



Углубленное программирование на языке С / С++



Лекция № 2

Отметьтесь на портале



- Посещение необязательное, но тем, кто пришёл, следует отмечаться на портале в начале каждого занятия
- Это позволяет нам анализировать, какие занятия были более или менее интересны студентам, и менять курс в лучшую сторону
- Также это даст возможность вам оставить обратную связь по занятию после его завершения

| пн, 7 октября | вт, 8 октября | ср, 9 октября | чт, 10 октября | пт, 11 октября | сб, 12 октября | вс, 13 октября |
|---------------|---|---------------|---|----------------|----------------|----------------|
| Занятий нет | 12:00 Офис Mail.Ru Gro Разработка выпускног... | Занятий нет | 18:35 213 ЛК Информационная безоп... | Занятий нет | Занятий нет | Занятий нет |

Расписание занятий с 30 сентября по 5 октября

Расписание занятий

Привет, друзья!

Предлагаю вам расписание занятий на эту неделю.

Начало всех занятий в **18:35** за исключением занятий выделенных в расписании отдельно.

1 семестр

30 сентября(понедельник) Frontend разработка (аудитория 430 ГК)
3 октября(четверг) Backend разработка (аудитория 430 ГК)

3 семестр

2 октября(среда) Разработка выпускного проекта (Офис Mail.ru)
3 октября(четверг) Информационная безопасность(аудитория 213 ЛК)

Семестровые курсы:

30 сентября(понедельник) Введение в промышленное программирование и структуры данных, Язык C++(113 ГК)

1 октября(вторник) Основы языка Python (113 ГК)

2 октября(среда) Основы языка Java (113 ГК)

3 октября(четверг) Основы языка C# (113 ГК)

4 октября(пятница) Основы языка C (113 ГК)

5 октября(суббота) Основы языка C++ (113 ГК)

Текущее занятие

Сейчас проводится занятие - **Разработка выпускного проекта Рубежный контроль 3.**

Оно проходит в аудитории **Офис Mail.Ru Group**.

- Отметьтесь, что вы пришли на занятие.
Так вы улучшите свою посещаемость и вас увидят преподаватель в своём "Журнале посещений".

- Оставьте отзыв о занятии и мы сможем улучшить учебный процесс.

Лекция №2. Оптимизация работы с оперативной и сверхоперативной памятью на языке С. Внешние библиотеки и линковщик. Основы системного программирования.



1. Вопросы выравнивания структур данных в памяти
2. Проблема упаковки переменных составных типов
3. Оптимизация работы с кэш-памятью ЦП ЭВМ
4. Работа с файлами и потоками ввода/вывода
5. Внешние библиотеки и линковщик
6. Основы многопроцессного программирования
7. Многопоточное программирование с использованием потоков POSIX
8. Постановка ИЗ к практикуму №2

Выравнивание объектов, размещаемых статически. GCC-атрибут `aligned` (1 / 2)



- Одним из способов повышения производительности программы является такое размещение данных в ОЗУ, при котором они эффективно загружаются в кэш-память ЦП. Для этого данные должны быть, как минимум, **выровнены на границу линии кэш-памяти данных 1-го уровня (L1d)**. Выравнивание объекта в ОЗУ обычно определяется характеристиками выравнивания, которые имеет соответствующий тип данных. При этом:
 - выравнивание **скалярного объекта** определяется собственной характеристикой выравнивания приписанного ему базового типа;
 - выравнивание **массива**, — если размер каждого элемента не кратен величине выравнивания, — распространяет свое действие только на элемент с индексом 0;
 - выравнивание объектов в программе на языке С может регулироваться **на уровне отдельных переменных и типов данных.**, для чего в компиляторе GCC служит атрибут `aligned`.

Выравнивание объектов, размещаемых статически. GCC-атрибут `aligned` (2 / 2)



- Выравнивание статически размещаемых переменных имеет силу как для глобальных, так и для автоматических переменных. При этом характеристика выравнивания, присущая типу объекта, полностью игнорируется.
- При выравнивании массивов гарантированно выравнивается только начальный, нулевой элемент массива.



Выравнивание объектов, размещаемых статически. Атрибут `aligned`: пример



```
// выравнивание, регулируемое на уровне объекта
// переменная qwd выравнивается на границу 64 байт
uint64_t qwd __attribute__((aligned(64)));


// выравнивание, регулируемое на уровне типа
// переменные типа al128int_t (сионим int)
// выравниваются на границу 128 байт
typedef int __attribute__((aligned(128))) al128int_t;
al128int_t aln;
```

Выравнивание объектов, размещаемых динамически. Функция `posix_memalign`



- Функция `posix_memalign`:
 - определена в стандарте POSIX 1003.1d;
 - имеет прототип

```
int posix_memalign(void **memptr, size_t alignment, size_t size);
```
 - выделяет неиспользуемый участок памяти размера `size` байт, выровненный на границу `alignment`, и возвращает указатель на него в `memptr`;
 - допускает освобождение выделенной памяти функцией `free()`.
- Требования к значению `alignment`:
 - кратно `sizeof(void*)`;
 - является целочисленной степенью числа 2.
- Ошибки (`EINVAL`, `ENOMEM`):
 - значение `alignment` не является кратной `sizeof(void*)` степенью 2;
 - недостаточно памяти.



posix_memalign: пример (1 / 2)



```
int b[7] = {1, [5] = 10, 20, [1] = 2}; // массив-источник
int *p = NULL,                                // массив-приемник
     errflag;                                 // код ошибки posix_memalign

// установить размер линии кэш-памяти данных 1-го уровня
// (L1d); типичное значение: 64 байта
long l1dcls = sysconf(_SC_LEVEL1_DCACHE_LINESIZE);
// проверить, удался ли вызов sysconf()
if (l1dcls == -1)
// если вызов sysconf() неудачен, использовать значение
// выравнивания по умолчанию
    l1dcls = sizeof(void*);
```



posix_memalign: пример (2 / 2)



```
// выделить память с выравниванием на границу строки L1d
errflag = posix_memalign((void**)&p, l1dcls, sizeof b);
if(!errflag)// в случае успеха posix_memalign возвращает 0
{
    printf("\nL1d cache line size is %ld\n", l1dcls);
    printf("p and &p are %p and %p\n", p, &p);
    p = memcpuy(p, b, sizeof(b));
    // ...
    free(p);
}
else
    printf("posix_memalign error: %d\n", errflag);
```

Упаковка переменных составных типов (1 / 2)



- Для структур данных актуален также не характерный для массивов и скаляров вопрос **упаковки данных**, обусловленный наличием у элементов структур индивидуальных характеристик выравнивания.
- **Проблема упаковки структур** заключается в том, что смежные (перечисленные подряд) элементы часто физически не «примыкают» друг к другу в памяти.

Упаковка переменных составных типов (2 / 2)



- Например (для x86):

```
typedef struct {  
    int          id;           // 4 байта  
    char         name[15];     // 15 байт  
    double       amount;       // 8 байт  
    _Bool        active;      // 1 байт  
} account;                // 28 байт (не 32 байта!)
```

- Наличие лакун, аналогичных выявленным 4-байтовым (13%) потерям в структуре `account`, вызывается совокупностью факторов:
 - архитектура процессора (напр., x86 или x86-64);
 - оптимизирующие действия компилятора;
 - выбранный программистом порядок следования элементов.

Реорганизация структур данных: рекомендации



- Реорганизация структур для повышения эффективности использования кэш-памяти должна идти **по 2 направлениям**:
 - декомпозиция тяжеловесных («божественных») структур на более мелкие, узкоспециализированные структуры, которые при решении конкретной задачи используются полностью либо не используются вообще;
 - устранение в структурах лакун, обусловленных характеристиками выравнивания типов их элементов ([см. ранее](#)).
- **При прочих равных условиях крайне желательно:**
 - переносить наиболее востребованные элементы структуры к ее началу ([при загрузке в кэш-память такие элементы структуры могут становиться «критическими словами», доступ к которым должен быть самым быстрым](#));
 - обходить структуру в порядке определения элементов, если иное не требуется задачей или прочими обстоятельствами.



Реорганизация структур данных: рекомендации: пример



```
typedef struct { // вар. 1: 28/32 байт (x86: 13% потерь)
    int          id;           // 4 байта
    char         name[15];     // 15 байт
    /* лакуна – 1 байт выравнивания */
    double       amount;       // 8 байт
    _Bool        active;       // 1 байт
    /* лакуна – 3 байта выравнивания */
} account_1;                                // 32 байта

typedef struct { // вар. 2: 28/28 байт (x86: 0% потерь)
    int          id;           // 4 байта
    char         name[15];     // 15 байт
    _Bool        active;       // 1 байт
    double       amount;       // 8 байт
} account_2;                                // 28 байт
```



Расположение в памяти структуры с вложенной структурой



```
struct result {                                // 64-битная ОС
    char id;
    struct detailed_result {
        char *name;
        short score;
    } details;
};
```



Расположение в памяти структуры с вложенной структурой



```
struct result {                                // 64-битная ОС
    char id;                                  // 1 байт
    /* Лакуна в 7 байт из-за выравнивания по char* */
    struct detailed_result {
        char *name;                            // 8 байт
        short score;                           // 2 байт
        /* Лакуна в 6 байт из-за выравнивания по char* */
    } details;                               // 16 байт
};                                         // 11/24 байт (54% потерь)
```



Packed vs unpacked-структуры



```
struct unpacked {  
    char a; // 4 байта  
    int i; // 4 байта  
} *s1;
```

```
const int N = 10;  
  
char data[N];  
data[0] = 'a';  
data[1] = 'b';  
for (int i = 2; i < N; ++i) { data[i] = 'c'; }  
s1 = (struct unpacked *)data;
```

```
s2 = (struct packed *)data;  
  
printf("%c, %c\n",  
       s1->a, s1->i);  
// Выведет ac
```

```
struct packed {  
    char a; // 1 байт  
    int i; // 4 байта  
} attribute ((packed)) *s2;
```

```
printf("%c, %c\n",  
       s2->a, s2->i);  
// Выведет ab
```

#pragma pack (1 / 2)



- Используется для сброса значений выравнивания до 1 байта локально для конкретной структуры, расположенной сразу после директивы
- В GCC:
 - #pragma pack(*n*) – явно переопределяет текущее значение выравнивания, не действуя стек
 - #pragma pack() – сбрасывает значение на значение по умолчанию, предоставляемое компилятором
 - #pragma pack(push[*n*]) – сохраняет текущее значение выравнивания во внутренний стек. Параметр *n* необязательный и позволяет задать новое значение выравнивания
 - #pragma pack(pop) – извлекает последнее значение из стека



#pragma mark (2 / 2) - пример



```
#pragma pack(push, 1)
struct command {
    char op_code;
    double value;
    short addr;
};

#pragma pack(pop) // восстанавливает предыдущее значение
                  // выравнивания
```



Расположение в памяти структур с битовыми полями



```
typedef struct {  
    short      address;  
    char       code;  
    int        header:1;  
    int        message:4;  
    int        checksum:7;  
} letter;
```



Расположение в памяти структур с битовыми полями



```
typedef struct {  
    short      address;      // 2 байта  
    char       code;         // 1 байт  
    int        header:1;     // 1 бит  
    int        message:4;    // 4 бит, всего 5 бит  
    /* лакуна в 4 бита до 2 байт */  
    int        checksum:7;   // 7 бит, всего 12 бит  
    /* лакуна в 1 байт из-за выравнивания по short*/  
} letter;                  // 5/6 байт (16.7% потерь)
```



Расположение в памяти структур с битовыми полями в зависимости от разрядности системы



```
typedef struct {  
    int bitfield1:31;  
    int bitfield2:31;  
    int bitfield3:1;  
    int bitfield4:1;  
} letter;
```

```
// 32-битная ОС - ?  
// 64-битная ОС - ?
```



Расположение в памяти структур с битовыми полями в зависимости от разрядности системы



```
typedef struct {  
    int bitfield1:31;  
    int bitfield2:31;  
    int bitfield3:1;  
    int bitfield4:1;  
} letter;  
  
// 32-битная ОС - 3 слова  
// 64-битная ОС - 1 слово
```

Реорганизация структур данных: недостатки и альтернатива решения



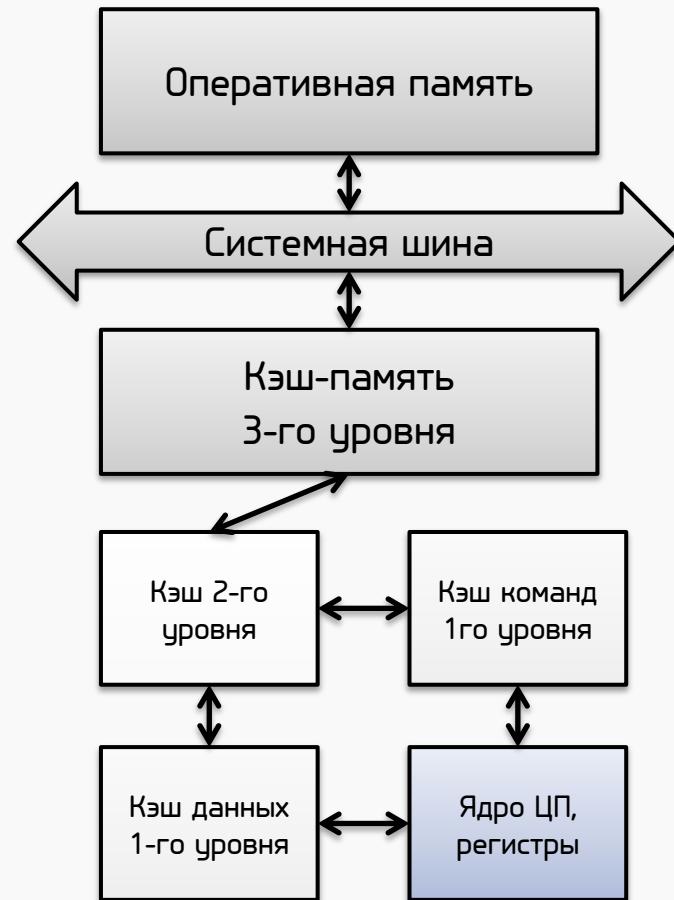
- Недостатки реорганизации:
 - снижение удобства чтения и сопровождения исходного кода;
 - риск размещения совместно используемых элементов (напр., длины вектора и адреса его начального элемента) на разных линиях кэш-памяти.
- Основная альтернатива реорганизации — замена стихийно выбранных типов данных наиболее адекватными по размеру, вплоть до использования битовых полей данных.

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 следующих слайдов и Прил. Д использованы материалы доклада А.В. Петрова 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:

| | | | | | |
|--|--|--|--|--|--|
| | | | | | |
| | | | | | |
| | | | | | |
| | | | | | |
| | | | | | |

$$A = (a_{ij})$$

Найти:

| | | | | |
|--|--|--|--|--|
| | | | | |
| | | | | |

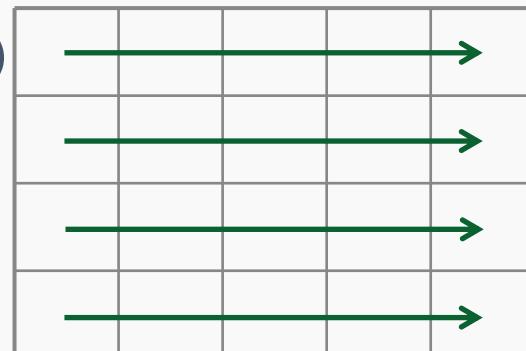
$$B = (b_j)$$

1



или

2



Эффективный обход двумерных массивов: анализ вариантов

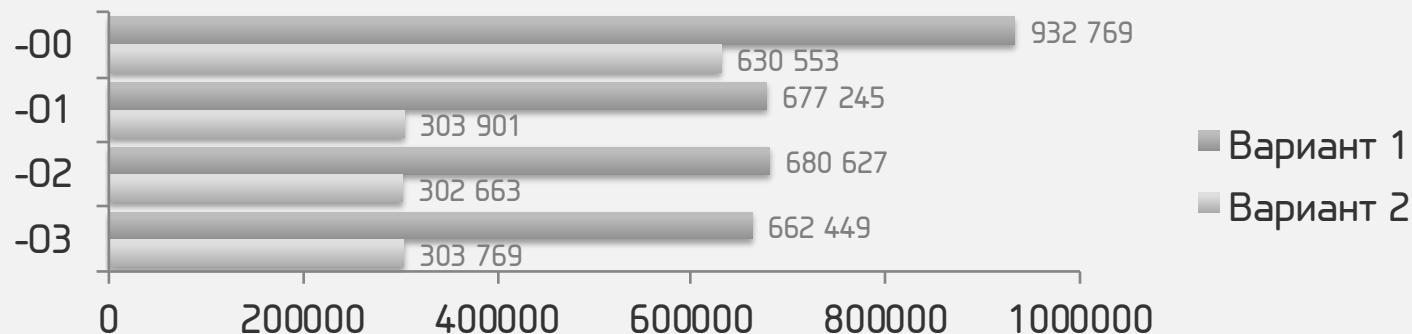


- **Размерность задачи:**
 - набор данных: $2^8 \times 2^8$ (2^{16} элементов, 2^{18} байт);
 - количество тестов: 100;
 - выбираемое время: минимальное.
- **Инфраструктура тестирования:**
 - x86: Intel® Core™ i5 460M, 2533 МГц, L1d: 2 × 32 Кб;
 - x86-64: AMD® E-450, 1650 МГц, L1d: 2 × 32 Кб / ядро;
 - 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, $\approx 2^{22}$ байт);
 - рандомизированная модификация элементов.

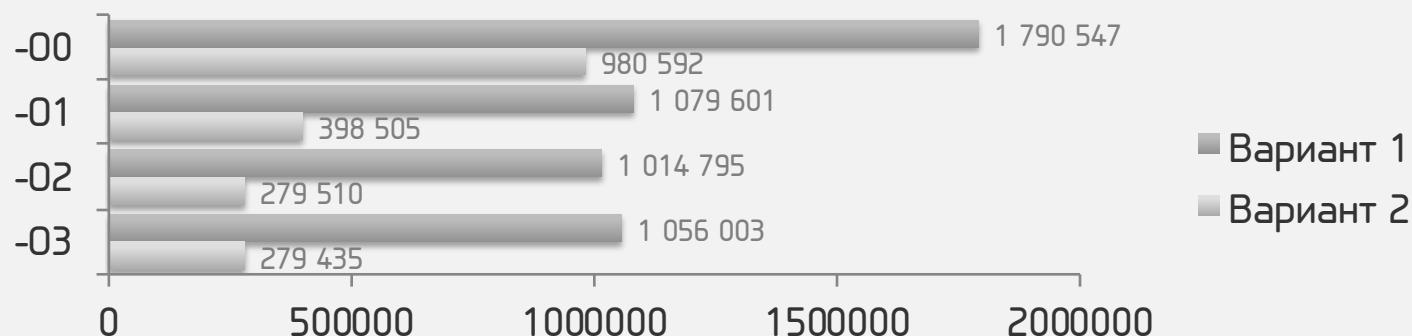
Эффективный обход двумерных массивов: сравнительная эффективность



Эффективность вариантов,
такты ЦП Intel x86



Эффективность вариантов,
такты ЦП AMD x86-64



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.



Array of structures to structures of arrays



```
struct Point {  
    float x;  
    float y;  
    float z;  
};  
  
// Array of structures  
Point points[100];  
  
// ...  
  
points[0].x = 1.0f;
```

```
struct Points {  
    float x[100];  
    float y[100];  
    float z[100];  
};  
  
// Structure of arrays  
Points points;  
  
// ...  
  
points.x[0] = 1.0f;
```

Что еще можно оптимизировать?



- **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.

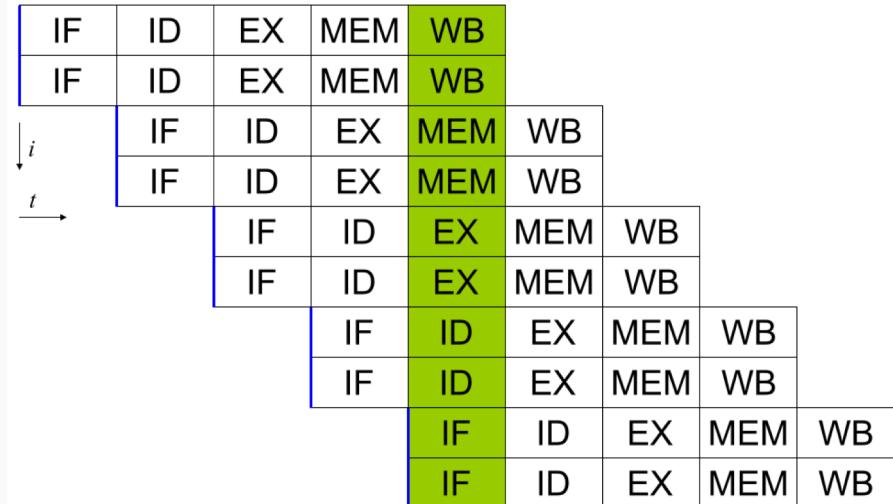
Оптимизация загрузки кэш-памяти команд: как процессор выполняет команды? (1 / 3)



На схеме представлена работа **суперскалярного двухконвейерного** процессора

Суперскалярный процессор – поддерживает **параллелизм** на уровне **инструкций**

Планирование выполнения порядка инструкций определяется вычислительным узлом динамически, а не статически компилятором



Функциональные узлы процессора:

IF – получение инструкции

ID – декодирование инструкции

EX – выполнение инструкции

MEM – доступ к памяти

WB – запись в регистр

Оптимизация загрузки кэш-памяти команд: асимметрия условий (2 / 3)



- 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 (3 / 3)



- 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 большие кодовые фрагменты. Порождаемый же при этом машинный код способен лучше задействовать конвейерную архитектуру микропроцессора.
- Обратной стороной встраивания является увеличение объема кода и большая 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()
{
    // ...
}
```

Работа с внешним миром

Обработка параметров командной строки



- Аргументы командной строки доступны через argc, argv
 - argc – количество аргументов
 - argv – список строковых аргументов
 - argv[0] – имя запускаемой программы
- Ручная обработка этих параметров не всегда удобна
 - gcc -Q --help=optimizers
- Можно использовать библиотеку getopt.h:
 - **int getopt(int argc, char *argv[], const char *optstring);**
 - **int getopt_long(int argc, char *argv[], const char *optstring, const struct option *longopts, int *longlIndex);**
 - **extern char *optarg;**
 - **extern int optind, optarg, optopt;**



Обработка параметров командной строки – пример getopt



```
int main(int argc, char *argv[]) {
    int op, n;
    char *opts = "n::c:h", *c;
    while ((opt = getopt(argc, argv, opts) != -1) {
        switch (opt) {
            case 'n':
                n = atoi(optarg); break;
            case 'c':
                c = optarg[0]; break;
            case 'h':
                printf("Usage: %s [-h|-n 10|-c 'a'] <args>", argv[0]); break;
            default: break; // opt = '?', opterr != 0 -> вывод ошибки
        }
    while (optind < argc) { /* оставшиеся аргументы-не опции */ }
}
```



Обработка параметров командной строки – пример getopt_long



```
/* struct option {  
    const char *name; // имя без “-”  
    int has_arg; // 0 - без арг., 1 - обяз., 2 - не обяз.  
    int *flag; // NULL -> getopt возвращает val,  
                // else -> getopt возвращает 0, *flag = val  
    int val;  
}; */  
  
int c, config_id;  
  
struct option options[] = {  
    {"help", 0, NULL, 'h'},  
    {"config", 1, &config_id, 'c'},  
    {NULL, 0, NULL, 0}  
};  
  
while ((c = getopt_long(argc, argv, "c:h", options, &optIdx)) != -1) {  
    /* ... */  
}
```

Работа с внешним миром

Работа с файлами



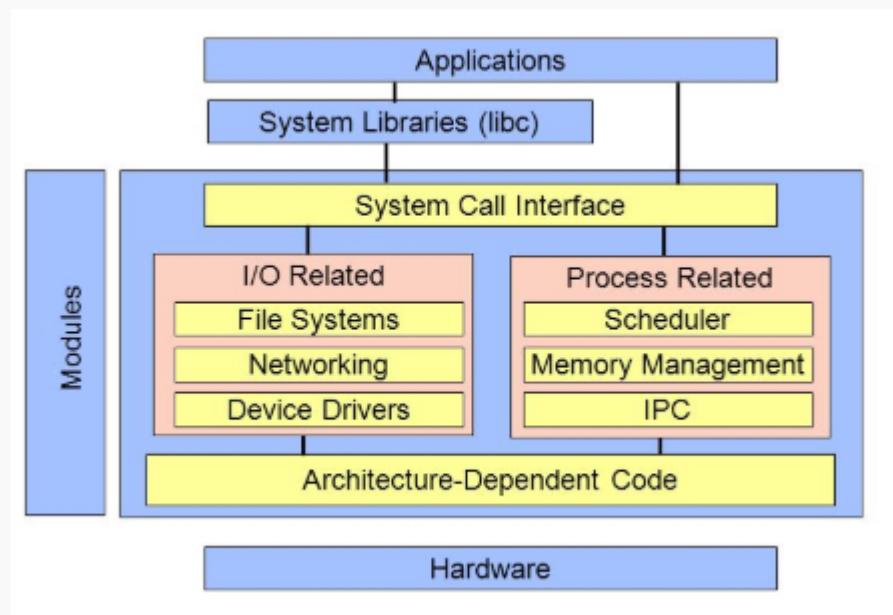
- Работа с файлами и внешними устройствами реализуется через **потоки ввода/вывода**
- Потоки бывают двух видов:
 - **текстовые** – поток символов
 - **двоичные** – поток байт
- Связь между файлом и потоком осуществляется с помощью **системных вызовов** через **дескрипторы**
- Файловые **дескрипторы** – ресурс, который необходимо не забывать **закрывать** после **открытия**
- При **закрытии** файла содержимое буфера **сбрасывается** на внешнее устройство
- При **завершении процесса** все **ресурсы** освобождаются

Работа с внешним миром

Системные вызовы



- Различают два пространства выполнения команд – **уровень пользователя и уровень ядра**
- **Системные вызовы** – уровень ядра
- Системный вызов – **дорогая операция**
- Стандартная библиотека буферизирует работу с внешними устройствами



Работа с внешним миром

Стандартные функции работы с файлами (1 / 3)



- Стандартные потоки – stdin, stdout, stderr
 - `./a.out <test.in >test.out 2>G1`
- Работа с потоками осуществляется через **FILE ***. Большая часть функций – в библиотеке **<stdio.h>**
 - **FILE *fopen(char *name, char *mode);**
 - mode: "r" – read, "w" – write, "a" – append, "r+" – read+write, "rb" – read binary ...
 - **int fclose(FILE *fp);**
- Работа с текстовыми потоками:
 - **int fputc(int ch, FILE *stream);**
 - **int fgetc(FILE *stream);**
 - **int ferror(FILE *stream);**
 - **int feof(FILE *stream);**
 - **int fprintf(FILE *restrict stream, const char *restrict format, ...);**
 - **int fflush(FILE *stream);**
 - ...

Работа с внешним миром

Стандартные функции работы с файлами (2 / 3)



- Работа с двоичными потоками:
 - **size_t fwrite(const void *buf, size_t size, size_t count, FILE *stream);**
 - Возвращает количество реально записанных элементов
 - **size_t fread(void *buf, size_t size, size_t count, FILE *stream);**
 - Возвращает количество реально прочитанных элементов
 - Оба метода сдвигают положение указателя файла
 - Необходимо вручную различать feof() и ferror()
- Пример:

```
FILE *f = fopen("books", "wb");
fwrite(&my_book,
       sizeof(struct book), 1, f);
fclose(f);
```



```
struct book {
    char name[20];
    int pages_count;
} my_book = {"Tutorial", 100};
```



Работа с внешним миром

Чтение из файла (3 / 3)



```
if ((f = fopen("books", "rb")) == NULL) {
    fprintf(stderr, "Failed to open file\n");
    return 1;
}

while (!feof(f)) {
    if (fread(&my_book, sizeof(struct book), 1, f) == 1) {
        printf("Book %s (%d pages)\n", my_book.name, my_book.pages_count);
    }
}

if (fclose(f)) {
    fprintf(stderr, "Failed to close file\n");
    return 1;
}

// $ hexdump books
// 00000000  54 75 74 6f 72 69 61 6c 00 00 00 00 00 00 00 00
// 00000010  00 00 00 00 64 00 00 00
```

Внешние библиотеки



- Увеличивают **повторное использование** кода
- Позволяют **ускорить сборку** проекта
- Повышают **инкапсуляцию** кода проекта
- Различают **статические и динамические** библиотеки
- **Статическая** библиотека
 - **архив объектных** файлов
 - для каждой **динамической** библиотеки используется **своя копия**
 - распространяется в виде **архива** и набора **заголовочных** файлов
 - добавляется к **исполняемому** файлу во время **линковки**, **увеличивая** его **размер**
- **Динамическая** библиотека
 - имеет тот же **формат**, что и **исполняемый** файл
 - используется **одна на процесс**
 - загружается **однократно** в ОС, но **данные** для разных процессов **разделяются**
 - может **заменяться** в **процессе работы** программы
 - подключается на этапе **запуска** приложения

Работа с внешними библиотеками



- Статическая библиотека:
 - Создание:
 - gcc -c *.c
 - ar -crv libmykernel.a *.o
 - (опционально) ranlib libmykernel.a // создание заголовочной секции для ускорения
 - Использование:
 - gcc -L. -lmykernel program.c
 - ./a.out
- Динамическая библиотека:
 - Создание:
 - gcc -fPIC -c *.c
 - gcc -shared -o libcommon.so *.o
 - Использование:
 - LD_LIBRARY_PATH=<путь к директории с библиотекой> ./a.out -L. -lcommon
 - dlopen, dlsym, dlclose – функции для доступа к динамическим библиотекам во время работы программы
 - в динамической библиотеке могут быть функции инициализации и финализации __init() и __fini()
 - посмотреть список зависимостей – ldd ./a.out



Пример подключения динамической библиотеки из программы



```
// libcommon.so -> int sum(int a, int b) { return a + b; }

#include <dlfcn.h> // при компиляции нужно указать -ldl

void *library; // объект для привязки внешней библиотеки
int (*myfunc)(int x, int y); // указатель на импортируемую функцию

// вызов __init(), если есть
library = dlopen("libcommon.so", RTLD_LAZY); // подключение библиотеки
if (!library) { /* ... */ }

myfunc = dlsym(library, "int_sum"); // загрузка функции
printf("%d + %d = %d\n", 10, 20, (*myfunc)(10, 20));

// вызов __fini(), если есть
dlclose(library); // отключение библиотеки - уменьшение счётчика ссылок на неё
```

Пример работы с внешними библиотеками с использованием CMake (1 / 3)



- Статическая библиотека:
 - Создание:
 - add_library(\${PROJECT_NAME}_lib STATIC lib/*.c)
 - target_include_directories(\${PROJECT_NAME}_lib PUBLIC \${PROJECT_SOURCE_DIR}/lib)
 - Использование:
 - add_executable(\${PROJECT_NAME} \${SOURCES})
 - target_link_libraries(\${PROJECT_NAME} PRIVATE \${PROJECT_NAME}_lib)
- Динамическая библиотека
 - Создание:
 - add_library(\${PROJECT_NAME}_lib SHARED src/*.c)
 - add_library(\${PROJECT_NAME}_lib::lib_1 ALIAS \${PROJECT_NAME}_lib)
 - target_include_directories(\${PROJECT_NAME}_lib PUBLIC \${PROJECT_SOURCE_DIR}/lib)
 - Использование:
 - add_executable(\${PROJECT_NAME} \${SOURCES})
 - target_link_libraries(\${PROJECT_NAME} PRIVATE \${PROJECT_NAME}_lib::lib1)

Пример работы с внешними библиотеками с использованием CMake (2 / 3)



Описание установки библиотеки myLib:

```
add_library(myLib STATIC mylib.c mylib.h)
install(TARGETS myLib EXPORT myLibTargets
        LIBRARY DESTINATION lib
        ARCHIVE DESTINATION lib
        INCLUDES DESTINATION include)
install(FILES mylib.h DESTINATION include)
install(EXPORT myLibTargets
        FILE myLibConfig.cmake
        NAMESPACE MyLib::
        DESTINATION lib/cmake/MyLib)
```

Подключение myLib к main:

```
add_executable(main main.c)
find_package(myLib
            CONFIG
            REQUIRED)
target_link_libraries(main
                      MyLib::myLib)
```

Установка myLib:

```
cmake .. -DCMAKE_INSTALL_PREFIX=~/install
cmake --build . --target install
```

Сборка main:

```
cmake .. -DmyLib_DIR=~/install/lib/cmake/MyLib
cmake --build .
```

Пример работы с внешними библиотеками с использованием CMake (3 / 3)



Описание модуля поиска библиотеки

./cmake/FindmainLib.cmake:

```
set(MYLIB_ROOT "$ENV{MYLIBDIR}"
    CACHE PATH "myLib root directory.")
find_path(MYLIB_INCLUDE_DIRS
    NAMES mylib.h
    HINTS $ENV{MYLIBDIR}
    PATHS ${MYLIB_ROOT}
    PATH_SUFFIXES include)
find_library(MYLIB_LIB
    NAMES myLib
    HINTS $ENV{MYLIBDIR}
    PATHS ${MYLIB_ROOT}
    PATH_SUFFIXES lib
    NO_DEFAULT_PATH)
include(FindPackageHandleStandardArgs)
FIND_PACKAGE_HANDLE_STANDARD_ARGS(
    MYLIB DEFAULT_MSG
    MYLIB_LIB MYLIB_INCLUDE_DIRS)
```

Подключение myLib к main:

```
add_executable(main main.c)
set(CMAKE_MODULE_PATH,
    ${CMAKE_MODULE_PATH},
    ${CMAKE_CURRENT_SOURCE_DIR}/cmake/")
# попытка поиска FindmyLib.cmake
find_package(myLib MODULE REQUIRED)
message("Found dirs = ${MYLIB_INCLUDE_DIRS}")
message("Found lib = ${MYLIB_LIB}")
target_include_directories(
    main PRIVATE ${MYLIB_INCLUDE_DIRS})
target_link_libraries(main ${MYLIB_LIB})
```

Сборка main:

```
cmake .. -DMYLIB_ROOT=~/install/
cmake --build .
```

Особенности процесса линковки



- **nm** – посмотреть список экспортируемых и импортируемых (неопределённых) функций объектного файла
- **-fPIC** – расположение кода библиотеки, не привязываясь к его абсолютному расположению в памяти
- При работе из C++ нужно оберачивать код функций на C в окружение **extern "C" { /* ... */ }**
- Правила работы линковщика:
 - линковщик проходит один раз в том порядке, который указан в дескрипторах
 - каждый файл анализируется на предмет экспортируемых и неопределённых элементов – они помещаются в специальные списки таблицы символов
 - при обработке библиотеки после обнаружения одного из искомых символов в объекте осуществляется повторный проход по всей библиотеке с целью разрешения неопределённых элементов объекта
 - Объекты библиотеки, не содержащие нужных символов, игнорируются
 - **-lc** добавляется автоматически
 - **-###** – вывод полного трейса работы линковщика
 - Должно выполняться ODR (One Definition Rule)



Пример 1

циклические зависимости



```
// main.c
int main() { func(0); }

// a.c
int bar(int);
int func(int i) {
    return bar(i + 1);
}

// b.c
int func(int);
int bar(int i) {
    return i > 3 ? i : func(i);
}
```

Как правильно?

- 1) gcc main.o -L. -la -lb
- 2) gcc main.o -L. -lb -la



Пример 1

циклические зависимости



```
// main.c
int main() { func(0); }

// a.c
int bar(int);
int func(int i) {
    return bar(i + 1);
}

// b.c
int func(int);
int bar(int i) {
    return i > 3 ? i : func(i);
}
```

Как правильно?

- 1) **gcc main.o -L. -la -lb**
- 2) **gcc main.o -L. -lb -la**



Пример 2

циклические зависимости



```
// main.c
int main() { func(0); }

// a.c
int bar(int);
int func(int i) {
    return bar(i + 1);
}

// c.c
int zoo(int i) {
    return 6 * i;
}
```

```
ar -r libac.a a.o c.o
```

Как правильно?

- 1) gcc main.o -L. -lac -lb
- 2) gcc main.o -L. -lb -lac

// b.c

```
int func(int);
int zoo(int);
int bar(int i) {
    return i > 3 ? zoo(i) : func(i);
}
```



Пример 2

циклические зависимости



```
// main.c
int main() { func(0); }

// a.c
int bar(int);
int func(int i) {
    return bar(i + 1);
}

// c.c
int zoo(int i) {
    return 6 * i;
}
```

```
ar -r libac.a a.o c.o
```

Как правильно?

- 1) gcc main.o -L. -lac -lb
 - 2) gcc main.o -L. -lb -lac
 - 3) gcc main.o -L. -lac -lb -lac**
- // b.c
- ```
int func(int);
int zoo(int);
int bar(int i) {
 return i > 3 ? zoo(i) : func(i);
}
```



# Пример 2

## циклические зависимости



```
// main.c
int main() { func(0); }

// a.c
int bar(int);
int func(int i) {
 return bar(i + 1);
}

// c.c
int zoo(int i) {
 return 6 * i;
}
```

Циклический re-scan

```
gcc main.o -L. -Wl,--start-group -lac -lb -Wl,--end-group
```

```
ar -r libac.a a.o c.o
```

Как правильно?

- 1) gcc main.o -L. -lac -lb
- 2) gcc main.o -L. -lb -lac
- 3) **gcc main.o -L. -lac -lb -lac**

```
// b.c
int func(int);
int zoo(int);
int bar(int i) {
 return i > 3 ? zoo(i) : func(i);
}
```

# Отметьтесь на портале



- Посещение необязательное, но тем, кто пришёл, следует отмечаться на портале в начале каждого занятия
- Это позволяет нам анализировать, какие занятия были более или менее интересны студентам, и менять курс в лучшую сторону
- Также это даст возможность вам оставить обратную связь по занятию после его завершения

|               |                                                   |               |                                         |                |                |                |
|---------------|---------------------------------------------------|---------------|-----------------------------------------|----------------|----------------|----------------|
| пн, 7 октября | вт, 8 октября                                     | ср, 9 октября | чт, 10 октября                          | пт, 11 октября | сб, 12 октября | вс, 13 октября |
| Занятий нет   | 12:00 Офис Mail.Ru Gro<br>Разработка выпускног... | Занятий нет   | 18:35 213 ЛК<br>Информационная безоп... | Занятий нет    | Занятий нет    | Занятий нет    |

**Расписание занятий с 30 сентября по 5 октября**

Расписание занятий

Привет, друзья!  
Предлагаю вам расписание занятий на эту неделю.  
Начало всех занятий в **18:35** за исключением занятий выделенных в расписании отдельно.

**1 семестр**  
30 сентября(понедельник) Frontend разработка (аудитория 430 ГК)  
3 октября(четверг) Backend разработка (аудитория 430 ГК)

**3 семестр**  
2 октября(среда) Разработка выпускного проекта (Офис Mail.ru)  
3 октября(четверг) Информационная безопасность(аудитория 213 ЛК)

**Семестровые курсы:**  
30 сентября(понедельник) Введение в промышленное программирование и структуры данных, Язык C++(113 ГК)

**Текущее занятие**

Сейчас проводится занятие - **Разработка выпускного проекта Рубежный контроль 3.**

Оно проходит в аудитории **Офис Mail.Ru Group**.

- Отметьтесь, что вы пришли на занятие.  
Так вы улучшите свою посещаемость и вас увидят преподаватель в своём "Журнале посещений".

- Оставьте отзыв о занятии и мы сможем улучшить учебный процесс.

# Порождение новых процессов

## Системный вызов fork()



- Любой пользовательский процесс в UNIX имеет свой **уникальный pid (process id)**, а также **родителя**.
- Для определения идентификаторов процесса и его родителя есть соответствующие функции в `<unistd.h>`:
  - `pid_t getpid();`
  - `pid_t getppid();`
  - `pid_t getpgid(); // получение id группы процессов. Обычно является pid порождающего группу процесса`
- В процессе своей работы может понадобиться **создать новый процесс**. Для этого существует **системный вызов fork**:
  - `pid_t fork();`
- При этом создаётся **полная копия процесса**, включая **набор дескрипторов, карту памяти и текущую исполняемую команду**, но с **уникальным pid**
- После **успешного завершения fork()** программа выполняется уже **двумя процессами – родительским и порождённым дочерним**.
  - Для **родительского fork()** возвращает **pid дочернего процесса**
  - Для **порождённого fork()** возвращает **0**
  - В случае **ошибки fork()** возвращает **-1**



# Пример работы fork (1 / 2)



```
int status = 0;

pid = fork();
if (-1 == pid) { printf("fork failed\n"); return 1; }
if (pid == 0) {
 printf("CHILD: Hello world!\n");
 exit(0);
}
printf("PARENT: Forked child\n");
```



# Пример работы fork (2 / 2)



Сколько раз будет выведено Hello world?

```
#include <unistd.h>
#include <stdio.h>

int main() {
 for (int i = 0; i < 10; ++i) {
 fork();
 printf("Hello world\n");
 }
 return 0;
}
```

# Завершение процессов (1 / 4)



- Процесс может завершить исполнение командой `exit(<status>)`.  
**Родительский процесс обязан проверить код возврата дочернего процесса**, иначе второй остаётся **зомби** до тех пор, пока не завершится родительский процесс
  - `void exit(int status);`
- В состоянии **зомби** процесс не отпускает ОС некоторые **ресурсы** – например, **дескриптор процесса**
- **Родительский** процесс может **проверить завершение дочернего** процесса при помощи:
  - функции `wait`
    - `pid_t wait(int *status)`
    - приостанавливает выполнение процесса, пока дочерний не завершится
    - если он уже завершился – `wait` моментально выходит
  - функции `waitpid`
    - `pid_t waitpid(pid_t pid, int *status, int options);`
  - сигнала `SIGCHLD`

# Завершение процессов (2 / 4)

## Функция waitpid



- **pid:**
  - <-1 – нужно ждать **любого дочернего процесса**, идентификатор группы процессов которого равен **абсолютному значению pid**
  - -1 – ожидание **любого дочернего процесса** – аналогично поведению **wait**
  - 0 – ожидание **любого дочернего процесса**, идентификатор группы процессов которого равен **идентификатору текущего процесса**
  - >0 – означает **ожидание дочернего процесса**, чей идентификатор **равен pid**
- **options:**
  - **WNOHANG** – означает немедленное возвращение управления, если **ни один** дочерний процесс **не завершил выполнение**
  - **WUNTRACED** – означает возврат управления и для **остановленных** (но **не отслеживаемых**) дочерних процессов, о статусе которых **ещё не было сообщено**.

# Завершение процессов (3 / 4)

## Функция waitpid



- **status** – адрес переменной, в которой сохраняется **статус дочернего процесса**:
  - **WIFEXITED(status) != 0**, если дочерний процесс завершился **успешно**
  - **WEXITSTATUS(status)** - возвращает **8 младших бит значения**, который дочерний процесс передал в `return` функции `main` или `exit()`
  - **WIFSIGNALED(status) – != 0** если дочерний процесс завершился из-за **необработанного сигнала**
  - **WTERMSIG(status)** – номер сигнала, который **привёл к завершению**, если **WIFSIGNALED != 0**
  - **WIFSTOPPED(status) != 0**, если процесс остановлен и установлен **WUNTRACED**
  - **WSTOPSIG(status)** – возвращает номер сигнала, из-за которого процесс был остановлен, если **WIFSTOPPED != 0**
- **Возвращает:**
  - **pid > 0**, если завершился дочерний процесс
  - **0**, если используется **WHOHANG**
  - **-1** в случае ошибки: **ECHILD** (не существует), **EINVAL** (неправильный `options`), **EINTR** (использовался **WHOHANG**, но был получен необработанный сигнал или **SIGCHLD**)



# Завершение процессов (4 / 4)

## Пример



```
int main() {
 int pid, status;
 pid = fork();
 if (pid == 0) {
 sleep(2);
 exit(EXIT_SUCCESS);
 }

 waitpid(pid, &status, 0);

 if (WIFSIGNALED(status)) { /* ... */ }
 else if (WEXITSTATUS(status)) { /* ... */ }

 return 0;
}
```



# Пример запуска другой программы



```
// команды семейства exec заменяют текущий образ программы другим
// int execl(const char *path, const char *arg0, ... /*, (char *)0 */);
// int execv(const char *path, char *const argv[]);
#include <stdio.h>
#include <unistd.h>

int main(int argc, char *argv[]) {
 printf("Будет выполнена программа %s...\n\n", argv[0]);
 printf("Выполняется %s", argv[0]);
 // или execv(argv[0], argv);
 execl(argv[0]," ", "Hello", "World!", NULL);
 return 0;
}
// <stdlib.h>
// system("ls -la"); // возвращает статус выхода команды
```

# Взаимодействие с другими процессами



- IPC – Inter Process Communication
- Может быть реализован через
  - файлы
  - mmap – отображение файлов/устройств в память
  - неименованные каналы (pipe, popen)
  - именованные FIFO-каналы
  - сигналы
  - сообщения
  - сокеты
  - сеть
- Позволяет **защищённо** обрабатывать данные в связи с **изоляцией** адресных пространств **разных** процессов
  - Пример – вкладки браузера
- **Независимость выполнения процессов** друг от друга
  - **Падение** одного **не приводит к падению** остальных



# ммар – пример отображения файла в память



```
#include <sys/mman.h>

// void *mmap(void *addr, size_t lengthint, int flags, int fd, off_t offset);
// int munmap(void *addr, size_t lengthint);

int fd = open("big_file", O_RDONLY, 0); // можно fopen + fileno
struct stat st;
stat("big_file", &st);
size_t file_size = st.st_size;
// MAP_PRIVATE - не записывать в файл; MAP_POPULATE - предзагрузка файла ядром
char *region = mmap(NULL, file_size, PROT_READ, MAP_PRIVATE | MAP_POPULATE, fd, 0);
if (region == MAP_FAILED) { /* ... */ }

write(1, region, file_size); // fileno(stdout) = 1

if (munmap(region, size) != 0) { /* ... */ }
close(fd);
```



# mmap – пример shared memory



```
#include <sys/mman.h>

// void *mmap(void *addr, size_t lengthint, int flags, int fd, off_t offset);
// int munmap(void *addr, size_t lengthint);

char *shared_memory = mmap(NULL, PAGESIZE, PROT_READ | PROT_WRITE,
 MAP_SHARED | MAP_ANONYMOUS, -1, 0);

int v = 0;
*shared_memory = 0;

if (!fork()) {
 *shared_memory = 42;
 v = 50;
} else {
 wait(NULL);
}

printf("Shared: %d, not shared: %d\n", *shared_memory, v);
// Shared: 42, not shared: 50
// Shared: 42, not shared: 0
```

# Неименованный канал (pipe)



- `#include <unistd.h>`
- `int pipe(int fd[2]);`
- Не позволяют передавать информацию между не родственными процессами
- Предоставляют одностороннюю передачу данных
- Возвращает:
  - 0 в случае успешного завершения, -1 – в случае ошибки
  - 2 открытых файловых дескриптора:
    - `fd[0]` – открыт для чтения,
    - `fd[1]` – для записи
- Используется совместно с `fork`
  - Важно помнить о копировании дескрипторов и закрывать ненужные



# pipe – пример



```
#define MAX_LEN 80
char str[] = "Simple string\n";
char buf[MAX_LEN];
int fd[2];
pipe(fd);
pid_t pid = fork();
if (-1 == pid) { /* Error handling ... */ }
else if (0 == pid) {
 close(fd[0]);
 write(fd[1], str, strlen(str) + 1);
 exit(0);
}
close(fd[1]);
read(fd[0], buf, sizeof(buf));
printf("Received %s\n", buf);
```

# Неименованный канал (popen)



- `#include <stdio.h>`

```
FILE *popen(const char *command, const char *type);
```

```
int pclose(FILE *stream);
```

- **command** – команда интерпретатора, обрабатывается **sh**, при этом используется **PATH**
- Создаёт канал между **вызывающим процессом и командой**
- Возвращает **NULL** в случае **ошибки**
- **Валидный указатель** в случае **успеха** можно использовать для **чтения** или для **записи** в зависимости от **значения type**:
  - “w”: **caller stdout -> command stdin**
  - “r”: **command stdout -> caller stdin**



# popen – пример



```
#include <stdio.h>
#include <stdlib.h>
#define MAX_LEN 80

int main(int argc, char *argv[]) {
 FILE *pipe_in;
 char read_buf[MAX_LEN];
 if ((pipe_in = popen(argv[1], "r")) == NULL) { exit(1); }
 while (fgets(readbuf, MAX_LEN, pipe_in)) {
 printf("%s", readbuf);
 }
 pclose(pipe_in);
 return 0;
}
// $./a.out "ls -la"
```



# Пример использования именованного FIFO-канала



```
#include <sys/stat.h>

// int mkfifo(const char *pathname, mode_t mode);
// односторонний именованный канал. Позволяет общаться неродственным процессам

int fd;
char *hellofifo = "/tmp/hellofifo";

// именованный FIFO-канал
if (mkfifo(hellofifo, 0666) != 0) {
 /* ... */
}

fd = open(hellofifo, O_WRONLY);
write(fd, "Hello", sizeof("Hello"));
close(fd);

unlink(hellofifo);

int fd;
char *hellofifo = "/tmp/hellofifo";
char buf[100];

fd = open(hellofifo, O_RDONLY);
read(fd, buf, sizeof(buf));
printf("Received %s\n", buf);
close(fd);
```

# Отправка сигнала другому процессу



- Для отправки сигнала процессу (в частности – его уничтожения) используется функция `kill`

```
#include <sys/types.h>
#include <signal.h>
int kill(pid_t pid, int sig);
```
- Если `pid > 0` – задаёт PID процесса, которому посыпается сигнал
- Если `pid = 0`, то сигнал посыпается всем процессам группы, к которой принадлежит текущий процесс
- `sig` – тип сигнала.:
  - `SIGKILL` – приводит к немедленному завершению процесса. Не может быть проигнорирован
  - `SIGTERM` – запрос на завершение процесса
  - `SIGCONT` – запрос продолжения работы из остановленного состояния
  - `SIGALRM` – сигнал-таймер
  - `SIGCHLD` – сигнал получается от ОС при завершении одного из дочерних процессов
  - `SIGUSR1, SIGUSR2` – пользовательские сигналы в POSIX
  - ...



# Пример добавления обработчика сигналов



```
#include <stdio.h>
#include <signal.h>
int counter = 0;
void handler(int signum) {
 if (signum == SIGALRM) {
 printf("Counter = %d\n", counter);
 alarm(1);
 }
 // signal(signum, SIG_DFL); // убрать обработчик сигнала – обработка по умолчанию
}
int main() {
 signal(SIGINT, handler);
 signal(SIGALRM, handler);
 while (1) { ++counter; }
 return 0;
}
```

# Обмен сообщениями между процессами



- Любой процесс с соответствующими привилегиями может поместить сообщение в очередь и считать его из очереди
- Для помещения сообщения не требуется наличия подключённых считывающих процессов
- Ядро хранит информацию о каждой очереди сообщений в виде структуры
- ```
#include <sys/msg.h>
struct msgid_ds {
    struct ipc_perm msg_perm; // разрешения чтения/записи
    struct msg *msg_first; // первое сообщение в очереди
    struct msg *msg_last; // последнее сообщение в очереди
    msglen_t msg_cbytes; // размер очереди в байтах
    msgqnum_t msg_qnum; // количество сообщений в очереди
    msglen_t msg_qbytes; // максимальный размер очереди в байтах
    pid_t msg_lspid; // pid последнего процесса, вызвавшего msgsnd()
    pid_t msg_lrpid; // pid последнего процесса, вызвавшего msgrcv()
    time_t msg_stime; // время отправки последнего сообщения
    time_t msg_rtime; // время последнего считывания сообщения
    time_t msg_ctime; // время последнего вызова msgctl()
}
$ ipcs // вывод списка системных очередей (в общем перечне IPC facilities)
```

Создание и подключение к очереди



- `#include <sys/msg.h>`
`int msgget(key_t key, int oflag);`
- Возвращает `>=0` в случае **успеха** и `-1` в случае **ошибки**
- Возвращаемое значение `msqid` используется функциями `msg` для доступа к очереди
- Идентификатор вычисляется на основе **указанного ключа** с помощью функции `ftok`. Может быть указан `IPC_PRIVATE` для **автогенерации ОС**
- `oflag` – комбинации разрешений чтения/записи

Работа с очередью (1 / 2)



- **int msgsnd(int msgid, const void *ptr, size_t length, int flag);**
 - возвращает 0 в случае успеха, -1 - в случае ошибки
 - ptr указывает на структуру вида

```
struct msghdr {  
    long mtype; // тип сообщения > 0  
    char mtext[N]; // данные  
};
```
 - flag – 0 или IPC_NOWAIT – неблокирующий вызов при недостатке места в очереди
- **ssize_t msgrcv(int msgid, void *ptr, size_t length, long type, int flag);**
 - возвращает количество успешно считанных байт данных
 - ptr – куда следует помещать данные (структура аналогична)
 - length – максимальный размер буфера данных (`sizeof(struct msghdr)` – `sizeof(long)`)
 - type – тип сообщения:
 - 0 – нужно считать первое сообщение
 - >0 – первое сообщение, тип которого равен указанному
 - <0 – первое сообщение с наименьшим типом, значение которого <= |type|
 - flag
 - IPC_NOWAIT – неблокирующий вызов при отсутствии искомого сообщения с возвратом ошибки ENOMSG
 - MSG_NOERROR – при превышении length данные обрезаются без возвращения ошибки E2BIG

Работа с очередью (2 / 2)



- **Блокировка при чтении** снимается, если произошло одно из следующего:
 - **появилось сообщение** с запрошенным типом
 - **очередь msqid** будет **удалена** из системы – ошибка **EIDRM**
 - **вызвавший процесс** будет **прерван перехватываемым сигналом** – **EINTR**
- **Управление очередями** осуществляется командой **msgctl**:
 - **int msgctl(int msgid, int cmd, struct msgid_ds *buff);**
 - **возвращает 0** в случае **успеха** и **-1** в случае **ошибки**
 - **cmd:**
 - **IPC_RMID** – **удаление** очереди **msqid** из системы. **Все сообщения теряются**. Для этой команды **третий аргумент** игнорируется
 - **IPC_SET** – **установка** значений **четырёх полей** структуры **msgid_ds** данной очереди равными значениями соответствующих полей структуры ***buff**: **msg_perm.uid**, **msg_perm.gid**, **msg_perm.mode**, **msg_qbytes**
 - **IPC_STAT** – возвращает **вызывающему процессу** через **аргумент buff** **текущее содержимое** структуры **msgid_ds** для **указанной очереди**



Пример работы с очередью (1 / 2)



```
#define MAX_SEND_SIZE 80

struct mymsgbuf {
    long mType;
    char mText[MAX_SEND_SIZE];
};

void send_message(int qid, struct mymsg *qbuf, long type, char *text) {
    qbuf->mType = type;
    strcpy(qbuf->mText, text);
    if (msgsnd(qid, (struct msgbuf *)qbuf, strlen(qbuf->mText) + 1, 0)) == -1) { /*...*/ }
}

void read_message(int qid, struct mymsg *qbuf, long type) {
    qBuf->mType = type;
    msgrcv(qid, (struct msgbuf *)qbuf, MAX_SEND_SIZE, type, 0);
    printf("Received %s\n", qbuf->mText);
}
```



Пример работы с очередью (2 / 2)



```
int qType = 1, status, stat, pid[argc], msgqid; struct mymsg qBuf;
if ((msgqid = msgget(IPC_PRIVATE, IPC_CREAT|0660)) == -1) { /* Queue creation error */ }
for (int i = 1; i < argc; ++i) {
    pid[i] = fork();
    if (-1 == pid[i]) { /* Error */ }
    else if (0 == pid[i]) {
        char str[40]; sprintf(str, "Hello world from %d", getpid());
        send_message(msgqid, (struct mymsg *)&qBuf, qType, str);
        exit(0);
    }
}
for (int i = 1; i < argc; ++i) {
    status = waitpid(pid[i], &stat, 0);
    if (pid[i] == status) { /* Child process pid[i] exited */ }
}
for (int i = 1; i < argc; ++i) {
    read_message(msgqid, &qBuf, qType);
}
if ((rc = msgctl(msgqid, IPC_RMID, NULL)) < 0) { /* Queue removal error */ }
```

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, удалось 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 прилинованного 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 отсоединенными (`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 (**EBUGY**);
 - **операция «закрыть»** (**блокирующий вариант**) может быть реализована как процедура, которая закрывает открытый мьютекс (и незамедлительно возвращает управление) или блокирует поток до момента открытия 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.
- Возможна ситуация **дедлока**

A->B

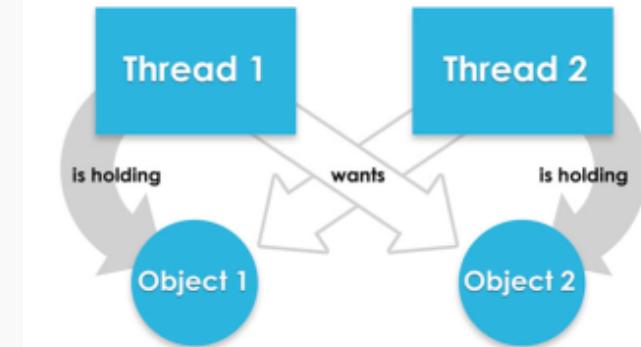
B->A

A.Lock

B.Lock

B.Lock

A.Lock



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 |

Переменные состояния



- Механизм **переменных состояния** (*condition variable*) позволяет организовывать практически любые протоколы взаимодействия потоков POSIX, блокируя их выполнение до наступления заданного события (выполнения предиката) или момента времени.
- Использование переменных состояния предполагает:
 - статическую инициализацию или динамическое создание и уничтожение переменных состояния;
 - 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 переменной состояния в Pthreads является `pthread_cond_t`.

Переменные состояния



```
// Consumer
```

```
pthread_mutex_lock(&mtx);
while (c <> 0) {
    pthread_cond_wait(cv, mtx);
}
pthread_mutex_unlock(&mtx);
process(c);
```

```
// Producer
```

```
pthread_mutex_lock(&mtx);
c = 42;
pthread_cond_signal(cv);
pthread_mutex_unlock(&mtx);
```



Использование переменных состояния: пример (1 / 4)



```
#include <pthread.h>

typedef struct {

    pthread_mutex_t mutex;      // мьютекс для защиты значения
    pthread_cond_t cond;        // перем. сост. для коммуник.
    int           value;       // защищаемое значение

} data_t;

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



Использование переменных состояния: 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 переменных состояния: 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 ПЕРЕМЕННОЙ СОСТОЯНИЯ
    // ТРЕБУЕТ ПРЕДВАРИТЕЛЬНОГО ЗАКРЫТИЯ МЬЮТЕКСА
    errflag = pthread_mutex_lock(&data.mutex);
    if(errflag != 0) // ...
```



Использование переменных состояния: 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;
}
```

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, переменные состояния, очереди сообщений и соответствующие программные вызовы и конструкции языка) синхронизируемых только по мере реальной необходимости;
 - повышает **удобство сопровождения и развития** кодовой базы.

Эволюция языка С

C11



- Основные цели:
 - Сокращение «доверия программисту», акцент на защищённости и безопасности исходного кода
 - Разделение функциональных возможностей на «обязательные» и «необязательные» для поддержки компилятором
- Некоторые нововведения:
 - поддержка многопоточности
 - обобщённые (type-depended) макросы
 - анонимные структуры и объединения
 - управление выравниванием объектов
 - статичные утверждения
 - появление безопасных вариантов функций (например, `gets_s`)
 - Спецификатор функции `_Noreturn`



Эволюция языка С Atomic (C11)



```
#include <stdio.h>
#include <stdatomic.h>
#include <pthread.h>
_Atomic int acnt;
int cnt;
void *count(void *input) {
    for (int i = 0; i < 10000; ++i) { acnt++; cnt++; }
    pthread_exit(NULL);
}

int main() {
    pthread_t tid[10];
    for (int i = 0; i < 10; i++) pthread_create(&tid[i], NULL, count, NULL);
    for (int i = 0; i < 10; i++) pthread_join(tid[i], NULL);
    printf("the value of acnt is %d\n", acnt);
    printf("the value of cnt is %d\n", cnt);
    return 0;
}
```

Эволюция языка С

Реализация spin-lock на Atomic (C11)



```
_Bool atomic_compare_exchange_weak(volatile A *object, // A - Atomic type
                                    C *expected,           // C - Non-Atomic type
                                    C desired);

_Atomic int lock = 0;
void *counting(void *input) {
    int expected = 0;
    for (int i = 0; i < 100000; i++) {
        unlock_count++;
        // if the lock is 0 (unlock), then set it to 1 (lock).
        while (!atomic_compare_exchange_weak(&lock, &expected, 1))
            // if the CAS fails, the expected will be set to 1,
            // so we need to change it to 0 again.
            expected = 0;
        lock_count++;
        lock = 0;
    }
    pthread_exit(NULL);
}
```

Практикум №2



- Решить индивидуальную задачу №2 в соответствии с формальными требованиями.
- Для этого:
 - **узнать постановку задачи из письма от Технопарка**
 - **разработать программу на языке С** в соответствии с заданием
 - **оформить задачу в виде Merge Request** в репозитории
 - **покрыть решение юнит-тестами**
 - **внедрить CI: линтеры, тесты с запуском valgrind**
 - **добиться прохождения всех проверок перед отправкой на реview**
 - **добавить своего ментора** проектной команды в качестве реviewера для проверки
- **Дедлайн:** Семинар № 2 (25.03.2020)
- Напоминание про **дедлайн по ДЗ №1** (до конца РК №1 – 11.03.2020):
 - разбиение на команды, продумывание идеи проекта и возможной реализации, разделение задач между участниками
 - ИЗ №1

Отметьтесь на портале



- Посещение необязательное, но тем, кто пришёл, следует отмечаться на портале в начале каждого занятия
- Это позволяет нам анализировать, какие занятия были более или менее интересны студентам, и менять курс в лучшую сторону
- Также это даст возможность вам оставить обратную связь по занятию после его завершения

| | | | | | | |
|---------------|---|---------------|---|----------------|----------------|----------------|
| пн, 7 октября | вт, 8 октября | ср, 9 октября | чт, 10 октября | пт, 11 октября | сб, 12 октября | вс, 13 октября |
| Занятий нет | 12:00 Офис Mail.Ru Gro Разработка выпускног... | Занятий нет | 18:35 213 ЛК Информационная безоп... | Занятий нет | Занятий нет | Занятий нет |

Расписание занятий с 30 сентября по 5 октября

Расписание занятий

Привет, друзья!

Предлагаю вам расписание занятий на эту неделю.

Начало всех занятий в **18:35** за исключением занятий выделенных в расписании отдельно.

1 семестр

30 сентября(понедельник) Frontend разработка (аудитория 430 ГК)
3 октября(четверг) Backend разработка (аудитория 430 ГК)

3 семестр

2 октября(среда) Разработка выпускного проекта (Офис Mail.ru)
3 октября(четверг) Информационная безопасность(аудитория 213 ЛК)

Семестровые курсы:

30 сентября(понедельник) Введение в промышленное программирование и структуры данных, Язык C++(113 ГК)

Текущее занятие

Сейчас проводится занятие - **Разработка выпускного проекта Рубежный контроль 3.**

Оно проходит в аудитории **Офис Mail.Ru Group**.

- Отметьтесь, что вы пришли на занятие.
Так вы улучшите свою посещаемость и вас увидят преподаватель в своём "Журнале посещений".

- Оставьте отзыв о занятии и мы сможем улучшить учебный процесс.

Оставьте обратную связь



Обратная связь позволяет нам улучшать курс **сразу**, не дожидаясь его завершения.

Умение оставлять конструктивную обратную связь является одним из важнейших при работе в команде, поэтому воспользуйтесь такой возможностью с целью дополнительного обучения.

Была ли понятна цель занятия, основная мысль?

1
 2
 3
 4
 5

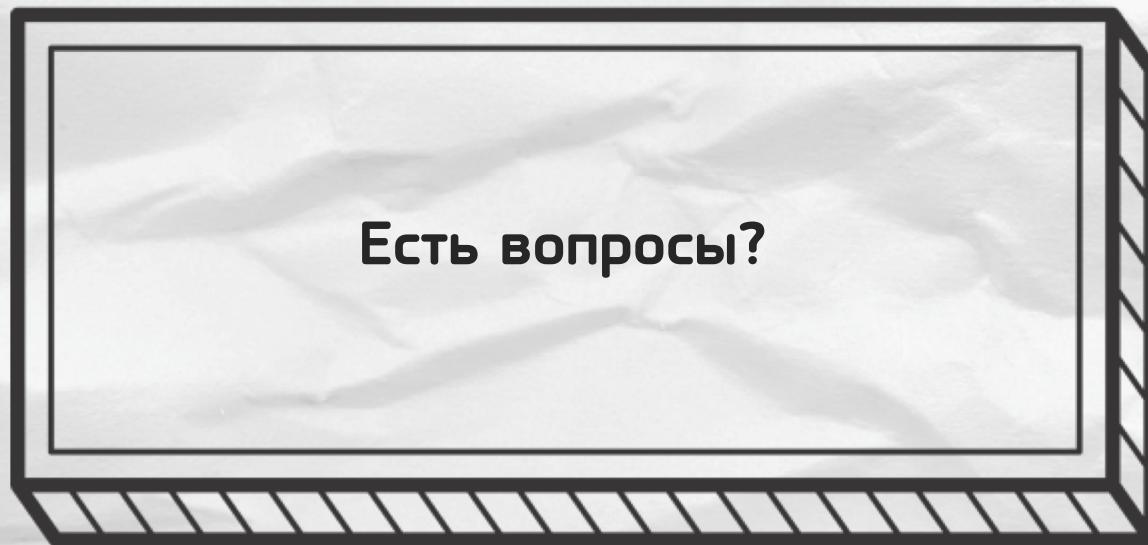
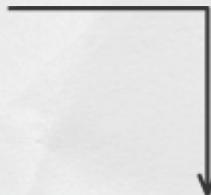
Оцените скорость подачи материала

очень медленно
 медленно
 нормально
 быстро
 очень быстро

Здесь вы можете оставить комментарий к занятию. Преподаватель не увидит автора записи, только текст комментария.

Дополнительный комментарий:

Это занимает **не более 5 минут** и абсолютно **анонимно!**



Приложения



Приложение А. Выравнивание переменных составных типов



- Объекты составных типов языка С выравниваются в соответствии с характеристикой, **наибольшей** среди характеристик выравнивания всех своих элементов, которая в большинстве случаев не достигает длины линии кэш-памяти.
- Другими словами, даже при «подгоне» элементов структуры под линию L1d размещенный в ОЗУ объект может не обладать требуемым выравниванием.
- Принудительное выравнивание статически и динамически размещаемых структур выполняется аналогично выравниванию скалярных переменных и начальных элементов массивов:
 - GCC-атрибут `aligned` в определении типа или объекта данных;
 - функция `posix_memalign`.

Приложение Б. Утилита `rhole`



- Самый доступный способ изучить физическое размещение элементов структуры в оперативной памяти и сопоставить их с загрузкой линий кэш-памяти L1d — утилита `rhole`, входящая в пакет `dwarves`.
- Использование утилиты `rhole` позволяет:
 - проанализировать размещение элементов структур относительно линий кэш-памяти данных;
 - получить варианты реорганизации структуры ([параметр `--reorganize`](#)).

Приложение В. Задача об умножении матриц (1 / 2)



- 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];
```

Задача об умножении матриц (2 / 2)



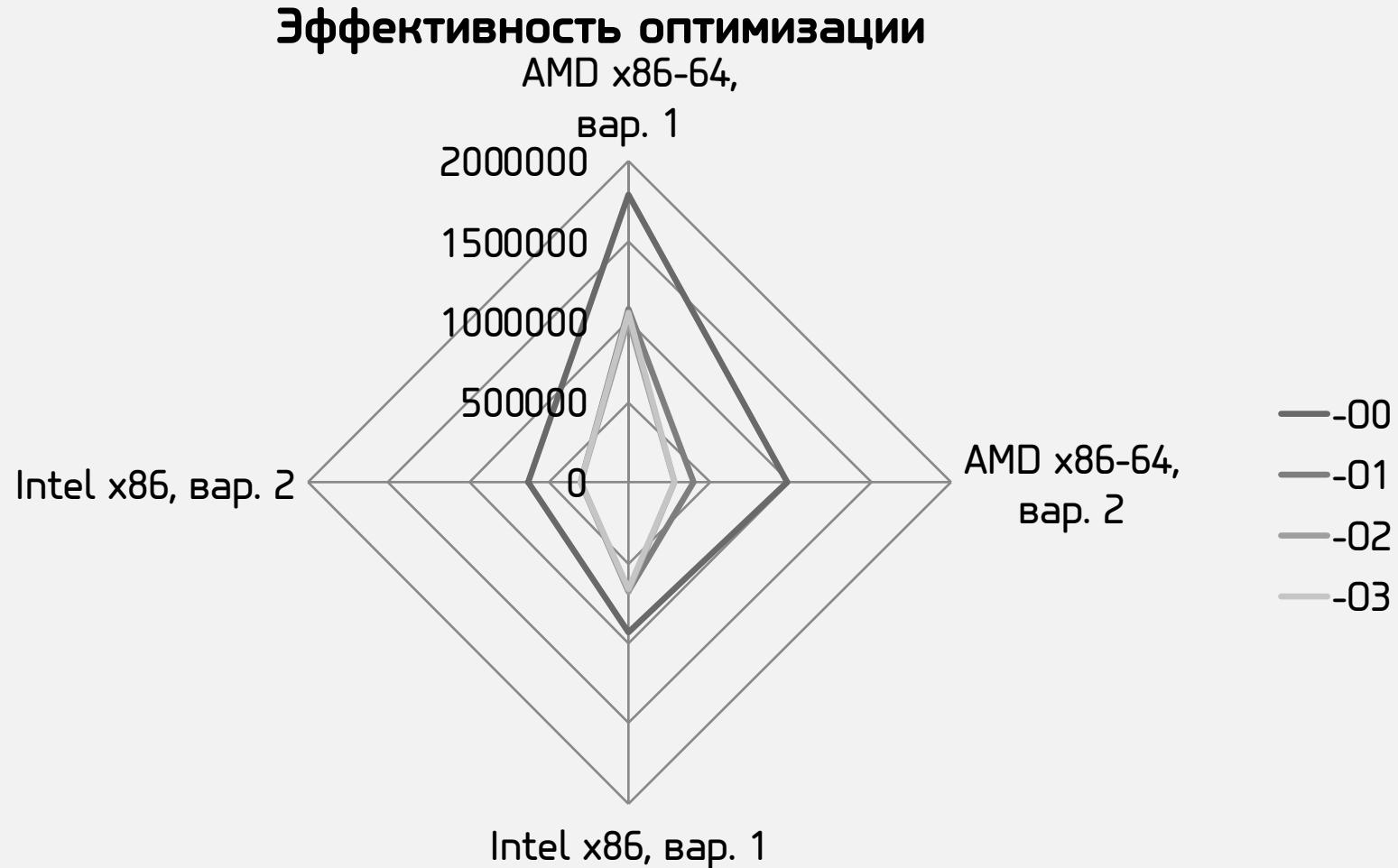
- Предварительное транспонирование второй матрицы повышает эффективность решения в 4 раза (с 16,77 млн. до 3,92 млн. циклов ЦП Intel Core 2 с внутренней частотой 2666МГц; У. Дреппер, 2007). Математически:

$$(AB)_{ij} = \sum_{k=0}^{N-1} a_{ik} b_{jk}^T$$

- На языке С:

```
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];
```

Приложение Г. Эффективный обход массивов: эффективность оптимизации



Приложение Д.

Утилита `rfuncst`. Формат DWARF



- Утилита `rfuncst` 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 `rfuncst` (как и утилиты `rahole`) 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`).

Приложение Е. 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:

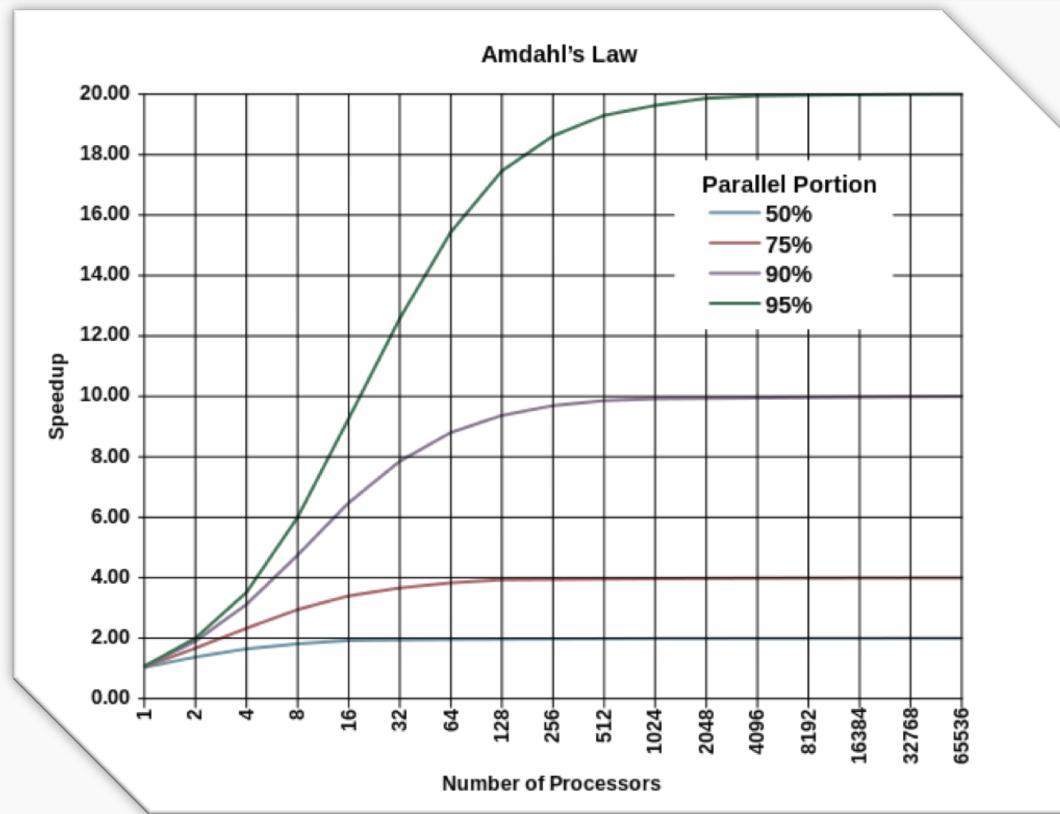
$$S_p = \frac{1}{\alpha + \frac{1 - \alpha}{p}}$$

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, $\lim_{p \rightarrow \infty} S_p = 20$.