Dynamic dispatch

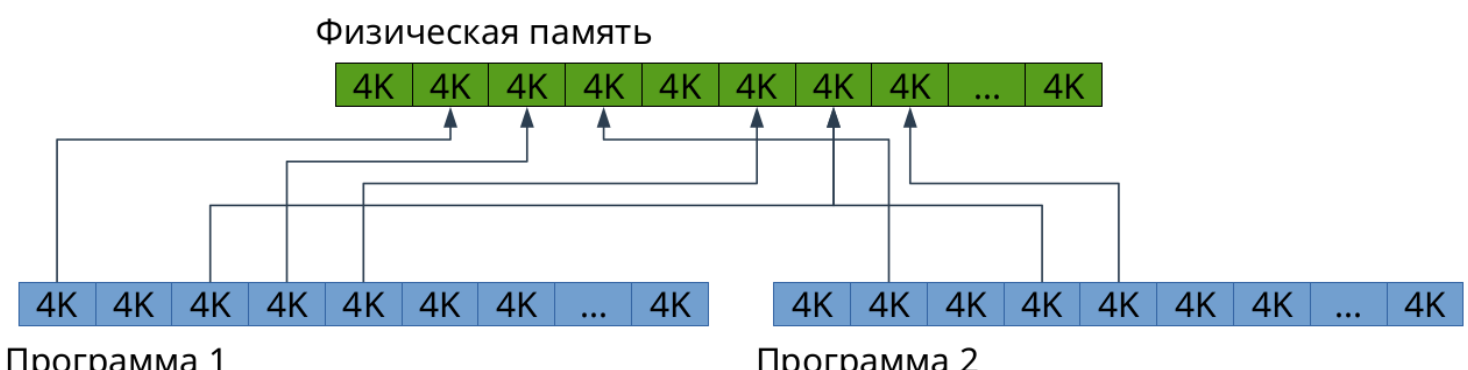
Илья Шпильков

27-10-2023

Спецкурс ФТШ

Пара слов про виртуальную адресацию

- Плохо, когда одна программа залезает в память другой
- Давайте изолируем адресное пространство программ
- Виртуальные адрес преобразуется в физический процессором
- ОС хранит отображение адресосв
- Хранить отображение слово -> слово дорого, поэтому отображаем страницы по 4кб



Устройства памяти процесса

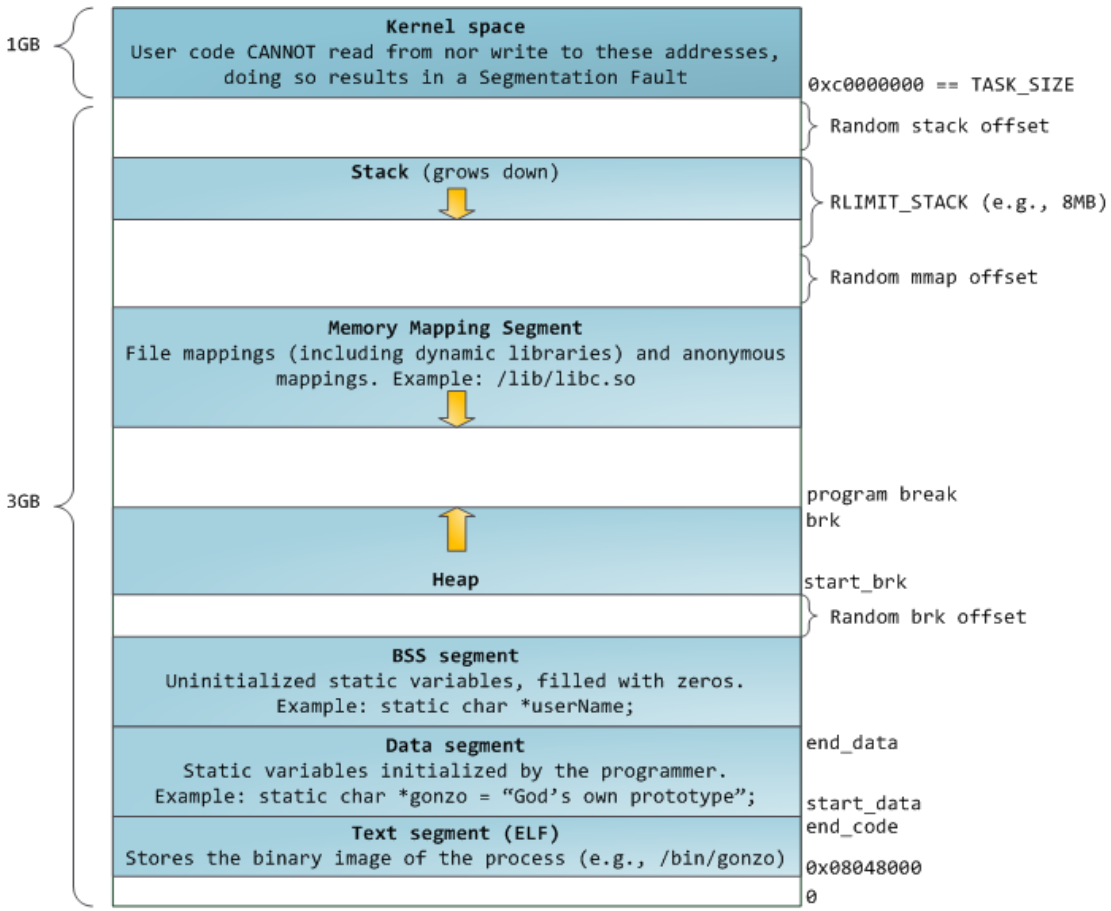


Рисунок 1: Память процесса

Использование указателей в C

Имеет специальное значение: `NULL`

Создание указателя:

- Взять ссылку на уже созданный объект
- Выделить через `malloc` (удалить через `free`)

Арифметика указателей:

- указатель можно сдвигать на знаковое число в пределах границы объекта (обычно массива)

Правила:

- Разыменовывать указатель только на валидный объект
- `T*` и `U*` не могут пересекаться (есть исключения)
- И т.д. (смотреть в стандарт)

Проблемы сырых указателей

`*const T` покрывает слишком широкий класс объектов:

- указатель на один объект
- опциональное значение (так как может быть `null`)
- массив
- способ вернуть несколько значений из функции

Проблемы сырых указателей

`*const T` покрывает слишком много стратегий управлением памятью:

- объект живет в течении всей программы
- значение заимствуется (ссылка)
- уникально владеет памятью и отвечает за ее освобождение
- разделяет память и ведеться подсчет ссылок

Умные указатели

Отдельные типы для каждой ситуации:

- `&/&mut T` – ссылка на заимствованый объект
- `Box<T>` – уникальное владение объектом (`drop` вызывается освобождает память)
- `&/&mut [T]` – заимствование массива
- `Box<[T]>` – владение массивом
- `Option<&T>` – ссылка, которая может быть `null`
- `Option<Box<T>>` – уникальное владение, которое может быть `null`
- `Rc<T>` – раздельное владение (`drop` уменьшает счетчик ссылок и освобождает память при нулевом счетчике)
- ...

Представление `[T]` в памяти

`[T]` – подряд лежащие в памяти объекты типа `T`. Можно считать, что это массив с произвольным размером, который не известен на этапе компиляции.

Для того, чтобы оперировать с `[T]` нужно уметь описывать его конечным количеством информации. Например, достаточно хранить указатель на начало массива и его длину.

`*const [T]` – на самом деле содержит два машинных слова: указатель на начало и длину (аналонично с `&[T]`, `Box<[T]>` и `Rc<[T]>`)

Другие DST

- `str` – Обертка вокруг `[u8]` с инвариантом: содержит валидный utf8.
- `OsStr` – Обертка вокруг `[u8]` с инвриантом: корректная строчка на платформе
- `Path` – Обертка вокруг `OsStr`, которая позволяет работать с сегментами пути

Dynamic dispatch

В трейтах была проблема: на месте вызова generic функции компилятор должен знать конкретные типы.

Dynamic dispatch

В трейтах была проблема: на месте вызова generic функции компилятор должен знать конкретные типы.

Это не позволяет использовать трейты для того, чтобы хранить множество данных объединенных общей функциональностью.

Dynamic dispatch

В трейтах была проблема: на месте вызова generic функции компилятор должен знать конкретные типы.

Это не позволяет использовать трейты для того, чтобы хранить множество данных объединенных общей функциональностью.

Нужно придумать общей тип, который будет объединять все типы, обладащие некоторой функциональностью.

Dynamic dispatch

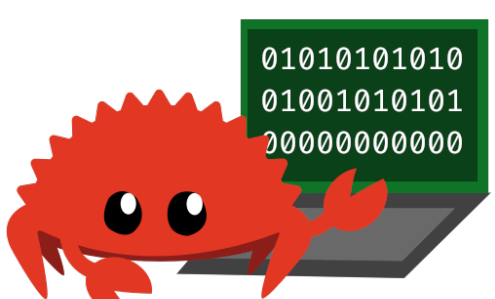
В трейтах была проблема: на месте вызова generic функции компилятор должен знать конкретные типы.

Это не позволяет использовать трейты для того, чтобы хранить множество данных объединенных общей функциональностью.

Нужно придумать общей тип, который будет объединять все типы, обладащие некоторой функциональностью.

Например, будем хранить два указателя: на сами данные и на таблицу указателей на функции. Теперь, на этапе исполнения можно досдать нужную функцию из таблицы и передать ей в качестве первого аргумента указатель на данные.

Динамический диспатч руками



dyn Trait

Реализации этой идеи встроена в компилятор

- `dyn Trait` это тип, для которого реализован `Trait`.
- `dyn Trait` можно хранить только за указателем (`&dyn Trait`, `Box<dyn Trait>`, `Rc<dyn Trait>`).
- Хранится как 2 указателя: на данные и на метаданные (в частности, таблица виртуальных функций).

Пример:

```rust
fn gradient(fns: &[&dyn Calc], x: f64) -> Vec<f64> {
  let mut res = Vec::new();

  for f in fns {
    res.push(derivative(f, x));
  }

  res
}
```

Пара слов об устройстве в памяти

Java:

- все объекты имеет одинаковое представление (указатель)
- этот указатель хранит два других (метаданные и данные)
- данные хранят список реализованных интерфейсов

Rust:

- `Ptr<dyn Trait>` представлен двумя указателями (метаданные и данные)

Impl on Forgein Types.

```
impl Greet for i32 {
  fn greet(&self, _: &str) {
    println!("*silence*")
  }
};

fn main() {
  let x: &dyn Greet = &1;
  x.greet();
}
```

`trait Sized/?Sized`

- По умолчанию, все типовые параметры имеют ограничение `Sized`
- Ограничение можно ослабить указав `T: ?Sized`
- Это ограничение не накладывается на `Self` для трейта. (по умолчанию `trait Foo: ?Sized {}`)
- Если сделать `trait Foo: Sized`, то его нельзя использовать с `dyn`

dyn A + B

Иногда возникает желание указать в качестве ограничения несколько `trait`-ов, но язык не позволяет этого сделать. Есть список разрешенных дополнительных баундов (`auto trait` + lifetime)

Как это будет выглядеть на этапе исполнения? Есть как минимум 2 варианта:

- Хранить ссылку на данные и указатель на объединение таблиц
- Хранить ссылку на данные и множество указателей на таблицы
- Никто не знает, как сделать хорошо, но проблему можно обойти:

```
1 trait AB: A + B {}
2
3 impl<T: A + B> AB for T {}
```

Object Safety

Исходя из реализации динамического полиморфизма следует ряд ограничений:

- все предшественники `trait`-а должны быть object-safe
- `trait` может содержать только функции. (*)
- функция может иметь только lifetimes в качестве generic параметров
- первым аргументом функции может быть только `self`, спрятанный за указатель:
  - `&Self`
  - `&mut Self`
  - `Box<Self>`
  - `Rc<Self>`
  - `Arc<Self>`

Object Safety

Есть ньюансы:

- Метод можно исключить, указав ограничение `Self: Sized`
- (*) Ассоциированные типы должны быть указаны, например, `dyn Index<i32, Output = i32>`

Пример не Object Safe трейтов

```
 1 pub trait Clone {
 2   fn clone(&self) → Self;
 3 }
 4
 5 pub trait Default {
 6   fn default() → Self;
 7 }
 8
 9 pub trait Hash {
10   fn hash<H>(&self, state: &mut H)
11       where H: Hasher;
12 }
```

Практические примеры

dyn Any

Механизм для эмуляции динамической типизации.

```
pub trait Any: 'static {
  fn type_id(&self) → TypeId;
}
```

- `'static` означает, что тип не должен содержать ссылок.
- Для всех типов, удволетворяющих ограничению, `Any` реализован автоматически.

Использование Any

Позволяет стирать тип через неявный каст:

```
let x: &dyn Any = &a;
```

И позже заглянуть внутрь через

```
pub fn downcast_ref<T>(&self) → Option<&T> where
  T: Any,
```

Debug, Display

```
1 pub trait Debug {
2   fn fmt(&self, f: &mut Formatter<'_>) → Result<(),
  Error>;
3 }
4
5 pub trait Display {
6   fn fmt(&self, f: &mut Formatter<'_>) → Result<(),
  Error>;
7 }
```

- Вместо возврата `String` пишется в буффер внутри `Formatter`

Устройство `fmt::Formatter`

Используется для `Debug` и `Display`, при этом, позволяет выводить в **любое** место

```
pub struct Formatter<'a> {
    flags: u32,
    fill: char,
    align: rt::Alignment,
    width: Option<usize>,
    precision: Option<usize>,
    buf: &'a mut (dyn Write + 'a),
}
```

Error

```rust
pub trait Error: Debug + Display {
    // Provided methods
    fn source(&self) → Option<&(dyn Error + 'static)> { ... }
    fn description(&self) → &str { ... } // deprecated
    fn cause(&self) → Option<&dyn Error> { ... } //
  deprecated
}
```

- `description` – **не** нужно использовать. Используйте `to_string`.
- `cause` – **не** нужно использовать: нет поддержки `downcast` (используйте `source`).
- `Error` – Object safe, поэтому можно использовать `dyn Error`
- у `dyn Error` есть методы для `downcast`.

Полиморфизм через `enum`

Если набор возможных типов определен заранее, то можно просто их определить в один `enum`.

Преимущества:

- Не нужно прятать объект за указателем
- Сохраняем контексные оптимизации

Недостатки:

- Нельзя расширять снаружи

enum_dispatch

Чтобы не писать бойлерплейт руками, придумали библиотеку

```rust
#[enum_dispatch]
trait KnobControl {
    // ...
}

#[enum_dispatch(KnobControl)]
enum Knob {
    LinearKnob,
    LogarithmicKnob,
}
```

Сравнение с другими языками

- Java/C# – интерфейсы
- C++ – концепты и виртуальные методы
- Haskell – тайп классы и экзистанциальные типы