c++

Инфраструктура компиляции

Компиляция c++ файлов для выполнения в юзерспейсе основана на том же механизме, что и сборка программ на C. Для удобства исходники програм на C и C++ разделены по директориям user и user-cxx соответственно. На текущий момент нет поддержки сборки проекта имеющего C и C++ исходники одновременно. Аналогично разделены стандартные заголовочные по директориям inc и inc-cxx (и точно также не доступны для разных яхыков, в данном случае это обусловлено необходимостью из-за несовместимости имён функций в C и C++). Реализации библиотечных функций разделены по директориям lib и lib-cxx, однако, для c++ проектов есть доступ к C библиотекам (но не наоборот), потому часть содержимого lib (всё кроме lib/libmain.c) используется при линковке пользовательской программы.

Порядок сборки и логика работы make файлов для сбоки c++ программ повторяет таковую для C и встраивается в существующую систему сборки проекта.

Файл должен присутствовать в переменной-списке USERAPPS в fd/Makefrag (в виде $(OBJDIR)/user-cxx/*имя_файла*, аналогично записям для програм на C). Это необходимо для загрузки исполняемого файла в файловую систему.

Сборкой пользовательской программы управляет файл user-cxx/Makefrag, он повторяет структуру такового для C програм с минимальными отличиями (одно из них — собственный скрипт линковки: user-cxx/user.ld). Этот makefile предполагает изменение пользователем в при необходимости, например, определения особых целей компиляции.

Сборкой стандартной библиотеки управляет файл lib-cxx/Makefrag, который также повторяет структуру lib/Makefrag, но дополнительно зависит от части lib/ артефактов для своей сборки. Он определяет исходники стандартной библиотеки C++ в переменной LIB_CXX_SRCFILES, которые могут быть на C, C++ и ассемблере, а также импортируемые артифакты из библиотеки C в переменной LIB_C_SRCFILES (содержимое которой, как было замечено ранее отличется от LIB_SRCFILES).

Конструкторы и деструкторы

Для конструкторов обычных объектов, похоже, делать ничего не пришлось. Для конструкторов глобальных объектов необходимо получить метку __ctors_start (определна в user.ld) и пройти по всем конструкторам вызывая их. Корректность порядка вызовов обеспечивается линковщиком, т.к. используется SORT_BY_INIT_PRIORITY.

Для конструкторов статических объектов необходимо исопльзовать механизм __cxa_guard_acquire. Была взята реализация из gcc, а после из неё успешно было выкинуто почти всё, т.к. использование __cxa_guard_acquire необходимо в случае если в многопоточной программе несколько потоков могут попытаться одновременно инициализировать статическую переменную, в таком случае необходима синхронизация, но т.к. в JOS все программы однопоточные, то эта часть кода не была портирована. Вторая необходимость — защита от рекурсивной инициализации:

```
class C {
public:
    C() {
        foo();
    }

...

};

foo() {
    static C{};
}
```

Как видно из примера эта проблема возможна и в однопоточном приложении и реализация в gcc в случае возникновения подобной ситуации вызывает экстренное завершение программы, и реализация для JOS повторяет это поведение. В итоге три функции __cxa_guard_acquire __cxa_guard_release и __cxa_guard_abort выполняют схему с 3 состояниями:

1. _GUARD_FREE — инициализация ещё не была начата
2. _GUARD_PENDING_BIT — инициализация в прогрессе
3. _GUARD_BIT — инициализация завершена

В конце (при вызове exit()) необходимо вызвать все деструкторы. Это обеспечивается механизмом atexit. Эта функция (и её builtin аналог из clang

__cxa_atexit) позволяет зарегистрировать набор функций, которые будут дёрнуты при дежурном выходе из main или при вызове exit(). В обоих случаях это приведёт к вызову __cxa_finalize в exit() и вызову всех зарегистрированных функций. На текущий момент можно зарегистрировать только 128 таких функций, но это допустимо, т.к. стандартом требуется возможность зарегистрировать не менее 32 таких функций. В g++ в соответствующем хандлере atexit не вызываются деструкторы глобальных объектов, потому необходимо пользоваться деструкторами определёнными в секции __dtors_start __dtors_end

Поддержка RTTI

Для использования механизма RTTI необходима поддержка со стороны стандартной библиотеки. Для компиляторов gcc и clang это заключается в рализации части __cxxabiv1 касающейся RTTI. Хотя можно было бы полностью с нуля реализовать этот механизм это нецелесообразно, т.е. реализация в libstdc++/libsupc++ не имеет никаких зависимостей от архитектуры системы и требует от стандартной библиотеки только наличия std::abort и new/delete. Модификации к файлам из libstdc++/libsupc++ потребовалась минимальная, заключающаяся исключительно в удалении некоторых используемых директив препроцессора, используемых системой сборки libstdc++.

Динамическая память

Динамическая память в данной реализации предоставляется двумя механизмами — malloc механизмом их C (cstdlib) и new механизмом из c++. Реализация механизма new тривиальна и просто вызывает соответствующие функции из cstdlib. В свою очередь malloc calloc realloc и free используют аллокатор, работающий в юзерспейсе per-process. Используется пул аллокатор. Данный аллокатор может иметь любое количество пулов с разделением каждого на блоки одного, но различного размера. Так же имеется возможность для создания нетривиальных правил аллокации памяти. Механизм позволяет во время компиляции составить список пулов с блоками одинкаовых или разны размеров и упорядочить их. Во время аллокации аллокатор пытается выделить память в первом по списку пуле, который имеет блоки достаточного размера и ещё не исчерпан. Выделение памяти под пулы происходит лениво, но под весь пул сразу (вызовом sys_alloc_region), потому имеет смысл задавать в списке несколько пулов одинакового размера, нежели один большего. Аллокатор выдаст память за O(1), т.к. число пулов ограничено на этапе компиляции, а в каждом пуле сразу же возвращается первый

доступный в списке блок. Возвращение памяти происходит аналогично за O(1). Исключением является aligned_alloc, который будет осуществлять проход по списку, пока не будет найден блок с нужным выравниванием. В случае если блок найден сразу, то поиск завершится за O(1), если же поиск продолжится, то сложность станет O(pool_size), т.к. пойдёт поиск по всем пулам по всему списку кажого. Однако, сложность гаранитрованно не превысит pool_size * pool_cnt.

Библиотеки из C

Портирование библиотек существующих для C (как стандартных, так и специфичных для JOS) не потребовалось. Необходимые заголовочные файлы были скопированы и изменены, чтобы соответствовать стандарту C++, напрмер все функции объявлены как extern “C”. Переработка кода потребовалась только для содержимого файлов entry.S и exit.c в силу того, что при запуске и выходе из c++ программы необходимо производить дополнительные операции с конструкторами и деструкторами классов.

Библиотеки C++

Обеспечена поддержка следующих заголовочных файлов

- <cstddef> (прямое включение доступного заголовка)
- <climits> (собственная реализация, просто добавлены все необходимые дефайны)
- <cfloat> (прямое включение доступного заголовка)
- <limits> (адаптация реализации из gcc, не работает для типов с плавающей точкой в силу отсутствия поддержки SSE)
- <version> (он есть и там есть целый один define)
- <cstdint> (прямое включение доступного заголовка)
- <cstdlib> (самостоятельно реализованы требуемы стандартом функции + alloc-функции, возможно ещё переработаю, там топорная реализация atexit)
- <new> (адаптирована реализация из gcc)
- <typeinfo> <type_traits> (реализация из gcc, минимальные изменеия, удаление и замена специальных макросов GCC, удаление частей кода необходимых для поддержки старых версий стандарта)
- <source_location> (адаптирована реализация из gcc)
- <initializer_list> (адаптирована реализация из gcc)

- <cstdarg> (прямое включение доступного заголовка)

Что не сделано

- <bit> (в процессе)
- <compare>
- <atomic> (ликвидирован?)
- <concepts> (ликвидирован?)
- <coroutines> (ликвидирован?)
- <exceptions> (ждёт соответствующей фичи)

По основынм фичам, реализовано всё (компиляция, бесшовное подключение заголовков, операторы new delete, конструкторы и деструкторы, в том числе глобальные), кроме исключений.

Тесты есть в директории user-cxx, среди них

- test.cpp тест базовых фич, на текущий момент тест конструкторов и деструкторов
- new-test.cpp malloc-test.cpp тесты работы с динамической памятью
- rtti-test тест динмаической информации о типах
- lib-test тест функциональности некоторых библиотек C++

Остаётся реализовать исключения и добавить их в стандартную библиотеку. Написать больше тестов. Сделать ревью кода и исправить недочёты.