Санкт-Петербургский Государственный Университет

Математико-механический факультет

Кафедра системного программирования

Методы взаимодействия прикладного приложения и ядра ОС

Курсовая работа студента 344 группы

Булычева Антона Дмитриевича

Научный руководитель Абусалимов Э.Ш.

Аспирант кафедры системного программирования

Санкт-Петербург

2013

Оглавление

Введение

Вычислительная система, выполняющая ту или иную управляющую задачу, обычно представляет собой целый комплекс программных средств, включающих в себя операционную систему, драйвера устройств, различные службы и прикладные программы. Все это выполняется на конкретной аппаратной платформе. Взаимодействие всех этих компонентов крайне важно, ведь от него во многом зависит эффективность данной системы.

Прикладное приложение – это программа, предназначенная для выполнения определенных пользовательских задач. При разработке программы, решающую целевую задачу, наиболее важно взаимодействие операционной системы и самого прикладного приложения.

Операционная система предоставляет пользовательскому приложению необходимые ресурсы вычислительной системы. Пользовательское приложение общается с операционной системой через какой-либо программный интерфейс, то есть через определенный набор команд, включающий следующие:

- открытие, закрытие, чтение, запись файлов
- создание нового процесса
- завершение процесса
- увеличение кучи

Постановка задачи

В моей курсовой работе я хотел произвести обзор существующих методов взаимодействия прикладного приложения и ядра ОС. Практическую часть работы я проводил на основе открытого проекта по созданию конфигурируемой операционной системы реального времени Embox, поскольку для таких систем параметры эффективности и предсказуемости особенно важны.

Таким образом, в данной курсовой работе передо мной стояли следующие задачи:

- Рассмотреть и реализовать в ОС Embox возможные методы взаимодействия прикладного приложения и ядра ОС
- Реализовать возможность во время сборки выбирать тип взаимодействия с ядром без изменения исходного кода приложения
- Произвести сравнение быстродействия реализованных методов

Обзор

Прикладное приложение при взаимодействии с ядром ОС обычно использует не низкоуровневые интерфейсы, а использует такие как стандартная библиотека языка Си, POSIX, WinAPI и так далее.

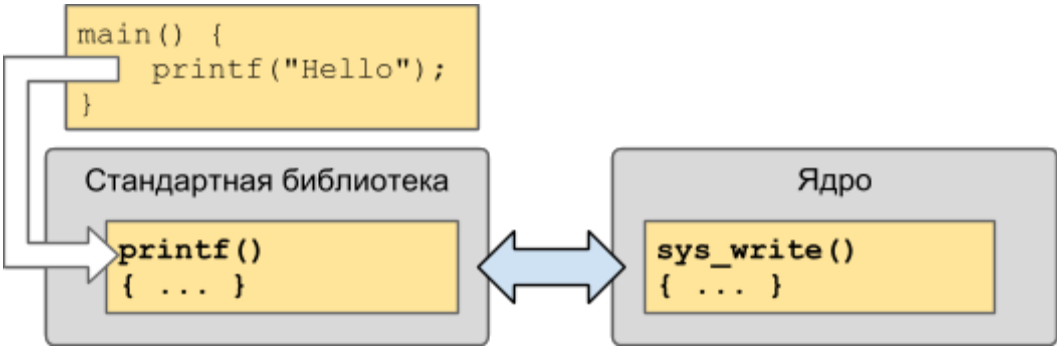
Стандартная библиотека языка Си

В моей курсовой работе рассматривался интерфейс работы с помощью стандартной библиотекой языка Си. Она содержит такие функции как `printf`, `scanf`, `malloc`, `free`, `exit` и т.д.

Существует множество реализаций стандартной библиотеки, поставляемых как с различными операционными системами, так и с компиляторами языка Си. Самые известные из них:

- GNU C Library (glibc) – самая распространенная реализация, используемая в большинстве дистрибутивов Linux. Имеет большой объем исходного кода и большие требования к ресурсам
- EGLIBC – вариант glibc, изначально ориентированный на встраиваемые системы
- BSD libc – реализация для BSD систем, встроенная в операционную систему и поддерживаемая общим репозиторием исходных кодов
- diet libc – реализация для встраиваемых систем. Основная цель проекта – создание максимально легких приложений
- Newlib – реализация, предназначенная для использования во встраиваемых системах. Представляет собой объединение нескольких библиотек. Легка в портировании на новые системы

Схематично взаимодействие приложения, стандартной библиотеки и ядра ОС можно представить следующим образом:



Приложение всегда линкуется вместе со стандартной библиотекой. Поэтому определяющим становится то, как взаимодействуют стандартная библиотека с ядром ОС. Существуют различные методы взаимодействия, имеющие свои преимущества и недостатки, и выбор того или иного зависит от конкретного случая. Речь об этих методах и пойдет дальше.

Исполнение в режиме пользователя

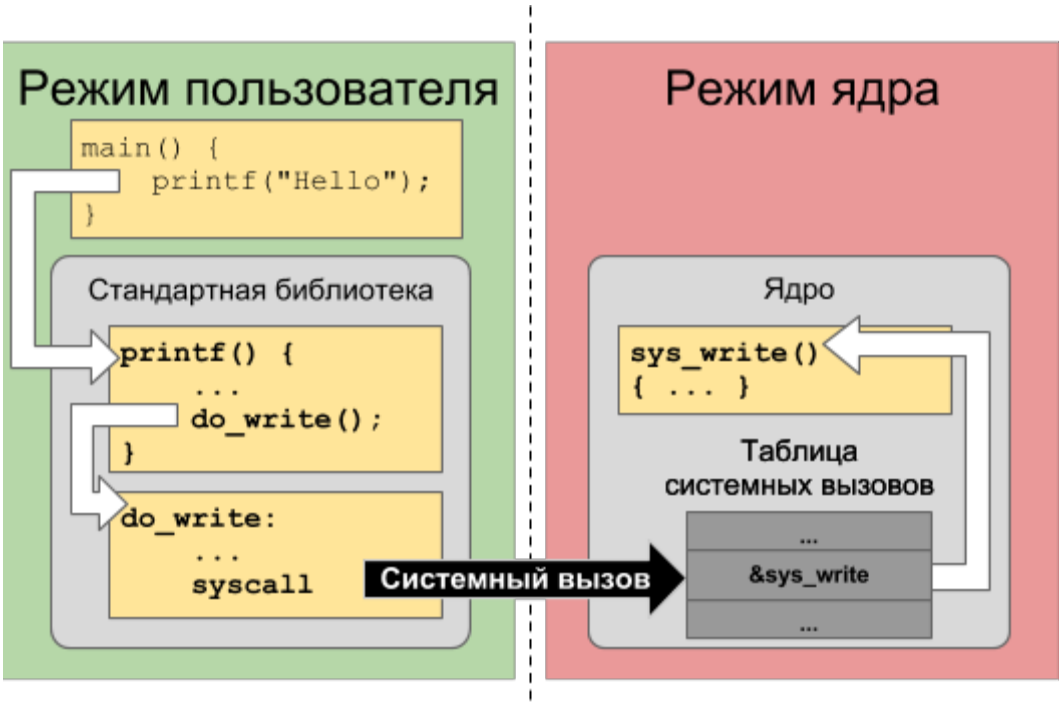
Большинство современных процессоров имеет два режима работы: привилегированный (режим ядра) и защищенный (режим пользователя). Их отличие заключается в том, что в пользовательском режиме нельзя:

- исполнять привилегированные функции процессора (например, включать и отключать прерывания)
- изменять системные регистры
- обращаться к защищенным страницам памяти
- читать и писать из защищенных портов компьютера

При таком разделении код операционной системы исполняется в режиме ядра, а код прикладного приложения – в режиме пользователя. Для взаимодействия используется механизм системных вызовов. Системный вызов является исключением, которое генерирует прикладная программа.

Взаимодействие осуществляется следующим образом. Каждой системной функции назначают свой номер, таким образом формируя таблицу системных вызовов. Когда прикладному приложению необходимо выполнить некоторую привилегированную функцию, он записывает на регистры процессора номер функции и ее параметры и с помощью системного вызова

(специальной инструкции процессора) передает управление обработчику системных вызовов, который является частью операционной системы и исполняется в режиме ядра.



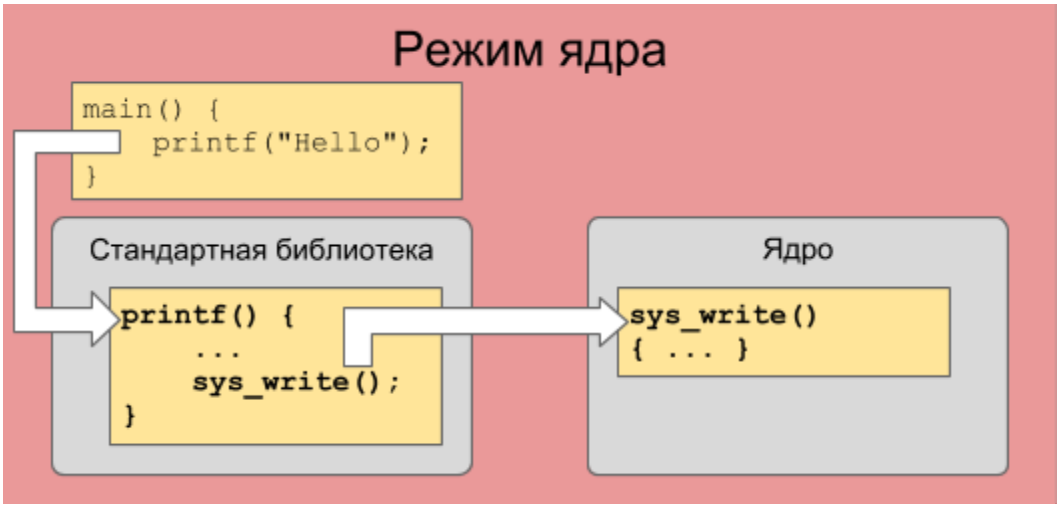
При использовании такого метода взаимодействия обеспечивается безопасность, и приложение не может вывести из строя систему. Также разработчику легче вести отладку: операционная система сразу же оповестит программу о выполнении ей недопустимой операции.

Но помимо преимуществ, этот метод имеет и недостатки. Один из них – наличие накладных расходов на осуществление системных вызовов. Кроме того, в случае встроенных систем, нацеленных на исполнение какого-то одного конкретного приложения (функционального ПО), отказ ФПО равносилен отказу всей системы, поэтому в данном случае преимущества безопасности не играют роли.

Исполнение в режиме ядра

Первый метод не всегда является допустимым, так как в системах с ограниченными ресурсами может не хватать памяти для кода, который обеспечивает возможность осуществления системных вызовов, а также для самой таблицы системных функций. В таких случаях используется другой метод, заключающийся в том, что приложение исполняется вместе с ядром операционной системы в одном режиме – режиме ядра. При таком подходе приложению не

нужно делать системный вызов, чтобы перейти из одного режима в другой, в отличие от предыдущего метода. Вместо этого оно совершает вызов обычной функции, и в данном случае необходимо знать ее адрес в памяти, а не номер. Для получения нужных адресов приложение можно слинковать вместе с ядром, тогда все адреса будут разрешены на этапе компиляции.



Главным недостатком данного метода является его небезопасность. Но при использовании данного метода исчезают накладные расходы на переход из одного режима в другой. Менее очевидными преимуществами являются локальность кэша и страниц памяти и большие возможности компилятора для оптимизации.

Реализация

На момент начала работы в ОС Embox был реализован только метод статической линковки функций ядра и пользовательских приложений. В рамках курсовой работы мною был реализован метод взаимодействия на основе системных вызовов.

Статическая линковка функций ядра

ОС Embox имеет внутреннюю реализацию стандартной библиотеки, используемую самим ядром. По умолчанию приложения линкуются напрямую с ядром ОС. Данный метод является самым простым из существующих методов. При этом методе пользовательское приложение собирается совместно с ядром ОС Embox, тем самым ядро ОС предоставляет приложению свою реализацию стандартной библиотеки. В этом случае адреса всех необходимых функций с их сигнатурами доступны на этапе сборки конечного образа системы.

Использование системных вызовов

Для реализации взаимодействия на основе системных вызовов мной была использована реализация стандартной библиотеки Newlib. Это наиболее популярная библиотека для встраиваемых систем. Ее преимущество заключается в том, что она легко может быть портирована под новую ОС и платформу, и объектный код после статической линковки с приложением получается сравнительно небольшим.

Библиотека Newlib требует реализации ряда системных вызовов, представленных в следующей таблице. Эти системные вызовы затем используются в реализации функций стандартной библиотеки (например, функция `write` используется в функции `printf`). Каждой системной функции был назначен номер.

№	Системная функция
1	`void **_exit**(int status)`
3	`ssize_t **read**(int fd, void *buf, size_t count)`
4	`ssize_t **write**(int fd, const void *buf, size_t count)`
5	`int **open**(const char *pathname, int flags, mode_t mode)`
6	`int **close**(int fd)`
45	`int **brk**(void *addr)`
108	`int **fstat**(int fd, void *buf)`

Когда прикладному приложению необходимо сделать системный вызов, оно помещает на регистры номер функции и передаваемые параметры, а затем генерирует программное прерывание. Осуществление системного вызова на платформе x86 происходит следующим образом:

1. Прикладная программа сохраняет:
    - номер системной функции на регистре `EAX`
    - параметры системной функции на регистрах `EBX`, `ECX`, `EDX`, `ESX`, `EDI`
2. Прикладная программа генерирует программное прерывание с помощью инструкции `INT` X, где X – номер прерывания
3. Обработчик прерывания с номером X сохраняет регистры на стек, находит в таблице системных вызовов адрес функции с номером, указанном на регистре `EAX`, и передает управление по этому адресу
4. Системная функция выполняет свою работу и по соглашению о вызовах сохраняет результат на регистр `EAX`
5. Обработчик прерываний восстанавливает все регистры, кроме `EAX`, и выходит из прерывания, тем самым возвращая управление прикладному приложению
6. Прикладное приложение получает результат выполнения системной функции, считывая

его из регистра `EAX`

Для простоты отладки номер прерывания (0x80) и номера системных функций были взяты из ядра ОС Linux. Это позволяет запускать в ОС Linux приложение, скомпилированное для ОС Embox.

Метод переключения

ОС Embox имеет высокую конфигурируемость, которая позволила добавить возможность задавать в конфигурационных файлах тип взаимодействия приложения и ядра ОС. При использовании вышеописанных подходов в обоих случаях код прикладного приложения и код модулей ядра не изменяются. В конфигурационных файлах задается лишь модуль, который реализует тот или иной тип взаимодействия, поэтому изменяется только набор объектных файлов для линковки и некоторые внутренние заголовочные файлы.

Конфигурационный файл для реализации слоя системных вызовов в ОС Embox выглядит следующим образом:

```
package embox.kernel

module syscall {
	source "syscall.c"

	depends embox.arch.syscall
	depends embox.kernel.syscall.syscall_table
}
```

Теперь, чтобы задать способ взаимодействия с помощью системных вызовов, необходимо в файл mods.config добавить строчку:

```
include embox.kernel.syscall
```

Измерения

Основное отличие методов заключается в том, что в одном случае системная функция вызывается напрямую, а в другом с помощью системного вызова. Это отличие неизбежно влечет за собой разницу в быстродействии двух подходов; величину этой разницы я попытался

оценить на практике.

Для того, чтобы сравнить быстродействие двух методов, в ядре ОС Embox была написана пустая функция, которая возвращает единственный принимаемый ей параметр:

```
int sys_func(int arg) {
     return arg;
}
```

Прикладное приложение выглядело следующим образом:

```
int main(void) {
     for (int i = 0; i < NCYCLES; i++) {
          trace_block_enter(&tb);
          sys_func(1);
          trace_block_leave(&tb);
     }
}
```

Для того, чтобы измерения были точными, приложение запускалось с выключенными прерываниями. В качестве тестового окружения был выбран симулятор QEMU для платформы x86. Измерения производились с отключением аппаратной виртуализации. Это было необходимо для того, чтобы QEMU полностью эмулировал ЦПУ и периферию, обеспечивая достоверность измерений.

Платформа x86 содержит два таймера: Programmable Interval Timer (PIT) и Time Stamp Counter (TSC). В своей курсовой работе я использовал TSC по следующим причинам:

- PIT имеет низкую точность
- Счетчик PIT быстро переполняется
- TSC имеет точность до такта

Объем выборки (NCYCLES) брался равным 10^5, 10^6 и 10^7. Вычислялись минимальное и среднее количества тактов, требуемых для одного вызова. Полученные результаты мало отличались от

запуска к запуску, что говорит о стабильности измерений. Следующая таблица содержит результаты измерений:

	Минимальное количество тактов	Среднее количество тактов
Без системных вызовов	**308**	**399**
С системными вызовами	**1343**	**1584**
Разница	**1035**	**1185**

Таким образом, я получил количественную оценку накладных расходов на осуществление системного вызова на платформе x86 в ОС Embox.

Результаты

В ходе работы над курсовой я получил следующие результаты:

- Рассмотрены два различных способа взаимодействия прикладного приложения и ядра ОС. Реализован метод взаимодействия на основе системных вызовов
- Реализована возможность при сборке выбирать тип взаимодействия с ядром ОС без изменения исходного кода приложения
- Произведены сравнения быстродействия двух реализаций

Список литературы

1. glibc – http://www.gnu.org/software/libc/
2. EGLIBC – http://www.eglibc.org/home
3. diet libc – http://www.fefe.de/dietlibc/
4. Newlib – http://sourceware.org/newlib/
5. Intel 64 and IA-32 Architectures Software Developer's Manual
6. How to Benchmark Code Execution Times On Intel IA-32 and IA-64 Instruction Set Architectures