Лекция 3. Работа с памятью, выполнение программ

Архитектура (немного истории)

- Гарвардская
  - разные как устройства хранения данных и инструкций, так и шины доступа к ним
- фон Неймана
  - совместное хранение команд и данных
- Гарвардская vs фон Неймана:

  + может быстрее работать (одновременно обращается и к командам и к данным)

  + может быть разная по типу память (и битности)

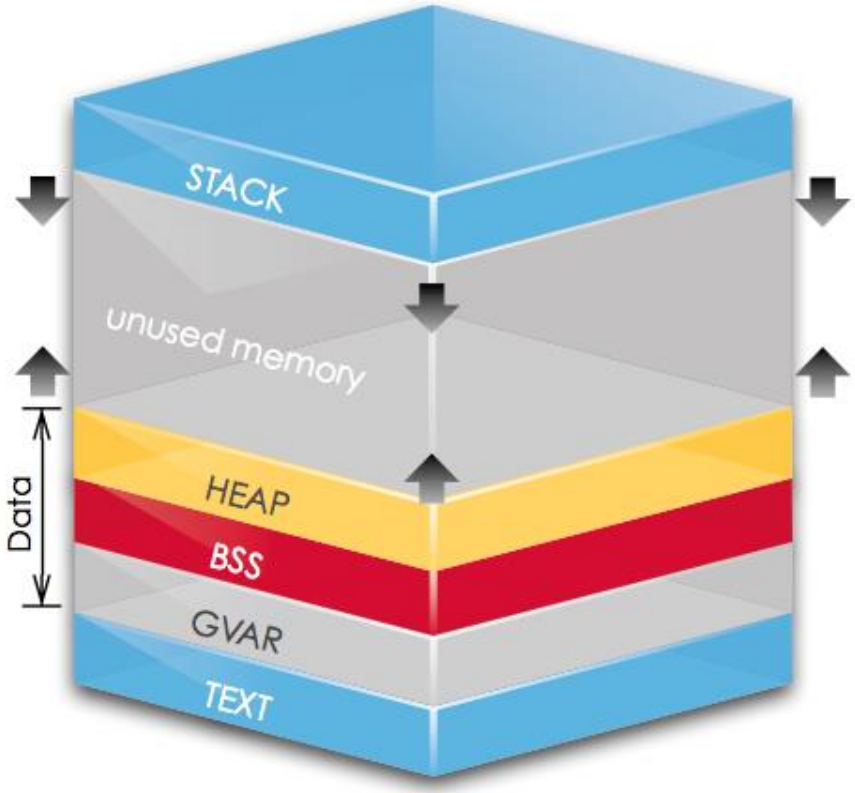
  + проще и дешевле, две шины – технически сложно

  + однородность: например, программы могут быть результатом другой программы
- Компромисс: гибридные архитектуры, например кэш L1 в процессорах x86

Процесс и потоки

- Процесс – ресурсы:
  - адресное пространство (память)
  - объекты ядра (файловые дескрипторы, объекты синхронизации, сокеты, …)
- Поток – выполнение инструкций
  - последовательность команд
  - стек
  - thread local storage (TLS)
  - используют общие ресурсы процесса

Устройство памяти процесса

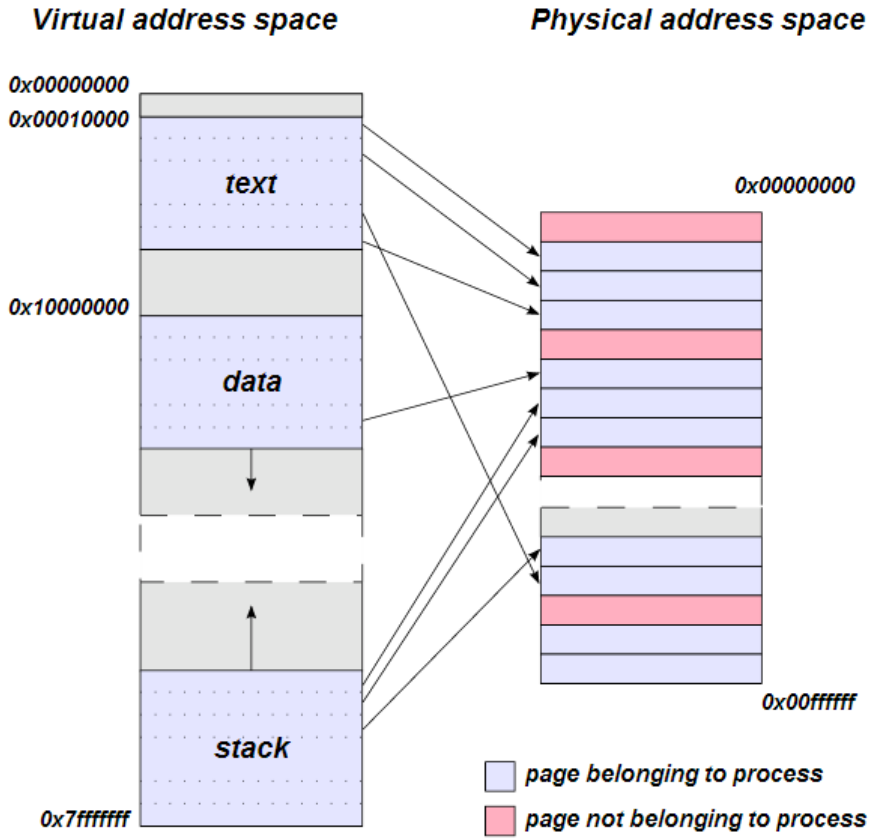


- from http://www.sw-at.com

- Сегмент кода (text)
- Сегмент данных:
  - Глобальные переменные
  - BSS (глобальные переменные без инициализации)
  - Heap (может быть не один)
- Сегмент стека
  - стек может быть не один

Страничная память

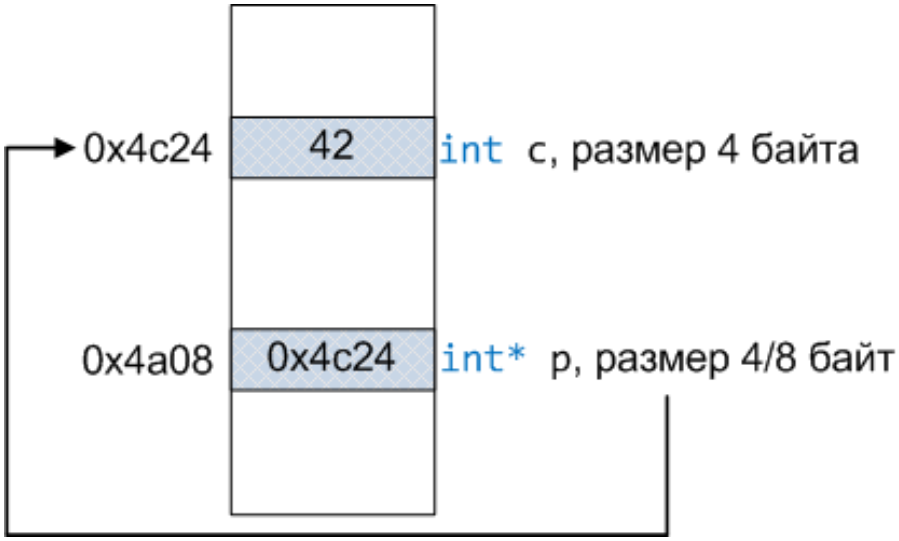
- Задачи:
  - избежать фрагментацию
  - изоляция процессов
  - страницы только для чтения и неисполняемые
  - свопинг
  - отображение в память файлов
  - разделяемая процессами общая память



from http://en.wikipedia.org/wiki/Page_table

Указатели

- Обычная переменная
  - Размер: машинное слово
  - Значение: адрес другой переменной




```
1. int c = 42; // 32 bits in LLP64/LP64
2. int* p = &c;
```

Разыменование
Взятие адреса

- Взятие адреса:

```
1.  int  c = 42;
2.  int *p = &c;
3.
4.  std::cout << &c;
```

```
1.  0x0038f7d8
```

- Разыменование:

```
1.  int  c = 10;
2.  int *p = &c;
3.
4.  *p = 5;
5.  std::cout << *p << " " << c << endl;
```

```
1.  5 5
```

Примеры

```
1.  int  c = 42;
2.  int* p = &c;
3.
4.  int* numbers[3] = {&c, 0};
5.  char**  arr_on_arr;
6.
7.  int* find_value(int* arr);
8.  int (*factorial)(int n);
```

Нулевой указатель и nullptr

- Гарантируется, что нет объектов с нулевым адресом – используем как указатель, который не ссылается на объект

```
1.  void make(int value)   { cout << "int value"; }
2.  void make(char* object){ cout << "char* object"; }
3.
4.  int main()
5.  {
6.      char* uno = 0;
7.
8.      const int NULL = 0;
9.      char* due = NULL;
10.
11.     char* tre = nullptr;
12.
13.     make(0);
14.     make(nullptr);
15.
16.     return 0;
17. };
```

Массивы

- Непрерывная последовательность объектов заданного типа
- Индексация `[0,n-1]`
- Размер – константа. Нужен динамический – std::vector или через `new`.
- В GCC, CLANG есть расширение (а для C99 – в стандарте) для VLA (variable length array).
- Строки – массивы символов

```
1.  int*   uno[3];
2.  double due[3] = {1, 2};
3.
4.  int matrix [3][3] = {{1, 2, 3}, {4, 5, 6}};
5.
6.  int arr    []  = {1, 2, 3};
7.  int bar_arr[2] = {1, 2, 3}; // error: too many initializers
8.
9.  int bad_matrix[3, 3];   // error!
10. due = {1, 3};           // error!
```

Указатели и массивы

- Массив можно использовать там же, где и указатель (как значение)
- При передаче в функцию теряется размер

```
1.  void sort(double* values, int size){/*...*/}
2.  int main()
3.  {
4.      const int n = 7;
5.      double fib[n] = {0, 1, 1, 2, 3, 5, 8};
6.
7.      double *beg = fib;
8.      double *end = &fib[n];
9.      double *mid = &fib[3];
10.
11.     bool b1 = beg[3] == 2;
12.     bool b2 = 4[fib] == 3; // never do like this!
13.
14.     bool b3 = sizeof(fib) == n * sizeof(double);
15.     // sizeof(beg)?
16.     sort(fib, n);
17. };
```

Арифметика указателей

```
1.  int  arr[] = {1, 2, 3, 4, 5, 6, 7};
2.  int* p = arr;
3.
4.  int* q = &arr[3];
5.  int* s = p + 5; // + 5 * sizeof(int)
6.
7.  *++p = 10;       // arr[1] == 10
8.
9.  ptrdiff_t dif = q - s; // -2
10. q + 2 == s;
```

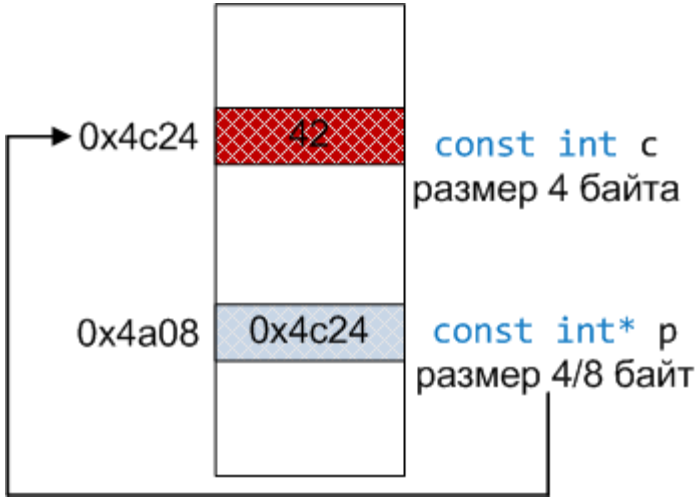
- Не сравнивайте (вычитайте) указатели из разных массивов
- Нельзя складывать

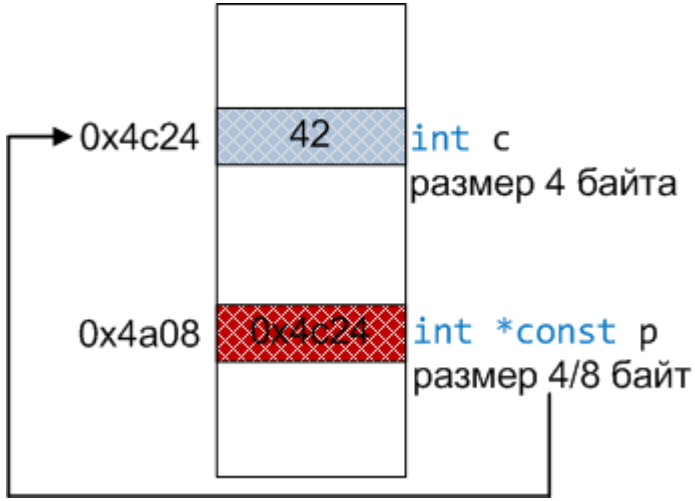
Константы

- Используйте вместо magic numbers
- Часто оптимизируются на этапе компиляции

```
1.  const double pi = 3.14;
2.  const double e; // error: must be initialized if not extern
3.  const double coef[3] = {1, 2, 1};
4.
5.  // just C++ type, not memory allocation type
6.  int value = 5;
7.  const int* pvalue = &value;
8.
9.  value   = 7; // ok
10. *pvalue = 9; // error
11.
12. // no optimization
13. extern const int answer;
14. const double* ppi = &pi;
```

Константные указатели






```
1.
2.  const int c = 0;
3.  const int *p = &c;
4.  int const *q = &c;
5.
6.  *p = 5; // error
7.   p = 0; // ok
```

```
1.  int c = 5;
2.  int *const p = &c;
3.
4.  *p = 5; // ok
5.   p = 0; // error
6.
7.  int const *const full = &c;
```

Ссылки

- Задает псевдоним переменной, обязана быть инициализирована
- Нельзя выполнять операции над ссылками
- Можно думать как о константном указателе, который всегда разыменован
- Можно сделать ссылку на указатель, но не указатель на ссылку

```
 1. void bound(int& x, int min, int max)
 2. { x = (x < min) ? min : ((x > max) ? max : x); }
 3.
 4. int main()
 5. {
 6.     int  x = 5;
 7.
 8.     int& y = x;
 9.     int const& z = x;
10.
11.     ++y;          // x == 6
12.     int* p = &y;// p == &x;
13.
14.     bound(x, -3, 4);
15.     return 0;
16. };
```

Инициализация константных ссылок

- Для обычной ссылки инициализатор должен быть `lvalue` объектом
- Для константной – не обязан (`T const&`):
  - если необходимо, неявное преобразование типа,
  - результат помещается во временный объект типа T,
  - временная переменная используется как инициализатор и живет, пока живет ссылка
- Константная ссылка часто используется как входной параметр для функций

```
1.  void use_ref (int& x)       {/*...*/}
2.  void use_cref(int const& x) {/*...*/}
3.
4.  // ...
5.  int x;
6.
7.  use_ref(x); // ok
8.  use_ref(5); // error
9.
10. use_cref(x); // ok
11. use_cref(5); // ok
```

Выделение памяти

- Выделение/освобождение памяти в heap: операторы `new/delete`
- При нехватке памяти генерируется исключение `std::bad_alloc`

```
1.
2.  try
3.  {
4.      int* p   = new int (42);
5.      int* arr = new int [get_count()];
6.
7.      delete p;
8.      delete [] arr;
9.  }
10. catch(std::bad_alloc const&)
11. {
12.     // ...
13. }
14.
```

new & delete *

- Placement new:

```
1.  void* p = …;
2.  T* pt = new (p) T(…);
3.  pt->~T();
```

- Переопределение операторов:

```
1.  void* operator new   (size_t );
2.  void  operator delete(void* p);
3.
4.  void* operator new   [] (size_t );
5.  void  operator delete[] (void *p);
```

Пара слов про умные указатели*

- RAII - Resource Acquisition Is Initialization
- Эта идиома очень удобна для создания объектов, владеющих ресурсами

```
1.  #include <boost/shared_ptr.hpp>
2.  #include <boost/make_shared.hpp>
3.
4.  int main()
5.  {
6.      using namespace boost;
7.
8.      // usual
9.      shared_ptr<T> p(new T(…));
10.
11.     // true way
12.     auto q = make_shared<T>(…);
13.
14.     return 0;
15. };
```

Утечки памяти (memory leaks)*

- Windows (debug runtime):

```
1.  int main()
2.  {
3.      _CrtSetDbgFlag(
4.          _CrtSetDbgFlag(_CRTDBG_REPORT_FLAG) |
5.          CRTDBG_LEAK_CHECK_DF);
6.
7.      // ...
8.  }
```

- Unix, linux: valgrind (with debug symbols)

```
1.  % valgrind --tool=memcheck --leak-check=yes my_test
2.
3.  ==5015== 100 bytes in 1 blocks are definitely lost in loss record 1 of 1
4.  ==5015== at 0x1B900DD0: malloc (vg_replace_malloc.c:131)
5.  ==5015== by 0x804840F: main (in /home/cpp/my_test.cpp:15)
```

Вопросы?