Министерство науки и высшего образования Российской Федерации
Федеральное государственное бюджетное образовательное учреждение высшего образования
«Московский государственный технический университет имени Н.Э. Баумана
(национальный исследовательский университет)»
(МГТУ им. Н.Э. Баумана)

ФАКУЛЬТЕТ «Информатика и системы управления»

КАФЕДРА «Теоретическая информатика и компьютерные технологии»

РАСЧЕТНО-ПОЯСНИТЕЛЬНАЯ ЗАПИСКА

К КУРСОВОЙ РАБОТЕ

НА ТЕМУ:

«Сравнение методов коммуникации между процессами в среде Linux»

Студент ИУ9-51Б (Группа) _____ (Подпись, дата) Старовойтов А.И. (И.О. Фамилия)

Руководитель _____ (Подпись, дата) Цалкович П.А. (И.О. Фамилия)

Консультант _____ (Подпись, дата) _____ (И.О. Фамилия)

2024 г.

СОДЕРЖАНИЕ

ВВЕДЕНИЕ

Linux занимает доминирующее положение среди операционных систем в сегменте вычислений на серверах[1], а также пользуется популярностью в качестве десктопной системы у разработчиков[2]. С учетом этого, а также развития модульных, мультипроцессных программных систем, перед разработчиками регулярно встает вопрос эффективной реализации обмена данными между процессами в среде операционной системы Linux.

При проектировании системы использующей межпроцессное взаимодействие, должны учитываться многие факторы, в том числе накладные расходы при применении таких методов. В этом контексте актуальными являются бенчмарки, наглядно показывающие скорость работы существующих интерфейсов обмена данными между процессами.

Целью данной работы является разработка клиентской библиотеки и проведение сравнительного анализа интерфейсов и производительности существующих средств межпроцессного взаимодействия.

1 Обзор предметной области

Межпроцессное взаимодействие — это обмен данными между потоками разных процессов, реалзизованный с помощью механизмов предоставляемых операционной системой.

Механизмы межпроцессной коммуникации в Linux разделяют на три большие категории по функциональному предназначению:[3]

— Коммуникационные (рисунок 1): фокусируются на обмене данными между процессами;

— Синхронизационные: для синхронизации действий между различными процессами;

— Сигналы: могут использоваться как для обмена данными, так и для синхронизации.

Как представлено на рисунке 1, зачастую несколько механизмов предоставляют схожий функционал. Это обусловлено рядом причин, в их числе портирование интерфейсов между вариантами UNIX-подобных систем и разработка новых интерфейсов для избавления от недостатков старых.

Но в некоторых случаях, механизмы предоставляющие схожий функционал, в реальности имеют сильно различающиеся возможности. Например, каналы, в отличие от FIFO, могут использоваться для коммуникации только между процессами, которые имеют общего предка. А потоковые сокеты единственные в этой группе могут использоваться для взаимодействия процессов на разных машинах по сети.

Рисунок 1 — Классификация коммуникационных механизмов межпроцессного взаимодействия в Linux.

1.1 Коммуникационные механизмы

Коммуникационные механизмы являются основным способом передачи данных между процессами в среде Linux. Их разнообразие представлено на рисунке 1.

Данная работа фокусируется именно на коммуникационных механизмах, т.к. именно они представляют наибольший интерес в плане измерения накладных расходов при их использовании.

Коммуникационные методы взаимодействия разделяют на две категории:[3]

— Механизмы передачи данных: их ключевое отличие заключается в операциях чтения и записи. Чтобы произвести обмен данными, требуется два системных вызова. Один процесс должен передать данные в буфер в ядре операционной системы, а другой затем считать их оттуда;

— Разделяемая память: делает доступными другому процессу данные, которые были записаны в разделенный между этими процессами регион памяти. Это позволяет обмениваться данными без накладных расходов на системные вызовы, но требует дополнительной синхронизации между процессами записывающими и считывающими данные.

1.1.1 Механизмы передачи данных

Механизмы передачи данных, в свою очередь, разделяют на три подкатегории:

— Поток байтов: операции чтения и записи могут оперировать различным количеством байтов, данные записанные отдельными операциями не разделяются между собой;

— Сообщения: отделенные друг от друга сообщения, операция чтения может считать сообщение только целиком, операция записи может записать сообщение тоже только целиком;

— Псевдотерминал: используются для программ, ориентированных на использование в терминале, например: ssh, эмулятор терминала.

При этом, выделяют два основных отличия механизмов передачи данных от разделяемой памяти:

— данные нельзя прочитать 2 раза, т.е. после чтения данные удаляются из буфера операционной системы;

— синхронизация между процессами считывающими и записывающими данные производится автоматически.

1.1.2 Разделяемая память

Разделяемую память можно описать как более низкоуровневый инструмент межпроцессной коммуникации, чем передача данных. Преимущество низких накладных расходов при использовании этого инструмента сглаживается необходимостью дополнительной ручной синхронизации, что помимо прочего, усложняет пользовательский код, но делает его более гибким.

1.1.3 Параметры сравнения механизмов межпроцессной коммуникации

При выборе механизма межпроцессной коммуникации, разработчики опираются на ряд факторов, таких как:

- Интерфейс взаимодействия: значительная часть механизмов обмена данными между процессами в Linux используют традиционные для POSIX-систем файловые дескрипторы, что делает интерфейс для работы с ними единообразным и взаимозаменяемым;
- Функциональность: поток байт или сообщения фиксированной длины, поддержка мультиплексирования ввода-вывода, поддержка уведомлений с помощью сигналов, поддержка нескольких слушателей и т.п.
- Возможность передачи данных по сети;
- Совместимость с другими реализациями UNIX;
- Управление доступом;
- Персистентность;
- Производительность.

1.2 Каналы

В UNIX-подобных операционных системах каналы являются первым реализованным методом межпроцессного взаимодействия. Впервые они появились в 1970-х годах.[3] Каналы естественным образом реализуют философию UNIX, позволяя использовать вывод одной команды, как ввод для другой команды, что используется командной оболочкой для составления конвееров обработки данных.

Интерфейс канала предполагает передачу данных в одну сторону. Для двунаправленной передачи данных создают два канала.

Категория “Поток байтов”, к которой относят каналы, означает, что данные не отделяются друг от друга при нескольких вызовах операции записи. Операцией чтения можно считывать из канала любое число байт, но только в той же последовательности, что они были записаны.

Разделенные друг от друга сообщения можно передавать через канал, но это требует дополнительной поддержки на стороне кода приложения. Например, можно отправлять и считывать сообщения фиксированной длины, либо перед телом сообщения отправлять его размер.

В ядре Linux существует константа `PIPE_BUF`. Операции записи, которые передают менее `PIPE_BUF` байт, всегда происходят атомарно. Кроме того, операции записи не блокируются, если в буфере операционной системы достаточно места для сохранения данных, которые еще не считаны читающим процессом. Начиная с версии 2.6.11, размер буфера установлен в 65’536 байт, и может быть расширен до 1’048’576 байт и более, в зависимости от настроек системы.

Каналы создаются с помощью системного вызова `pipe()`, который создает два открытых файловых дескриптора для чтения и записи соответственно. Операция чтения производится с помощью `read()`, а записи с помощью `write`. Это стандартный способ работы с файловыми дескрипторами в Linux.

Чтобы коммуницировать между процессами с помощью канала, необходимо создать канал с помощью вызова `pipe()`, а затем создать дочерний процесс с помощью `fork()`. При этом дочерний процесс наследует открытые файловые дескрипторы родительского процесса. Затем, дочернему и родительскому процессу нужно закрыть файловые дескрипторы соответствующие противоположным концам канала, т.е. читающему процессу нужно закрыть дескриптор для записи, а записывающему заркыть дескриптор для чтения. Это позволит читающему процессу обнаружить, когда записывающий процесс закончит работу с каналом и закроет свой дескриптор для записи. После этих манипуляций, процессы могут использовать операции чтения и записи на соответствующих файловых дескрипторах для коммуникации через канал.

1.3 FIFO

FIFO реализует тот же концепт (“Поток байт”), что и каналы, но допускает взаимодействие несвязанных через `fork()` процессов. Это достигается тем, что FIFO имеет название в файловой системе и имеет интерфейс обычного файла. Таким образом, FIFO иногда называют именованным каналом.

FIFO создается с помощью вызова `mkfifo()`, в который передается путь в файловой системе. Этот вызов создает файл FIFO по переданному пути, после чего процессы могут открыть файловые дескрипторы для чтения или записи используя тот же путь, применяя для этого системный вызов `open()`.

Кроме создания, интерфейс для работы с именованными каналами не отличается от обычных каналов, но предоставляет большую гибкость, т.к. процессы не должны быть связанными.

1.4 Потоковые сокеты

В общем случае, сокеты позволяют передавать данные между процессами на одной машине (хосте), либо на разных машинах, соединенных сетью.

Для коммуникации между двумя процессами, оба процесса должны создать сокет. Один из процессов назначает сокету адрес, который должен быть известен второму процессу, чтобы подключиться к нему.

Сокет создается системным вызовом `socket(domain, type, protocol)`, который возвращает файловый дескриптор для дальнешей работы с созданным сокетом. Параметр `domain` принимает одну из констант, означающую домен для создания сокета. Домен отвечает за метод идентификации сокета (т.е. формат его адреса) и возможность взаимодействия по сети. В этой работе рассматриваются сокеты домена `UNIX`, т.к. они позволяют взаимодействовать процессам в рамках одного хоста, что позволяет избежать накладных расходов при передаче данных по сети.

Параметр `type` системного вызова `socket()` принимает два значения: `SOCK_STREAM` для потоковых сокетов и `SOCK_DGRAM` для датаграммных сокетов.

Потоковые сокеты предоставляют механизм надежной, двунаправленной, потоковой передачи данных между процессами, где:

— надежность означает, что операционная система гарантирует либо целостную передачу данных другому процессу, либо уведомление передающего процесса об ошибке;
— двунаправленность означает, что данные через сокет могут передаваться как от первого процесса второму, так и от второго к первому;
— потоковая передача означает, что данные передаются без разделения на сообщения.

Для сокетов в домене `UNIX` существует упрощенный способ создания и подключения: системный вызов `socketpair(domain, type, protocol, sockfd[2])`, где `domain`, `type`, `protocol` соответствуют параметрам `socket()`, а в массив `sockfd[2]` — записываются два файловых дескриптора ссылающихся на уже сконфигурированные сокеты. Далее, приложение может вызвать `fork()`, чтобы созданный процесс осуществлял обмен данными с родительским процессом через унаследованные файловые дескрипторы сокетов. При этом, сокеты не привязываются к конкретному адресу, а значит недоступны для других, насвязанных, процессов.

После создания и конфигурирования, можно использовать стандартные вызовы `read()` и `write()` для чтения и записи в сокет.

1.5 Датаграммные сокеты

Датаграммные сокеты предоставляют интерфейс передачи данных в виде специальных сообщений, называемых датаграммами. Это позволяет передавать данные в виде детерминированных сообщений, но надежность передачи не гарантируется. Сообщения могут приходить не в том порядке, в каком были отправлены, могут дублироваться или не приходить вовсе.

Датаграммные сокеты создаются аналогично потоковым, но в качестве параметра `type` системного вызова `socket()` передается константа `SOCK_DGRAM`.

Для датаграммных сокетов также работает упрощенный метод создания через вызов `socketpair()`.

После создания и конфигурирования, можно использовать стандартные вызовы `read()` и `write()` для чтения и записи в сокет.

1.6 POSIX очереди

Очереди сообщений позволяют процессам обмениваться детерминированными сообщениями. Основным отличием POSIX очередей является возможность задавать приоритет сообщениям, чтобы более приоритетные возвращались операцией чтения раньше, чем менее приоритетные.

Очередь сообщений создается с помощью вызова `mq_open()`, в который передается название очереди. Если очередь с таким именем не существует, она тут же создается, иначе возвращается дескриптор для существующей очереди.

Запись и чтение осуществляются с помощью вызовов `mq_send()` и `mq_receive()` соответственно. После использования, дескриптор очереди необходимо закрыть с помощью `mq_close()` и удалить с помощью `mq_unlink()`.

1.7 System V очереди

`System V` очереди предоставляют схожий с POSIX очередями функционал для обмена сообщениями, но их интерфейс уступает POSIX аналогу по ряду причин:

— более сложный интерфейс;

— использование целочисленных ключей вместо строковых названий;

— не используется механизм подсчета ссылок для освобождения ресурсов.

Но `System V` очереди все еще могут представлять интерес при реализации межпроцессного взаимодействя, т.к. имеют функционал назначения типа отправляемым сообщениям и гибкий интерфейс их получения на основе типов.

Очередь `System V` создается с помощью вызова `msgget()`, в который передается ключ для создаваемой очереди, а возвращается идентификатор созданной очереди. Далее, полученный идентификатор используется для взаимодействия с очередью.

После получения идентификатора очереди вызовом `msgget()`, запись и чтение из очереди сообщений производится с помощью вызовов `msgsnd` и `msgrcv` соответственно. После завершения работы с очередью ее необходимо удалить с помощью вызова `msgctl(fd, IPC_RMID, nullptr)`.

1.8 Отображаемая память

Отображаемая память может использоваться для различных задач, требующих высокую производительность, включая межпроцессную коммуникацию. Для этой цели отображение создается в родительском процессе, затем с вызовом `fork()` дочерний процесс наследует отображение памяти и ее изменения становятся доступными без задержек и промежуточных буферов. Отображаемая память бывает двух видов:

1. отображение файла;
2. анонимное отображение.

В контексте межпроцессной коммуникации, отображение файла может использоваться для случаев, когда данные должны сохраняться при перезапуске процесса или системы в целом.

Недостатком отображаемой памяти для передачи данных между процессами является необходимость дополнительной синхронизации, которая обычно реализуется с помощью семафоров, но подразумевает более сложную логику на стороне пользовательского кода.

Отображение памяти создается с помощью вызова `mmap`, в который передается тип отображения. Вызов `mmap` возвращает указатель на начало отображения, который затем можно использовать для работы с памятью. Изменения отображаемой памяти доступны процессу с таким же отображением, а в режиме отображения файла автоматически сохраняются в файл на диске ядром операционной системы.

1.9 Разделяемая память

Разделяемая память предоставляет способ создания отображения памяти для несвязанных вызовом `fork()` процессов и без использования отображения файла.

Для инициализации разделямой памяти, делается системный вызов `shm_open`, в который передается имя разделяемой памяти. Этот вызов возвращает файловый дескриптор, который затем передается в вызов `mmap`.

2 Разработка библиотеки и бенчмарков

Для измерения производительности, в данной работе разработана клиентская библиотека, реализующая широкий набор наиболее актуальных методов межпроцессной коммуникации в среде операционной системы Linux, а также набор бенчмарков (тестов производительности) для каждого реализованного класса.

2.1 Архитектура библиотеки

Диаграмма классов библиотеки приведена на рисунке 2.

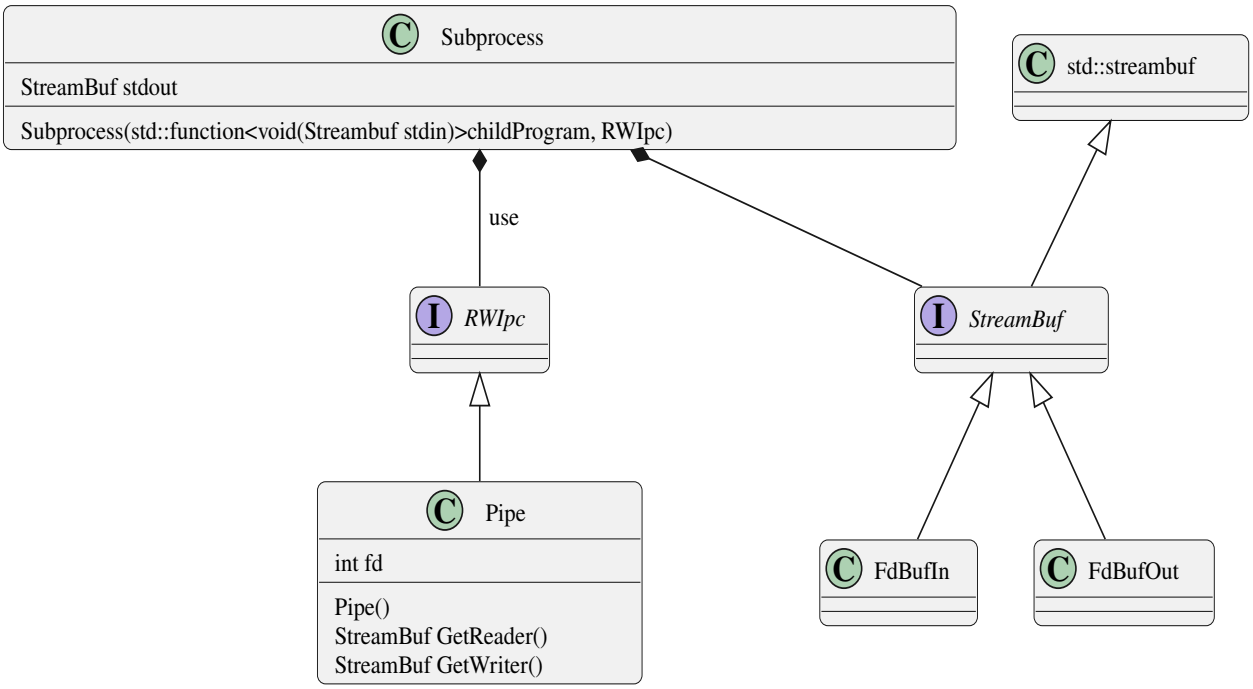

Рисунок 2 — Диаграмма классов библиотеки.

1. Класс `Subprocess` Этот класс отвечает за корректное создание подпроцесса с помощью вызова `fork()`.
   - — Программа подпроцесса: код, исполняемый подпроцессом является функцией языка программирования;
   - — Ввод-вывод: предполагается, что подпроцесс передает какие-то данные родительскому процессу через интерфейс межпроцессного взаимодействия, доступный в функции, исполняемой подпроцессом. Аналогичный интерфейс для получения данных предоставляется в родительском процессе.

2. Интерфейс `RWIpc`. Этот интерфейс является абстракцией над механизмами межпроцессного взаимодействия. Он используется в классе Subprocess для получения объектов, реализующих интерфейсы потокового чтения и записи;
3. Классы `FdBufOut` и `FdBufIn`. Предоставляют интерфейс потоковых буферов для реализации чтения и записи во внешний ресурс. В данном случае, для классов `FdBufOut` и `FdBufIn` этим ресурсом являются файловые дескрипторы, что позволяет использовать их для взаимодействия со значительной частью механизмов межпроцессного взаимодействия;
4. Классы, реализующие интерфейс `RWIpc`. Отвечают за инициализацию соответствующих интерфейсов операционной системы, а также создание объектов, реализующих операции чтения и записи. Также, могут предоставлять собственные реализации классов для чтения и записи.

Эти классы в совокупности позволяют осуществлять однонаправленную передачу данных между процессами, но легко могут масштабироваться для двунаправленного взаимодействия.

Реализация межпроцессного взаимодействия между процессами, связанными вызовом `fork()`, обусловлена невозможностью реализации взаимодействия несвязанных процессов для некоторых механизмов, реализуемых операционной системой. При этом, производительность приведенных в работе методов не зависит от связанности задействованных процессов, поэтому результаты тестов будут применимы для различных архитектур приложений, использующих методы межпроцессной коммуникации.

2.2 Архитектура тестов производительности

Тесты производительности должны иметь одинаковую стрктуру и логику передачи данных:

1. Генерация буфера со случайными данными заданной длины, где длина задается в параметрах бенчмарка. Данный буфер будет использоваться для передачи данных.
2. Создание буфера для чтения данных. Размер этого буфера должен совпадать с размером буфера для передачи данных.

3. Измеряемая часть бенчмарка. Создает подпроцесс, который последовательно в цикле записывает данные из буфера с помощью интерфейса межпроцессного взаимодействия. Количество итераций определяется параметром бенчмарка.
4. Продолжение измеряемой части бенчмарка. Считывает через интерфейс межпроцессного взаимодействия записанные подпроцессом данные.

При этом, измеряемая часть бенчмарка запускается n раз в рамках одного прогона бенчмарков, где n определяется алгоритмом таким образом, чтобы с наибольшей точностью измерить среднее время выполнения итерации теста.

После выполнения, результаты бенчмарков выводятся в терминал (по умолчанию), либо в формате `JSON` для последующего анализа.

2.3 Выбор инструментов разработки

Для разработки библиотеки был выбран язык `C++`[4] из-за следующих его особенностей:

1. Высокая производительность. Для измерения накладных расходов при использовании межпроцессной коммуникации важно, чтобы язык реализации был быстродействующим, для правильной оценки максимально возможной производительности механизмов ядра операционной системы;
2. Поддержка ООП и RAII. Позволяет эффективно реализовать управление системными ресурсами, такими как файловые дескрипторы.
3. Широко известные инструменты для тестирования корректности и производтельности кода. Таковыми являются библиотеки `GoogleTest`[5] для юнит-тестов и `google/benchmark`[6] для микробенчмарков.
4. Поддержка вызовов функций из библиотек языка C. Системные вызовы в Linux реализованы в виде библиотек на языке C. Возможность их вызова без дополнительных оберток является преимуществом как для реализации соответствующих бенчмарков, так и для системного программирования в целом.
5. Инструменты для статического анализа кода и санитайзеры. Позволяют проверить корректность реализации библиотеки и гарантируют отсутствие утечек памяти и других системных ресурсов.

Подробнее рассмотрим библиотеку для измерения производительности кода `google/benchmark`[6]:

1. Библиотека `google/benchmark` предоставляет минималистичный интерфейс для реализации тестов производительности, схожий с юнит-тестами;
2. `google/benchmark` может конфигурироваться для вывода результатов бенчмарков в виде машиночитаемых форматов, таких как JSON;
3. `google/benchmark` поддерживает флаги для запуска отдельных бенчмарков, что помогает в их отладке и итеративной разработке;
4. `google/benchmark` позволяет легко и декларативно параметризировать тесты, что добавляет гибкости в сравнении кода при различных параметрах.

2.4 Разработка классов для межпроцессного взаимодействия

Из приведенного на рисунке 1 разнообразия механизмов межпроцессной коммуникации, для сравнения выбраны следующие механизмы:

— Каналы. Наиболее широко применяемый метод, отличающийся простотой интерфейса и неплохими показателями производительности;
— Именованные каналы (FIFO). Аналог каналов для несвязанных процессов;
— Потоковые сокеты. Интересны с точки зрения универсальности интерфейса для взаимодействия как на одном хосте, так и по сети;
— Датаграммные сокеты. Аналогично потоковым сокетам;
— POSIX очереди. Позволяют передавать детерминированные сообщения, а также задавать им приоритет;
— System V очереди. Аналогично `POSIX` очередям, но позволяют задавать целочисленный тип сообщения;
— Анонимные отображения памяти. Самый быстрый способ передавать данные между процессами. Используются для взаимодействия между связанными процессами;

— Файловые отображения памяти. Аналогичны анонимным отображениям, но позволяют сохранять данные, переданные между процессами после завершения этих процессов или перезапуска системы. Также, позволяют соединять несвязанные вызовом `fork()` процессы.
— POSIX разделяемая память. Аналогично файловым отображениям памяти, но без персистентного хранения данных в файле.

Следующие механизмы не включены в сравнение:

— System V разделяемая память. По функционалу и производительности идентична `POSIX` разделяемой памяти, но уступает удобством интерфейса.
— Псевдотерминалы. Имеет специфичный круг задач, для которых может применяться.
— Сигналы и синхронизационные методы. Не подходят для передачи значительных объемов данных.

3 Реализация библиотеки

Для выполнения поставленной задачи необходимо реализовать библиотеку для межпроцессного взаимодействия, а также тесты производительности каждого реализованного метода.

Разработка проекта осуществлялась в редакторе кода Emacs[7]. Данный редактор может быть использван в качестве IDE благодаря обширной библиотеке плагинов.

Для сборки использовалось окружение NixOS[8]. Этот дистрибутив Linux построен на пакетном менеджере `nix` и позволяет декларативно задавать пакетные зависимости для каждого проекта.

В качестве системы сборки использован CMake[9], а компилятором выступает clang.

3.1 Особенности реализации класса Subprocess

Объявление класса Subprocess представлено на листинге 10.

Для реализации чтения и записи через интерфейс межпроцессного взаимодействия, используются потоковые буферы `std::streambuf`. Класс Subprocess принимает два таких класса InBuf и OutBuf, наследованные от `std::streambuf`, для чтения и записи соответственно. Для этого объявляется концепт StreamBuf, как `std::derived_from` от `std::streambuf`.

Также, в качестве концепта объявляется интерфейс `RWIpc`, объект, удовлетворяющий которому, принимается конструктором класса Subprocess. Этот интерфейс требует два метода: GetReader и GetWriter, которые возвращают инициализированные объекты `StreamBuf` для чтения и записи через механизм межпроцессной коммуникации.

Конструктор класса Subprocess отвечает за создание подпроцесса (листинг 1). Для этого, в созданном с помощью системного вызова `fork()` процессе, запускается функция subprogram, куда передается результат `ipc.GetWriter()`.

При этом, объекты типа `StreamBuf` отвечают за деаллокацию системных ресурсов, выделенных при создании объекта типа RWIpc.

Листинг 1: Определение конструктора класса Subprocess

```
namespace stewkk::ipc {

template <StreamBuf InBuf, StreamBuf OutBuf>
Subprocess<InBuf, OutBuf>::Subprocess(std::function<void(OutBuf)>
↪  subprogram, RWIpc auto ipc)
    : stdout(-1) {
  auto pid = Fork();

  if (pid == 0) {
    subprogram(ipc.GetWriter());
    std::exit(0);
  }

  child_pid_ = pid;
  stdout = ipc.GetReader();
}

}  // namespace stewkk::ipc
```

3.2 Особенности реализации потоковых буферов

Потоковые буферы реализуют интерфейс записи и чтения во внешний интерфейс. Пример объявления класса для чтения из файлового дескриптора[10] приведен на листинге 2.

Конструктор класса принимает файловый дескриптор и сохраняет в виде члена класса, а деструктор закрывает его. Переменные `buf_` и `use_buf_` имитируют буфер из одного символа для посимвольного чтения. Методы `uflow()` и `underflow()` служат для посимвольного чтения, а `sgetn()` для чтения size байт в буфер buf. Определения этих функции представлены на листинге 11.

Объявление класса для записи в файловый дескриптор представлено на листинге 3. Интерфейс аналогичен клссу FdBufOut, но методы для записи отличаются: `sputn()` для записи size байт в буфер и `overflow()` для посимвольной записи. Определения методов класса FdBufOut представлены на листинге 12.

Наследование от класса `std::streambuf` позволяет использовать данные классы в интерфейсах `std::iostream` стандартной библиотеки C++.

Листинг 2: Объявление класса для чтения из файлового дескриптора

```
#include <cstdint>
#include <optional>
#include <streambuf>

namespace stewkk::ipc {

class FDBufIn : public std::streambuf {
public:
  explicit FDBufIn(std::int32_t fd);
  ~FDBufIn();
  ...
  std::streamsize sgetn(char* buf, std::streamsize size);
private:
  int_type uflow() override;
  int_type underflow() override;
private:
  std::optional<std::int32_t> fd_;
  char buf_;
  bool use_buf_;
};

}  // namespace stewkk::ipc
```

Листинг 3: Объявление класса для записи в файловый дескриптор

```
#include <cstdint>
#include <optional>
#include <streambuf>

namespace stewkk::ipc {

class FDBufOut : public std::streambuf {
public:
  explicit FDBufOut(std::int32_t fd);
  ~FDBufOut();
  ...
  std::streamsize sputn(const char* buf, std::streamsize size);
private:
  int_type overflow(int_type c) override;
private:
  std::optional<std::int32_t> fd_;
};

}  // namespace stewkk::ipc
```

Схожим образом реализованы потоковые буферы для методов межпроцессной коммуникации, которые осуществляют операции чтения и записи не через файловые дескрипторы. Во всех реализациях деструкторы потоковых буферов отвечают за деаллокацию системных ресурсов.

3.3 Особенности реализации обработки ошибок

Обработка ошибок производится с помощью исключений. Для этого объявлены два класса, наследующихся от стандартных типов исключений из библиотеки языка C++ (листинг 4).

Листинг 4: Объявление классов исключений

```
#include <system_error>

namespace stewkk::ipc {

class SyscallError : public std::system_error {
  using std::system_error::system_error;
};

SyscallError GetSyscallError();

class IpcError : public std::logic_error {
  using std::logic_error::logic_error;
};

}  // namespace stewkk::ipc
```

Реализация функции `GetSyscallError()` представлена на листинге 13. Она использует переменную `errn` для определения номера ошибки системного вызова, а затем создает объект класса `SyscalError`, который наследуется от `std::system_error`, что позволяет выводить строковое название ошибки и ее описание.

Вызов `GetSyscallError()` используется в обертках над системными вызовами при обработке ошибок. Например, на листинге 5 приведена реализация системного вызова `close()` с проверкой возвращаемого значения. Если вызов вернул `-1`, что означает ошибку, функция бросает исключение, полученное с помощью `GetSyscallError()`.

Листинг 5: Реализация обертки над системным вызовом close()

```
#include <unistd.h>

namespace stewkk::ipc {

void Close(std::int32_t fd) {
  auto res = close(fd);
  if (res == -1) {
    throw GetSyscallError();
  }
}

}  // namespace stewkk::ipc
```

3.4 Особенности реализации класса Pipe

Рассмотрим реализацию классов для поддержки различных механизмов межпроцессной коммуникации на примере каналов.

На листинге 15 приведено объявление класса Pipe. Данный класс реализует интерфейс `RWIpc` для создания канала, а также инициализации потоковых буферов для чтения и записи из этого канала.

Реализация методов приведена на листинге 14. В конструкторе Pipe вызывается функция `pipe()`, которая возвращает два файловых дескриптора: для чтения и записи из канала.

Далее, в функциях `GetWriter()` и `GetReader()`, закрывается один из файловых дескрипторов, который в текущем процессе (родительском для `GetReader()` и дочернем для `GetWriter()`) больше не нужен. Оставшийся файловый десприптор передается в конструктор потокового буфера для работы с файловыми дескрипторами. Созданные таким образом потоковые буферы непосредственно осуществляют запись или чтение в канал.

4 Тестирование

Тестирование разработанной библиотеки осуществлялось с помощью юнит-тестов и тестов производительности.

4.1 Тестирование корректности

Юнит-тесты покрывают все реилизованные механизмы межпроцессного взаимодействия, и построены по одному шаблону. На листинге 6 приведен пример юнит-теста для тестирования класса Pipe.

Листинг 6: Реализация теста передачи данных через канал

```
#include <gmock/gmock.h>

#include <stewkk/ipc/fdstreambuf.hpp>
#include <stewkk/ipc/pipe.hpp>
#include <stewkk/ipc/subprocess.hpp>

using ::testing::Eq;

namespace stewkk::ipc {

namespace {

void ChildProgramm(FDBufOut out) { out.sputn("hello", 5); }

}  // namespace

TEST(PipeTest, ReceivesMessageFromSubprocess) {
  Subprocess<FDBufIn, FDBufOut> child(ChildProgramm, Pipe());
  std::string got;
  got.resize(5);

  child.stdout.sgetn(got.data(), got.size());

  ASSERT_THAT(got, Eq("hello"));
}

}  // namespace stewkk::ipc
```

В приведенном тесте инициализируется подпроцесс, в который передается объект класса Pipe и подпрограмма, которая записывает сообщение из 5 символов в канал. Затем в теле теста считывает сообщение из 5 символов и сравнивается

с искомым, записанным ранее, значением. Таким образом, проверяется корректность передачи данных между процессами.

Юнит-тесты и бенчмарки были дополнительно собраны и запущены в конфигурациях с санитайзерами `-fsanitize=undefined` и `-fsanitize=address`, а также с программой `valgrind`. Эти инструменты не выявили никаких проблем, что гарантирует правильную работу с памятью.

4.2 Тестирование производительности

Тестирование производительности производилось с помощью микробенчмарков, реализованных на основе инструмента `google/benchmarks`. Для каждого поддержаного в библиотеке способа межпроцессной коммуникации проверялась производительность однонаправленной передачи данных от дочернего процесса, созданного с помоью `fork()` к родительскому процессу. Тесты производительности параметризировались размером сообщения в байтах, который принимал значения от 16 до 8192 с множителем прогрессии равным 2. Также, предусмотрена глобальная конфигурация общего числа передаваемых байт.

Каждый из бенчмарков построен по единой схеме. Пример реализации бенчмарка для каналов представлен на листинге 7.

Листинг 7: Реализация бенчмарка каналов

```
#include <benchmark/benchmark.h>

#include <stewkk/ipc/fdstreambuf.hpp>
#include <stewkk/ipc/pipe.hpp>
#include <stewkk/ipc/subprocess.hpp>

namespace stewkk::ipc {

static void BM_Pipe(benchmark::State& state) {
  std::size_t size = state.range(0);
  std::size_t count = (kOverallSize + size - 1) / size;
  auto message = GenerateRandomMessage(size);

  std::string got(message.size(), ' ');

  for (auto _ : state) {
    Subprocess<FDBufIn, FDBufOut> child(
        [&count, &message](FDBufOut out) {
          for (std::size_t i = 0; i < count; ++i) {
            out.sputn(message.data(), message.size());
          }
        },
        Pipe());
    for (std::size_t i = 0; i < count; ++i) {
      child.stdout.sgetn(got.data(), got.size());
    }
  }
}

BENCHMARK(BM_Pipe)->RangeMultiplier(2)->Range(kRangeFrom, kRangeTo);

}  // namespace stewkk::ipc
```

Бенчмарк начинается с блока инициализации. В нем вычисляется число сообщений для передачи, которое зависит от общего количества передаваемых байт разделенного на размер одного сообщения. Также в этом блоке по зафиксированному сиду генерируется случайное содержимое одного сообщения, и заполняется пробельными символами буфер такого же размера, который будет использоваться для чтения сообщений в родительском процессе. Если для данного метода межпроцессной необходимости нужны дополнительные параметры, например путь к файлу FIFO, они также создаются и настраиваются в этом блоке.

После блока инициализации следует цикл, где производятся измерения производительности. В нем сначала создается дочерний процесс, который также в

цикле записывает данные в буфер для отправки через механизм межпроцессной коммуникации. В каждой итерации внешнего цикла передается 40’000 байт, согласно глобальной конфигурации.

После создания подпроцесса, родительский процесс в аналогичном цикле считывает 40’000 байт из буфера межпроцессной коммуникации.

Каждая итерация цикла для измерения производительности регистрирует время выполнения этой самой итерации. Алгоритм инструмента `google/benchmark` в начале выполняет несколько холостых итераций (без замеров времени выполнения). Это делается для «прогрева» кэшей процессора, что позволяет снизить шум в результатах бенчмарков. Последующие итерации выполняются с замерами времени и повторяются до тех пор, пока не наступит общий дедлайн по вермени выполнения теста производительности.

Исходя из результатов замеров подсчитываются статистические данные, такие как медиана и среднее время выполнения итерации бенчмарка. Полученные данные выводятся в терминал, а также сохраняются в формате `JSON`.

Для уменьшения шума в результатах бенчмарков согласно рекомендациям LLVM[11] и документации [12] `google/benchmarks` были сделаны следующие настройки:

— сборка в режиме Release, использование статических версий библиотек. Команда для сборки представлена на листинге 8;
— отключена рандомизация адресного пространства (листинг 9);
— отключено динамическое изменение частоты ядер процессора (листинг 9);
— по-максимуму остановлены сторонние процессы в системе, например браузер.

Листинг 8: Сборка проекта и запуск бенчмарков

```
cmake .. -D CMAKE_BUILD_TYPE=Release -DBUILD_SHARED_LIBS=OFF -D
↪  BASE_LINK_FLAGS='-static-libstdc++;-static-libgcc'
cmake --build .
./bin/benchmarks --benchmark_min_warmup_time=5 --benchmark_repetitions=5
↪  --benchmark_out=out.json --benchmark_out_format=json
```

Листинг 9: Настройки системы для бенчмарков

```
sudo cpupower frequency-set --governor performance
echo 0 > /proc/sys/kernel/randomize_va_space
echo 1 > /sys/devices/system/cpu/intel_pstate/no_turbo
```

Таким образом, для большинства тестов разброс значений не превысил 0.5%.

Результаты в виде `JSON` затем были обработаны с помощью скрипта на языке Python (листинг 16). Данный скрипт использует библиотеку pandas для преобразования данных в нужный формат, а затем строит график с помощью библиотеки matplotlib.

График с результатами замеров представлен на рисунке 3. На оси абсцисс отмечены размеры передаваемых пакетов в байтах, а на оси ординат среднее время, затраченное на передачу 40’000 байт в миллисекундах.

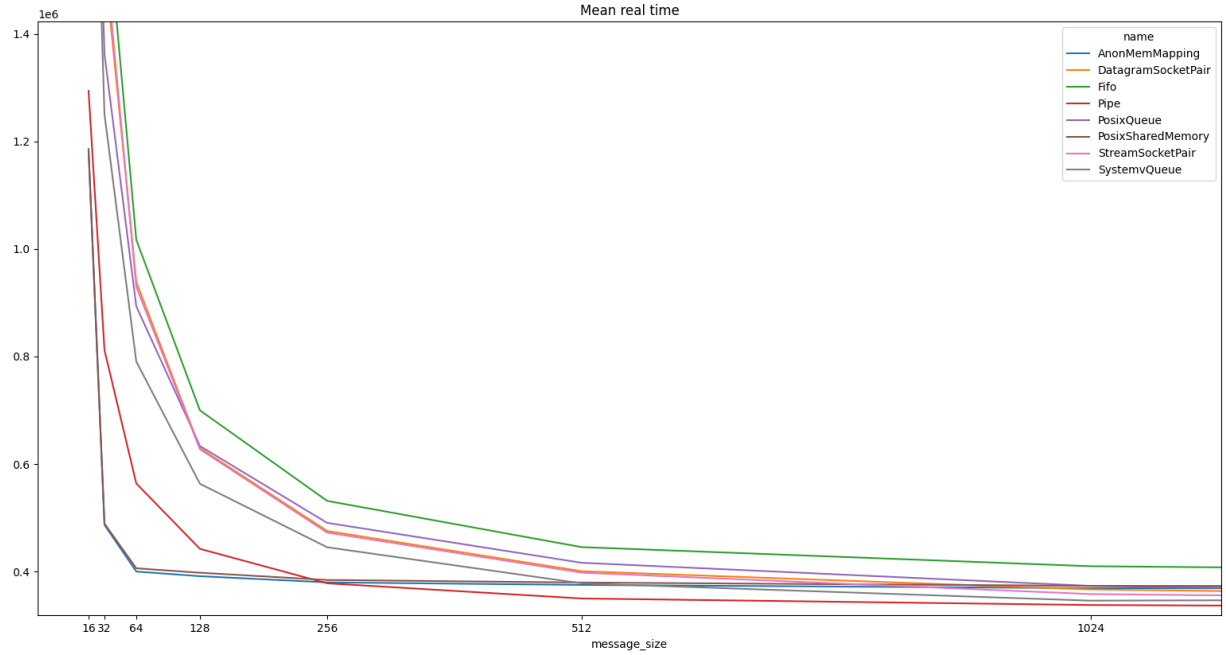

Рисунок 3 — Сравнительный график производительности средств межпроцессной коммуникации.

Как и ожидалось, самую высокую производительность для передачи сообщений небольшого размера показала разделяемая память и анонимная отображаемая память. Этот отрыв обуславливается экономией на системных вызовах и лишних операциях копирования в буфер ядра, которые присущи остальным методам межпроцессной коммуникации.

На больших размерах буферов разницы практически не оказалось, т.к. количество системных вызовов было сведено к минимуму.

Файловые отображения показали плохие результаты на любом размере буфера, т.к. зависят от быстродействия персистентной файловой системы. Поэтому они были намеренно исключены из графика сравнения на этапе подготовки данных.

Стоит упомянуть, что производительность средств межпроцессной коммуникации может меняться от конкретной конфигурации системы и режима нагрузки, и, например, зависеть от версии ядра Linux. Измерения приведенные в данной работе выполнены на версии ядра `6.6.52`, но могут быть легко воспроизведены на любой целевой системе и гибко сконфигурированны под целевой режим нагрузки путем изменения размеров и/или количества передаваемых пакетов.

ЗАКЛЮЧЕНИЕ

В ходе выполнения курсовой работы была разработана библиотека для реализации межпроцессного взаимодействия на базе операционной системы Linux, проведены тесты производительности интерфейсов данной библиотеки и на их основе оценена производительность различных механизмов передачи данных между процессами.

Основной целью данной курсовой работы было проведение сравнительного анлиза средств межпроцессной коммуникации в операцинной системе Linux. Результаты выполнения задачи демонстрируют корректность реализации библиотеки, на основе которой реализованы бенчмарки, что подтверждается автоматизированными юнит-тестами. Разработанные тесты производительности предоставляют гибкий инструмент для сравнения производительности методов межпроцессной коммуникации в различных условяих и режимах нагрузки. Были получены и оценены данные о производительности наиболее релевантных в контексте обмена информацией между процессами интерфейсов.

В заключение, несмотря на достигнутые положительные результаты, существует потенциал для дальнейшего улучшения библиотеки. Это включает в себя тестирование и измерение производительности на основе различных реализаций UNIX систем, а также расширение интерфейса для двунаправленной коммуникации.

СПИСОК ИСПОЛЬЗОВАННЫХ ИСТОЧНИКОВ

1. Исследование рынка серверных операционных систем. — URL: https://www.fortunebusinessinsights.com/server-operating-system-market-106601 ; (дата обращения: 04.12.2024).
2. Опрос разработчиков на ресурсе Stack Overflow. — URL: https://survey.stackoverflow.co/2023/#section-most-popular-technologies-operating-system ; (дата обращения: 04.12.2024).
3. Kerrisk M. The Linux programming interface: a Linux and UNIX system programming handbook. — No Starch Press, 2010. — (дата обращения: 25.11.2024).
4. Референсная документация C++. — URL: https://en.cppreference.com/w/ ; (дата обращения: 15.12.2024).
5. Документация библиотеки GoogleTest. — URL: https://google.github.io/googletest/ ; (дата обращения: 15.12.2024).
6. Библиотека google/benchmark. — URL: https://github.com/google/benchmark ; (дата обращения: 15.12.2024).
7. Проект Doom Emacs. — URL: https://github.com/doomemacs/doomemacs ; (дата обращения: 15.12.2024).
8. Дистрибутив Linux NixOS. — URL: https://nixos.org/ ; (дата обращения: 15.12.2024).
9. Система сборки CMake. — URL: https://cmake.org/ ; (дата обращения: 15.12.2024).
10. Реализация потокового буфера в проекте Poco. — URL: https://github.com/pocoproject/poco/blob/b380b57d5d999444906e42754db13f87902a1794/Foundation/include/Poco/UnbufferedStreamBuf.h ; (дата обращения: 05.12.2024).
11. Рекомендации по запуску бенчмарков от проекта LLVM. — URL: https://llvm.org/docs/Benchmarking.html ; (дата обращения: 21.12.2024).

12. Рекомендации по уменьшению вариативности бенчмарков от проекта google/benchmarks. — URL: https://github.com/google/benchmark/blob/main/docs/reducing_variance.md ; (дата обращения: 21.12.2024).

ПРИЛОЖЕНИЕ А

Листинг 10: Определение класса Subprocess

```
#include <concepts>
#include <cstdlib>
#include <functional>
#include <streambuf>

namespace stewkk::ipc {

template <typename I>
concept StreamBuf = requires(I i) { std::derived_from<I, std::streambuf>;
↪  };

template <typename I>
concept RWIpc = requires(I i) {
  { i.GetReader() } -> StreamBuf;
  { i.GetWriter() } -> StreamBuf;
};

template <StreamBuf InBuf, StreamBuf OutBuf> class Subprocess {
public:
  template <RWIpc Ipc> Subprocess(std::function<void(OutBuf)> subprogram,
  ↪  Ipc ipc);
  ~Subprocess();
  Subprocess(const Subprocess& other) = delete;
  Subprocess& operator=(const Subprocess& other) = delete;

  InBuf stdout;

private:
  pid_t child_pid_;
};

}  // namespace stewkk::ipc
```

Листинг 11: Определение методов класса FdBufIn

```
namespace stewkk::ipc {

FDBufIn::~FDBufIn() {
  if (fd_.has_value()) {
    Close(fd_.value());
  }
}

FDBufIn::int_type FDBufIn::underflow() {
  if (use_buf_) {
    return buf_;
  }
  char c;
  auto ok = read(fd_.value(), &c, sizeof(c));
  if (ok == -1) {
    return EOF;
  }
  if (ok != sizeof(c)) {
    return EOF;
  }
  use_buf_ = true;
  buf_ = c;
  return c;
}

FDBufIn::int_type FDBufIn::uflow() {
  if (use_buf_) {
    use_buf_ = false;
    return buf_;
  }
  char c;
  auto ok = read(fd_.value(), &c, sizeof(c));
  if (ok == -1) {
    return EOF;
  }
  if (ok != sizeof(c)) {
    return EOF;
  }
  return c;
}

std::streamsize FDBufIn::sgetn(char* buf, std::streamsize size) {
  use_buf_ = false;
  return read(fd_.value(), buf, size);
}

}  // namespace stewkk::ipc
```

Листинг 12: Определение методов класса FdBufOut

```
namespace stewkk::ipc {

FDBufOut::FDBufOut(std::int32_t fd) : fd_(fd) {}

FDBufOut::~FDBufOut() {
  if (fd_.has_value()) {
    Close(fd_.value());
  }
}

FDBufOut::FDBufOut(FDBufOut&& other) noexcept :
↪  fd_(std::exchange(other.fd_, std::nullopt)) {}

FDBufOut& FDBufOut::operator=(FDBufOut&& other) noexcept {
  fd_ = std::exchange(other.fd_, std::nullopt);
  return *this;
}

FDBufOut::int_type FDBufOut::overflow(int_type c) {
  if (c == EOF) {
    return EOF;
  }

  auto ok = WriteFD(fd_.value(), c);
  if (!ok) {
    return EOF;
  }

  return c;
}

std::streamsize FDBufOut::sputn(const char* buf, std::streamsize size) {
  auto res = write(fd_.value(), buf, size);
  return res;
}

}  // namespace stewkk::ipc
```

Листинг 13: Реализация GetSyscallError()

```
#include <cerrno>
#include <cstring>
#include <format>

namespace stewkk::ipc {

SyscallError GetSyscallError() {
  auto err = errno;
  return SyscallError(err, std::system_category(), std::format("{}",
  ↪  strerrorname_np(err)));
}

}  // namespace stewkk::ipc
```

Листинг 14: Объявление класса Pipe

```
#include <cstdint>

#include <stewkk/ipc/fdstreambuf.hpp>

namespace stewkk::ipc {

class Pipe {
public:
  using ReadFD = std::int32_t;
  using WriteFD = std::int32_t;

  Pipe();

  FDBufIn GetReader();
  FDBufOut GetWriter();

private:
  ReadFD read_fd_;
  WriteFD write_fd_;
};

}  // namespace stewkk::ipc
```

Листинг 15: Определение методов класса Pipe

```
#include <array>
#include <cstdint>
#include <tuple>
#include <utility>

#include <unistd.h>

#include <stewkk/ipc/errors.hpp>
#include <stewkk/ipc/syscalls.hpp>

namespace stewkk::ipc {

namespace {

std::pair<Pipe::ReadFD, Pipe::WriteFD> MakePipe() {
  std::array<std::int32_t, 2> pipe_file_descriptors;
  if (pipe(pipe_file_descriptors.data()) == -1) {
    throw GetSyscallError();
  }
  return {
      pipe_file_descriptors[0],
      pipe_file_descriptors[1],
  };
}

}  // namespace

Pipe::Pipe() { std::tie(read_fd_, write_fd_) = MakePipe(); }

FDBufIn Pipe::GetReader() {
  Close(write_fd_);
  return FDBufIn(read_fd_);
}

FDBufOut Pipe::GetWriter() {
  Close(read_fd_);
  return FDBufOut(write_fd_);
}

}  // namespace stewkk::ipc
```

Листинг 16: Скрипт для построения графика результатов бенчмарков

```
#!/usr/bin/env python3

import pandas as pd
import numpy as np
import matplotlib.pyplot as plt
import matplotlib
import json
import re

def main():
    matplotlib.use('TkAgg')
    json_data = None
    with open('out.json', 'r') as f:
        json_data = json.load(f)
    df = pd.DataFrame(json_data['benchmarks']).set_index('name')
    df = df[df['aggregate_name'] == 'mean']
    df['message_size'] = df.index
    df['message_size'] = df['message_size'].transform(lambda s:
    ↪  int(re.match('BM_.*\/([0-9]+)_mean', s)[1]))
    df = df.filter(items=['message_size', 'real_time'])
    df.index = pd.DataFrame(df.index)['name'].transform(lambda s:
    ↪  re.match('BM_([^</]*)', s)[1])
    df = df.pivot(columns='message_size', values='real_time').T
    df = df.filter(regex='(?!(^FileMemMapping$))^.*$')

    print(df)
    plot = df.plot(title='Mean real time', xticks=[16, 32, 64, 128, 256,
    ↪  512, 1024, 2048, 4096, 8192])
    plt.show()


if __name__ == "__main__":
    main()
```