Домашнее задание 5. Пространство пользователя и системные вызовы

Я

April 24, 2016

Contents

1 Основное задание

В этом домашнем задании вам необходимо добавить к вашей ОС пространство пользователя и предоставить интерфейс для взаимодействия между пространством пользователя и ядром ОС.

Для выполения этого задания вам нужно выполнить следующие небольшие подзадачи:

1. загрузка и запуск исполняемого файла формата ELF из initramfs в нижнюю часть адресного пространства процесса;
2. реализация интерфейса системных вызовов, системный вызов write, который будет печатать переданное из пространства пользователя сообщения в последовательный порт;
3. реализация системного вызовова fork; системный вызов создает точную копию процесса; важно что он создает именно копию, новый процесс не разделяет общую память пространства пользователя с родительским процессом;

Обратите внимание, что второе задание можно выполнить не выполняя полностью первое и последнее. Например, вы можете запустить код скомпонованный вместе с ядром в непривелигированном режиме и тестировать системные вызовы в этом коде (задание без теста не принимается).

2 Загрузка и запуск ELF файла

В задании вам не нужно уметь загружать и запускать любые файлы формата ELF, достаточно, если вы сможете предоставить исходники и скрипт для сборки ELF файла, который вы можете запустить.

Сама по себе загрузка ELF не сложный процесс, если у вас уже есть файловый интерфейс, функции аллокации страниц и функции для работы с таблицей страниц;

Все что вам нужно, это прочитать таблицу програмных заголовков, найти те из них, тип которых PT_LOAD, настроить в таблице страниц отображение, которое покроет диапазон от p_vaddr до $p\_vaddr + p\_memsz$ и скопировать в этот диапазон память участок файла начиная со смещения p_offset и до $p\_offset + p\_filesz$. На этом загрузка файла заканчивается. Определения нужных структур и констант вы можете найти в файле *elf.h* в папке *src*.

Запуск этого файла в пространстве пользователя требует меньше кода, но может быть концептуально более сложной задачей. Для этого вам необходимо создать на стеке структуру, которая будет прочитана из памяти инструкцией *iret*, которую мы будем использовать для передачи управления в пространство пользователя.

Эта структура должна включать в себя селекторы стека и селекторы кода пользовательского пространства, указатель стека и указатель команд, а также регистр флагов. Все они будут записаны инструкией *iret* в соответствующие регистры процессора и таким образом управление будет переданно в пространство пользователя.

В качестве селекторов стека и кода используйте *USER_DATA* и *USER_CODE* из файла *memory.h*. А в качестве указателя комманд используйте точку входа указанную в заголовке ELF файла. Указатель стека остается на ваше усмотрение, например, вы можете возложить настройку стека на пользовательское приложение, или в ядре выделить память под стек где-нибудь вверху пользовательского адресного пространства (ниже дыры в каноническом адресном пространстве).

3 Системные вызовы

Самый простой способ (но не единственный) организовать системные вызовы в x86 это использовать перрывания. Для этого вам необходимо проделать несколько простых шагов. О каждом подробнее далее.

3.1 Дескриптор прерывания в IDT

Первая и самая простая часть - завести дескриптор *IDT* для нашего будущего системного вызова. Вы можете выбрать любую неиспользуюмую запись *IDT* для этого.

Дескриптор системного вызова почти совпадает с дескриптором обычного прерывания, с той лишь разницей, что значение поля *DPL* должно быть равно 3, в противном случае системный вызов будет приводить к *Genral Protection Error*.

Кроме этого, обычно, на время выполнения системного вызова не нужно выключать прерывания, поэтому для системных вызовов зачастую используют *trap gate* вместо *interrupt gate*.

3.2 Обработчик системного вызова

Ествественно вы должны также предоставить обработчик прерывания, который по параметрам системного вызова разебертся, чего от него хочет пространство пользователя. Поэтому вам нужно продумать вомзожность передвать параметры в системный вызов, а также уметь возвращать из него значение (например, если вы реализуете системный вызов *fork*).

Самый простой способ передвать параметры в системный вызов - использовать регистры. Например, вы можете зафиксировать за регистром *RAX* номер прерывания, по которому ядро узнает, какой именно системный вызов запросило пространство пользователя, а в регистре *RBX* передвать указатель на структуру, которая описывает параметры системного вызова.

Конкретный интерфейс остается на ваше усмотрение, но важно, чтобы система не падала, если из пространства пользователя придет неверный номер системного вызова (проверка параметров, по хорошему тоже нужна, но она не так трививальна, как кажется на первый взгляд) - в этом случае вы должны вернуть какой-то признак ошибки. Самый простой способ возвращать значения из системного вызова - регистры.

Соответсвенно, в вашем приложении, которое будет запущено в пространстве пользователя должен быть функция, которая знает как и в каком виде нужно передавать параметры в системный вызов, и как получать результат.

3.3 Task State Struct

Структура состояния задачи в 64-битном режиме выглядит довольно просто, она содержит указатели стека для всех уровней привелегий кроме низшего, остальные поля нас не интересуют и их можно заполнить нулями. Формат можно посмотреть в разделе 7.7 TASK MANAGEMENT IN 64-BIT MODE документации Intel.

Мы собираемся использовать только два уровня привелегий - 0 и 3. Поэтому нас на самом дере интересует только одно поле этой структуры - запись соответсвующая указателю стека ядра. В эту запись мы, скорее всего, хотим записать вершину стека ядра выделенного при создании потока. Т. е. при системном вызове в *RSP* запишется указатель вершины стека, который вы аллоцировали для фашего потока при создании.

При переключении потоков, нам достаточно поменять это поле внутри *TSS* и больше никаких действий касающихся *TSS* при переключении потоков производить не требуется.

3.4 Запись TSS в GDT

Для нашего *TSS* в *GDT* требуется завести дескриптор, этот дескриптор, должен хранить адрес и размер нашей *TSS*, а также некоторые служебные поля, которые нам не инетерсны, но заполнить их всеравно придется.

Если вы пользуетесь предоставленным вам кодом, то определение *GDT* находится в файле *bootstrap.S*, вы можете убедиться, что там зарезервированы записи в конце таблицы как раз под *TSS*.

Все, что вам нужно это получить указатель на эту таблицу и заполнить соответствующую запись нужными данными. Чтобы получить указатель на *GDT* используйте функцию *get_ gdt_ ptr* из файла *memory.h*.

Теперь касательно полей, которые вам нужно заполнить. Формат дескриптора вы можете, опять же найти в документации в разделе 7.2.3 TSS Descriptor in 64-bit mode. Частью жтого формата является тип записи. Все возможные типы перечислены в разделе 3.5 SYSTEM DESCRIPTOR TYPES все той же документации Intel - найти нужный не состаит труда.

В качестве значений *Base* и *Limit* используйте виртуальный адрес *TSS* и и ее размер минус единица, бит гранулярности, соответственно, должен быть сброшен, так как размер указывается в байтах.

Наконец в качестве значения *DPL* нужно указать наименьший уровень привелегий, т. е. 3.

После того, как вы заполнили запись нужными значениями, вам необходимо загрузить селектор этой записи в регистр *TR*, для этого воспользуйтесь функцией *load_ tr* из файла *memory.h* (хотя эта функция отностися скорее к потокам, чем к памяти). На этом все мучения с *TSS* заканчиваются, вам остается лишь подменять одно поле в *TSS* при переключении потоков.

4 Вопросы и ответы