“i = %d\n”, i);
        }
    }
    return -1;
}
```

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 статических переменных, не получивших значение при инициализации (инициализированы нулевыми битами):
  - располагается «выше» области данных (занимает более старшие адреса);
  - по требованию загрузчика ОС до входа в main() может эффективно заполнить BSS 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

- **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)

- 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 (PC), указатель на стек (SP) и т.д. Потоки компактнее, быстрее и более адаптивны, чем процесс UNIX, однако, являются не единственным способом **асинхронной организации вычислений**.
- 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

- 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 (in parallel) или с произвольным чередованием (concurrently)].
- Асинхронные приложения **по-разному выполняются** на системах с одним и несколькими доступными программисту вычислительными узлами (uniprocessor vs. multiprocessor).
- Асинхронное программирование опирается на понимание разработчиком основ диспетчеризации, синхронизации и параллелизма, а также ставит перед ним вопросы **потоковой безопасности и реентерабельности** программного кода.

Элементы многопоточного программирования

- Поддержка многопоточного программирования со стороны ОС предполагает три важнейших аспекта: **контекст исполнения**, механизмы **диспетчеризации** (планирования исполнения и переключения контекстов) и **синхронизации** (координации совместного использования контекстами их общих ресурсов).
- С архитектурных позиций, поддержка потоков POSIX состоит:
  - в предоставлении программисту ряда специфических «неясных» (opaque) 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.

Элементы многопоточного программирования: пример (1 / 2)


```
#include <pthread.h>
void *thread_routine(void *arg)
{
    int errflag;
    // ...
    // отсоединить «себя» как поток POSIX до завершения
    errflag = pthread_detach(pthread_self());
    // проверить, удался ли вызов pthread_detach()
    if(errflag != 0)
        // ...
    // штатно завершить поток с возвратом значения void*
    return arg;        // вариант: return NULL и пр.
}
```

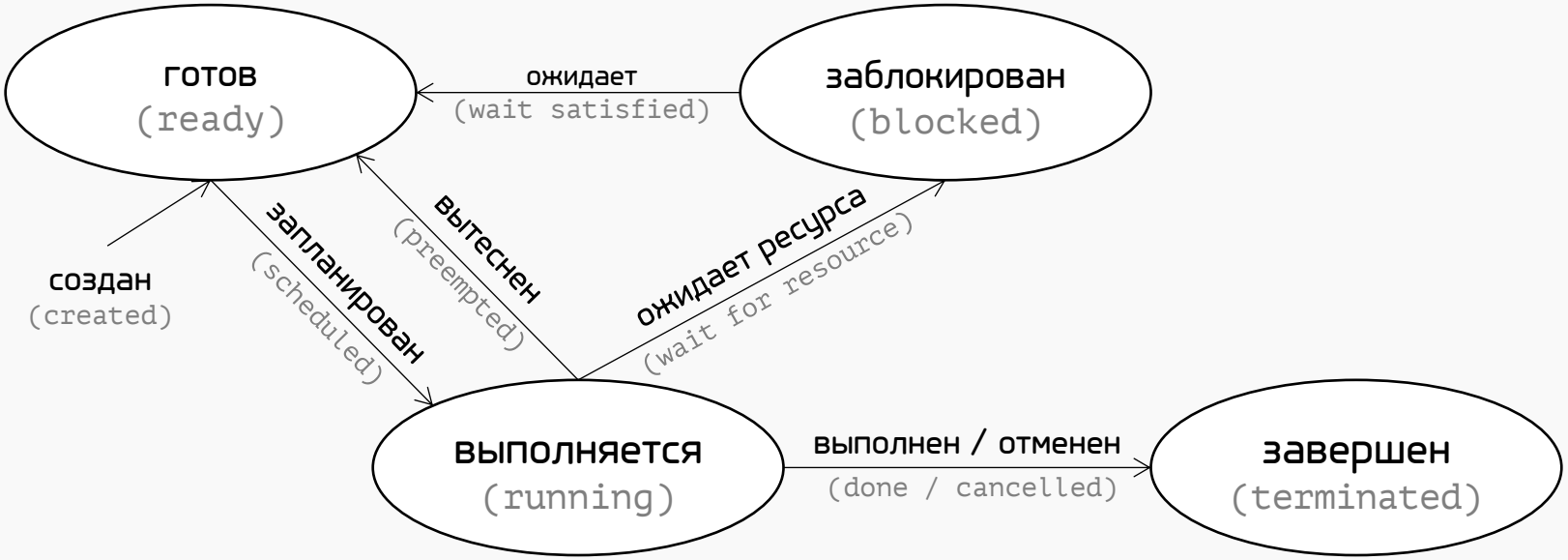
Элементы многопоточного программирования: пример (2 / 2)


```
int main(int argc, char *argv[])
{
    int         errflag;
    pthread_t   thread;         // идентификатор потока

    // создать и запустить на исполнение поток POSIX
    errflag = pthread_create(
        &thread, NULL, thread_routine, NULL);
    // проверить, удался ли вызов pthread_create()
    if(errflag != 0)
        // ...
    // штатно завершить гл. поток и связанный с ним процесс
    return EXIT_SUCCESS;
}
```

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

- 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 (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

- 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

- **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, как правило, 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)

- 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»** (условный, или неблокирующий вариант) может быть реализована как булева функция, незамедлительно возвращающая «истину» для открытого мьютекса (который при этом ей закрывается) или «ложь» для закрытого мьютекса (EBUSY);
  - **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 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.

Использование мьютекса: пример (1 / 2)


```
#include <pthread.h>
typedef struct {
    pthread_mutex_t mutex;    // мьютекс для защиты значения
    int             value;    // защищаемое значение
} data_t;

// разделяемый объект; мьютекс инициализируется статически
data_t              data = {PTHREAD_MUTEX_INITIALIZER, 1};

void *thread_routine(void *arg)
{
    pthread_mutex_t    *mutex = &data.mutex;
    int                errflag;
```

Использование мьютекса: пример (2 / 2)


```
    // закрыть (блокировать) мьютекс до изменения данных
    errflag = pthread_mutex_lock(mutex);
    // проверить, удался ли вызов pthread_mutex_lock()
    if(errflag != 0)  // ...

    data.value *= 2;  // изменить данные

    // открыть (отпереть) мьютекс после изменения данных
    errflag = pthread_mutex_unlock(mutex);
    // проверить, удался ли вызов pthread_mutex_unlock()
    if(errflag != 0)  // ...
    return arg;       // вариант: return NULL и пр.
}
```

Использование нескольких мьютексов: стратегии блокировок (1 / 2)

- По архитектурным соображениям одного мьютекса может быть недостаточно (напр., мьютекс A может защищать «голову» списка, а мьютекс B — текущий обрабатываемый элемент), при этом существует риск возникновения взаимных блокировок и прочих проблем синхронизации потоков.
- Защита двумя и более взаимно независимых данных является тривиальной, необходимость же одновременного закрытия двух и более мьютексов по взаимно зависимым (связанным) данным в разных потоках может повлечь проблемы.

Использование нескольких мьютексов: стратегии блокировок (1 / 2)

- Успешными стратегиями блокировок по двум и более мьютексам являются:
  - **фиксированная иерархия блокировок** — установление единого порядка закрытия и открытия мьютексов в каждом потоке;
  - **«попытаться и откатить»** — блокирующее закрытие первого мьютекса из набора с последующим неблокирующим (напр., pthread_mutex_trylock()) вызовом закрытия для оставшихся и «аварийным» открытием всех мьютексов в случае неудачи.

Критерий	Фиксированная иерархия	«Попытаться и откатить»
Эффективность	**Выше**	Ниже
Гибкость	Ниже	**Выше**
Строгость протокола закрытия	Выше	Ниже

Условные переменные

- Механизм **условных переменных** (condition variable) позволяет организовывать практически любые протоколы взаимодействия потоков POSIX, блокируя их выполнение до наступления заданного события (выполнения предиката) или момента времени.
- Использование условных переменных предполагает:
  - статическую инициализацию или динамическое создание и уничтожение условных переменных;
  - ожидание выполнения условия (с возможностью указать абсолютное астрономическое время снятия блокировки — блокировка с тайм-аутом);
  - широковещательное оповещение и (или) передачу сигналов (не смешивать с сигналами UNIX!).
- Переносимым типом условной переменной в Pthreads является pthread_cond_t.

Использование условных переменных: пример (1 / 4)


```
#include <pthread.h>
typedef struct {
    pthread_mutex_t mutex;    // мьютекс для защиты значения
    pthread_cond_t  cond;     // усл. перем. для коммуник.
    int             value;    // защищаемое значение
} data_t;

// разделяемый объект; мьютекс и условная переменная
// инициализируются статически
data_t              data =    {PTHREAD_MUTEX_INITIALIZER,
                               PTHREAD_COND_INITIALIZER, 0};
```

Использование условных переменных: пример (2 / 4)


```
void *thread_routine(void *arg)
{
    int                 errflag;
    errflag = pthread_mutex_lock(&data.mutex);
    if(errflag != 0)   // ...

    data.value = 1;    // изменить данные
    // послать сигнал об изменении разделяемых данных
    errflag = pthread_cond_signal(&data.cond);
    if(errflag != 0)   // ...
    errflag = pthread_mutex_unlock(&data.mutex);
    if(errflag != 0)   // ...
    return arg;        // вариант: return NULL и пр.
}
```

Использование условных переменных: пример (3 / 4)


```
int main(int argc, char *argv[])
{
    int                 errflag;
    struct timespec     timeout;

    // ...
    timeout.tv_sec = time(NULL) + 1;
    timeout.tv_nsec = 0;

    // БЛОКИРОВКА ПОТОКА НА УСЛОВНОЙ ПЕРЕМЕННОЙ
    // ТРЕБУЕТ ПРЕДВАРИТЕЛЬНОГО ЗАКРЫТИЯ МЬЮТЕКСА
    errflag = pthread_mutex_lock(&data.mutex);
    if(errflag != 0)   // ...
```

Использование условных переменных: пример (4 / 4)


```
// int main(int argc, char *argv[])
    while(data.value == 0) {  // 1-я проверка предиката
        errflag = pthread_cond_timedwait(
                        &data.cond, &data.mutex, &timeout);
        if(errflag == ETIMEDOUT)
            break;     // завершение блокировки по тайм-ауту
        else // ошибка при вызове pthread_cond_timedwait()
    }
    if(data.value != 0) ...   // 2-я проверка предиката

    errflag = pthread_mutex_unlock(&data.mutex);
    if(errflag != 0) // ...
    return EXIT_SUCCESS;
}
```

Закон Дж. Амдала (1 / 2)

- Фундаментальное ограничение на рост производительности системы с увеличением количества вычислительных узлов сформулировано Дж. Амдалом (англ. Gene Amdahl), установившим, что **ускорение выполнения кода за счет распараллеливания ограничено временем, затрачиваемым на последовательные действия**.

При подготовке сл. 34 – 36 использованы материалы тренинга А.В. Петрова «Проектирование высокопроизводительных систем» (2014).

Закон Дж. Амдала (2 / 2)

- **Закон Амдала** (Amdahl's Law, 1967). Если задача разделяется на несколько частей, суммарное время ее выполнения на параллельной системе не может быть меньше времени выполнения самого длинного фрагмента. Формально:

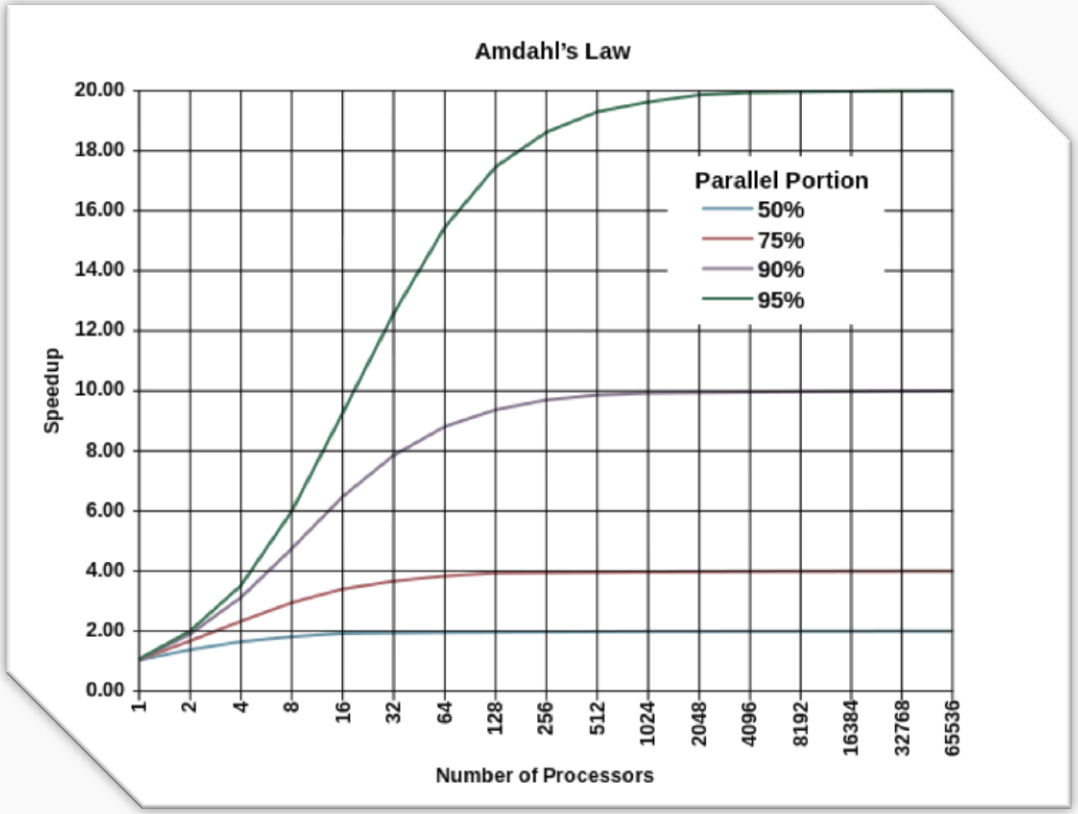
$$S_p = \frac{1}{\alpha + \frac{1-\alpha}{p}}$$

где α — доля вычислений, которая может быть получена только последовательными расчетами;

p — количество параллельных узлов (потоков), или разрешенных к использованию процессоров;

S_p — ускорение системы в сравнении с 1-процессорной. ■

Закон Дж. Амдала: пример



На рис. представлены кривые сравнительного ускорения S_p программы при помощи параллельных вычислений при $\alpha = 0{,}05$. Заметим, $\lim\limits_{p\to\infty} S_p = 20$.

Преимущества многопоточного программирования

- Наряду с выгодой от эффективного использования аппаратного («истинного») параллелизма, многопоточное программирование:
  - стимулирует к поддержанию **оптимальной модульной структуры** исходного кода;
  - способствует **ясному отражению зависимостей** между частями программы на уровне исходного кода, а не комментариев в нем;
  - содействует **«здоровой» изоляции** независимых или слабо связанных между собой вычислительных путей (потоков), явным образом (через мьютексы, семафоры, условные переменные, очереди сообщений и соответствующие программные вызовы и конструкции языка) синхронизируемых только по мере реальной необходимости;
  - повышает **удобство сопровождения и развития** кодовой базы.

Практикум №2

Постановка задачи

- Решить индивидуальную задачу №2 в соответствии с формальными требованиями.
- Для этого:
  - авторизоваться в АСТС и узнать в ней постановку задачи.

Алексей Петров

Спасибо за внимание!

Приложения

Приложение А.
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 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).