Operating Systems

Userspace

Me

April 19, 2016

План лекции

- Userspace и системные вызовы.
- Исполняемые файлы.
- Динамические библиотеки.

Userspace vs kernelspace

- kernelspace (о нем вы уже знаете):
  - kernelspace код использует наивысший приоритет исполнения (на x86 $CPL == 0$);
  - kernelspace память общая для всех процессов
    - часть таблиц страниц процессов ссылаются на одну и ту-же память;
  - имеет неограниченный доступ ко всем ресурсам;

Userspace vs kernelspace

- kernelspace (о нем вы уже знаете):
  - kernelspace код использует наивысший приоритет исполнения (на x86 $CPL == 0$);
  - kernelspace память общая для всех процессов
    - часть таблиц страниц процессов ссылаются на одну и ту-же память;
  - имеет неограниченный доступ ко всем ресурсам;
- userspace (о нем вы еще не знаете):
  - userspace код использует низший приоритет исполнения (на x86 $CPL == 3$);
  - userspace память своя у каждого процесса (с оговорками);
  - для доступа к ресурсам должен спросить kernelspace;
  - обычно приложения работают в userspace, чтобы они не мешали друг другу;

Приоритет исполнения

- Userspace код одного процесса не должен вредить другим процессам, например:
  - userspace код не может менять таблицу страниц;
  - userspace код не может менять GDT (специфично для x86);

Приоритет исполнения

- Userspace код одного процесса не должен вредить другим процессам, например:
  - userspace код не может менять таблицу страниц;
  - userspace код не может менять GDT (специфично для x86);
- Userspace не может выполнять некоторые инструкции:
  - такие инструкции называют привилегированными;
  - это инструкции изменяющие состояние системы;
  - например, *lgdt* или *mov* в регистр *cr3*;

Userspace vs kernelspace memory

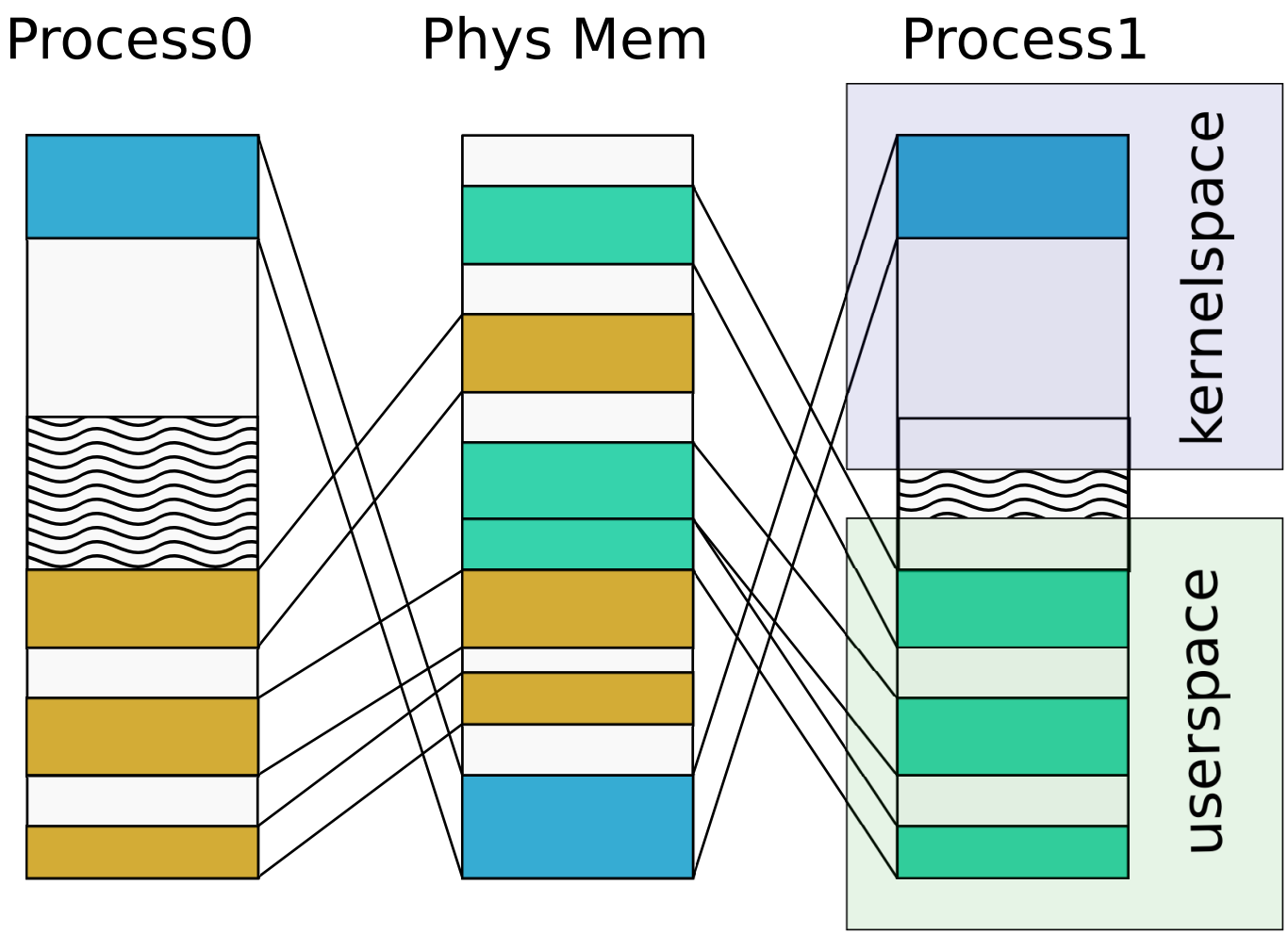

Figure : Kernelspace vs Userspace Memory

Передача управления между userspace и kernelspace

- Userspace сильно ограничен в возможностях, но он должен обращаться к ядру ОС
  - нужна возможность передать управление в kernelspace;
  - нельзя просто разрешить перейти в kernelspace;
  - передать управление в kernelspace можно только определенному доверенному коду;
  - доверенный код обрабатывает запрос от userspace кода;

Передача управления между userspace и kernelspace

- Userspace сильно ограничен в возможностях, но он должен обращаться к ядру ОС
  - нужна возможность передать управление в kernelspace;
  - нельзя просто разрешить перейти в kernelspace;
  - передать управление в kernelspace можно только определенному доверенному коду;
  - доверенный код обрабатывает запрос от userspace кода;
- Есть и обратная задача - передать управление из kernelspace в userspace:
  - ядро ОС запускается в привилегированном режиме работы процессора;
  - после обработки запроса в kernelspace, мы должны как-то вернуть управление в userspace;

Системные вызовы

- Системный вызов - интерфейс передающий управление между userspace и kernelspace кодом:
  - userspace код сохраняет данные запроса в определенных (создателями ОС) местах, например, в регистрах;
  - userspace код делает системный вызов (мы поговорим об это дальше);
  - системный вызов производит переключение режима работы процессора и вызывает код ядра ОС (нужна аппаратная поддержка);
  - код ядра ОС сохраняет все что нужно сохранить (например, volatile регистры) и обрабатывает запрос;

Реализация системных вызовов в x86

- Помните о прерываниях?
  - при возникновении прерывания, управление передается обработчику прерывания ядра;
  - прерывания можно генерировать специальной командой *int*;
  - выделим неиспользуемый номер прерывания под системные вызовы;
  - возврат из системного вызова - *iret*;

Смена привилегий в x86

- Если прерывание сгенерировано инструкцией *int*, то процессор проверят поле *DPL* дескриптора прерываний;
- *CPL* (тот что в регистре *CS*) должен быть меньше или равен *DPL* в дескрипторе;
- таким образом userspace код может вызвать только системный вызов;
- для прерываний сгенерированных устройствами и исключений эта проверка не выполняется;

Переключение стека в x86

- Обычно мы хотим, чтобы обработчик прерывания пользовался своим стеком, а не userspace стеком
  - мы не хотим, чтобы userspace код мог потом прочитать со стека какие-то данные ядра ОС;
  - мы не знаем, сколько стека использовал userspace и сколько осталось нам;

Переключение стека в x86

- Обычно мы хотим, чтобы обработчик прерывания пользовался своим стеком, а не userspace стеком
  - мы не хотим, чтобы userspace код мог потом прочитать со стека какие-то данные ядра ОС;
  - мы не знаем, сколько стека использовал userspace и сколько осталось нам;
- Поэтому, если обработчик прерывания прерывает userspace код нужно сменить стек:
  - для этого (и многого другого) в x86 используется *TSS*;

Task State Segment

- В структуре *TSS* хранится указатель стека, который будет загружен в *RSP*;
- обычно, в *TSS* записывают вершину стека ядра потока исполнения и обновляют значение при переключении потоков;
- в *GDT* должна быть специальная запись описывающая *TSS*;
- в специальный регистр *TR* должен быть записан селектор *TSS* дескриптора в *GDT*;
- формат *TSS* описан в *7.7 TASK MANAGEMENT IN 64-BIT MODE* документации *Intel*

Исполняемые файлы

- Откуда берется код в userspace?
  - этот код может быть скомпилирован вместе с кодом ядра ОС;
  - ядро ОС может загрузить этот код из файла;

Исполняемые файлы

- Откуда берется код в userspace?
  - этот код может быть скомпилирован вместе с кодом ядра ОС;
  - ядро ОС может загрузить этот код из файла;
- Чтобы ОС могла загрузить исполняемый файл, он должен соответствовать определенному формату:
  - формат определяет ОС, но есть несколько распространенных;
  - например, ELF, PE, Mach-O;
  - они похожи, но как всегда детали отличаются;

Исполняемые файлы

- Формат исполняемого файла должен предоставить следующую базовую информацию:
  - какие части файла по каким адресам в памяти нужно положить;
  - какие участки памяти какими данными нужно заполнить;
  - по какому адресу передать управление (упрощенно, где в памяти *main*);
  - или указать на того, кто знает как работать с исполняемым файлом (например, добавить #! и путь к интерпретатору в начало файла);

Формат ELF

```
struct elf_hdr {
    uint8_t e_ident[ELF_NIDENT];
    uint16_t e_type;
    uint16_t e_machine;
    uint32_t e_version;
    uint64_t e_entry;
    uint64_t e_phoff;
    uint64_t e_shoff;
    uint32_t e_flags;
    uint16_t e_ehsize;
    uint16_t e_phentsize;
    uint16_t e_phnum;
    uint16_t e_shentsize;
    uint16_t e_shnum;
    uint16_t e_shstrndx;
} __attribute__((packed));
```

- *ELF* файл начинается с заголовка с общей информацией;
- тип, версия, архитектура - ОС должна проверить, что они соответствуют ее ожиданиям;
- заголовок хранится адрес точки входа - *e_entry*;

Program Header

- *Program Header*, упрощенно, описывает участок памяти, который должна подготовить ОС:
  - некоторые из них ОС должна прочитать из файла и сохранить в памяти;
  - для некоторых просто должна быть выделена и инициализирована память;
  - некоторые имеют служебное значение и могут быть проигнорированы;

Program Header

- *Program Header*, упрощенно, описывает участок памяти, который должна подготовить ОС:
  - некоторые из них ОС должна прочитать из файла и сохранить в памяти;
  - для некоторых просто должна быть выделена и инициализирована память;
  - некоторые имеют служебное значение и могут быть проигнорированы;
- Заголовок указывает на таблицу *Program Header*-ов:
  - *e_phoff* - смещение таблицы в файле;
  - *e_phnum* - количество заголовков в таблице;
  - *e_phentsize* - размер одной записи;

Program Header

```
struct elf_phdr {
        uint32_t p_type;
        uint32_t p_flags;
        uint64_t p_offset;
        uint64_t p_vaddr;
        uint64_t p_paddr;
        uint64_t p_filesz;
        uint64_t p_memsz;
        uint64_t p_align;
} __attribute__((packed));
```

- *p_type* - тип заголовка, нас интересует только тип *PT_LOAD* (значение 1);
- *p_vaddr* и *p_memsz* - адрес и размер в памяти;
- *p_offset* и *p_filesz* - смещение и размер в файле;
- *p_filesz* может быть меньше *p_memsz*;

Загрузка ELF файла

- Проверяем заголовок:
  - в *e_ident* хранятся магическая последовательность, класс и порядок байт;
  - правильная магическая последовательность: "\x7f""ELF";
  - класс *ELF_CLASS64* (значение 2);
  - порядок байт *ELF_DATA2LSB* (значение 1);
  - кроме того полезно проверить поле *e_type*, там должно быть значение *ELF_EXEC* (значение 2);
  - подробности можно найти в спецификации *ELF* и в *ABI*;

Загрузка ELF файла

- Проходимся по *Program Header*-ам и подготавливаем память:
  - положение *Program Header*-ов указано в заголовке, который мы проверили;
  - для каждого *Program Header* с типом *PT_LOAD* нужно подготовить участок памяти по адресу *p_vaddr* размера *p_memsz*;
  - в этот участок памяти нужно прочитать *p_filesz* байт из файла по смещению *p_offset*;

Загрузка ELF файла

- Проходимся по *Program Header*-ам и подготавливаем память:
  - положение *Program Header*-ов указано в заголовке, который мы проверили;
  - для каждого *Program Header* с типом *PT_LOAD* нужно подготовить участок памяти по адресу *p_vaddr* размера *p_memsz*;
  - в этот участок памяти нужно прочитать *p_filesz* байт из файла по смещению *p_offset*;
- Подготовка стека:
  - зачастую выделять стек для процесса - задача ОС (это определяют создатели ОС);
  - обычно после загрузки приложению нужно передать какие-то параметры (командная строка, переменные окружения и прочее);
  - часто их просто копируют на стек, выделенный ОС;

Динамическая компоновка

- Библиотеки, упрощенно, разделяют на статические и динамические:
  - статические библиотеки компонуются вместе с исполняемым файлом;
  - динамические библиотеки не являются частью исполняемого файла, а загружаются по требованию;

Динамическая компоновка

- Библиотеки, упрощенно, разделяют на статические и динамические:
  - статические библиотеки компонуются вместе с исполняемым файлом;
  - динамические библиотеки не являются частью исполняемого файла, а загружаются по требованию;
- Зачем нужны динамические библиотеки:
  - не нужно дублировать один и тот же код между множеством исполняемых файлов;
  - можно избежать дублирования одного и того же кода в памяти загрузив его в единственном экземпляре;

Динамическая компоновка

- Динамическая библиотека может быть загружена по любому адресу
  - исполняемый файл не может заранее знать, по какому адресу находятся функции и данные библиотеки;
  - библиотека сама не знает по какому адресу находятся ее функции и данные;

Динамическая компоновка

- Динамическая библиотека может быть загружена по любому адресу
  - исполняемый файл не может заранее знать, по какому адресу находятся функции и данные библиотеки;
  - библиотека сама не знает по какому адресу находятся ее функции и данные;
- Таким образом нужно решить несколько задач:
  - загрузить все необходимые динамические библиотеки в память;
  - сообщить приложению адреса нужных функций и данных;
  - сделать так, что библиотека могла быть загружена и работать по любому адресу;

Динамический компоновщик

- В *ELF* файле может быть специальный *Program Header* с типом *PT_INTERP* (значение 3)
  - этот заголовок описывает участок файла где хранится строка;
  - эта строка - путь к динамическому компоновщику;
  - ОС загружает динамический компоновщик вместе с исполняемым файлом и передает ему управление;

Динамический компоновщик

- В *ELF* файле может быть специальный *Program Header* с типом *PT_INTERP* (значение 3)
  - этот заголовок описывает участок файла где хранится строка;
  - эта строка - путь к динамическому компоновщику;
  - ОС загружает динамический компоновщик вместе с исполняемым файлом и передает ему управление;
- Динамический компоновщик должен найти и загрузить нужные библиотеки:
  - информация о динамических библиотеках хранится в *Program Header* с типом *PT_DYNAMIC*;
  - но ОС это не волнует - дальше всю работу делает динамический компоновщик;

Обращение к данным

- Программа (или динамическая библиотека) может не знать где хранятся нужные данные до загрузки
  - для разрешения проблемы используется *Global Offset Table (GOT)*;
  - внешним переменным сопоставляются записи в *GOT*;
  - динамический компоновщик знает адрес *GOT* и заполняет ее правильными значениями;
  - код программы (библиотеки) тоже должен найти *GOT* для этого используется относительная адресация;

Обращение к данным

- Программа (или динамическая библиотека) может не знать где хранятся нужные данные до загрузки
  - для разрешения проблемы используется *Global Offset Table (GOT)*;
  - внешним переменным сопоставляются записи в *GOT*;
  - динамический компоновщик знает адрес *GOT* и заполняет ее правильными значениями;
  - код программы (библиотеки) тоже должен найти *GOT* для этого используется относительная адресация;
- Относительная адресация:
  - мы можем не знать адрес *GOT*;
  - но мы можем знать, что она находится на определенном смещении от исполняемого кода;
  - возьмем указатель команд *RIP* (для x86) и прибавим к нему смещение;

Вызов функций

- Как и сданными, программа (или библиотека) может не знать адреса функции до загрузки
  - компоновщик (не динамический), создает таблицу *Procedure Linkage Table (PLT)*;
  - каждая запись в *PLT* это переход по адресу записанному в *GOT*;

Вызов функций

- Как и сданными, программа (или библиотека) может не знать адреса функции до загрузки
  - компоновщик (не динамический), создает таблицу *Procedure Linkage Table (PLT)*;
  - каждая запись в *PLT* это переход по адресу записанному в *GOT*;
- Ленивое разрешение имен:
  - динамический компоновщик не обязан заполнять все записи в *GOT* соответствующие функциям;
  - можно организовать так, чтобы при вызове функции вызывался динамический компоновщик;
  - тогда можно разрешать имена и загружать библиотеки только, если к ним было обращение;