Лабораторная работа №3

Теоретический материал

Цель лабораторной работы состоит в написании кода менеджера памяти для учебной ОС.

Первым компонентом менеджера памяти является распределитель физической памяти для ядра. Наш распределитель будет работать со *страницами*, чей размер составляет 4096 байт. В рамках лабораторной работы нужно описать структуры данных, которые будут содержать информацию о свободных и занятых страницах памяти, о том, сколько процессов могут использовать каждую выделенную страницу, а также создать функции для выделения и освобождения страниц памяти.

Второй компонент менеджера памяти – это *виртуальная память*, которая отображает виртуальные адреса, используемые ядром и пользовательским ПО, на адреса в физической памяти. Блок управления памятью (Memory Management Unit (MMU)) выполняет это отображение с помощью множества таблиц страниц, когда команды используют память. В ходе выполнения лабораторной работы нужно будет модифицировать код учебной ОС JOS для настройки таблиц страниц в соответствии с заданием.

Для выполнения лабораторной работы скачайте с сайта дисциплины архив lab3.tar.gz. И добавьте находящиеся в нем файлы в проект JOS. Архив имеет следующую структуру:

- `lab3`
  - `inc`
    - `memlayout.h`
  - `kern`
    - `kclock.c`
    - `kclock.h`
    - `pmap.c`
    - `pmap.h`

Файлы должны быть добавлены в каталоги `inc` и `kern` проекта JOS.

Файл `memlayout.h` описывает формат виртуального адресного пространства, которое нужно будет реализовать, изменив файлы `pmap.c`. В `memlayout.h` и `pmap.h` определена структура `PageInfo`, которую нужно использовать для отслеживания свободных страниц физической памяти. `kclock.c` и `kclock.h` работают с часами и CMOS RAM, в которую BIOS записывает в том числе и размер физической памяти ПК. `pmap.c` считывает эту информацию.

Обратите особое внимание на файлы `memlayout.h` и `pmap.h`, так как для выполнения лабораторной работы понадобятся многие определения, которые в них содержатся. Файл `inc/mmu.h` также содержит много определений, которые будут полезны при выполнении этой лабораторной работы.

Управление физическими страницами

Операционная система должна отслеживать, какие части физической памяти свободны, а какие в настоящий момент заняты. JOS управляет физической памятью *постранично*, поэтому она может использовать MMU для отображения и защиты каждого элемента выделенной памяти.

В этой части лабораторной нужно написать распределитель страниц физической памяти. Он отслеживает свободные страницы с помощью связного списка объектов структурного типа `struct PageInfo`, каждый из которых соответствует одной странице физической памяти.

Задание 1

В файле `kern/pmap.c` нужно реализовать следующие функции:

- `boot_alloc()`
- `mem_init()` (только до вызова `check_page_free_list(1)`)
- `page_init()`
- `page_alloc()`
- `page_free()`

Функции `check_page_free_list()` и `check_page_alloc()` предназначены для тестирования распределителя страниц физической памяти. Для проверки нужно загрузить JOS и посмотреть, выводит ли функция `check_page_alloc()` сообщение «`check page alloc succeeded!`».

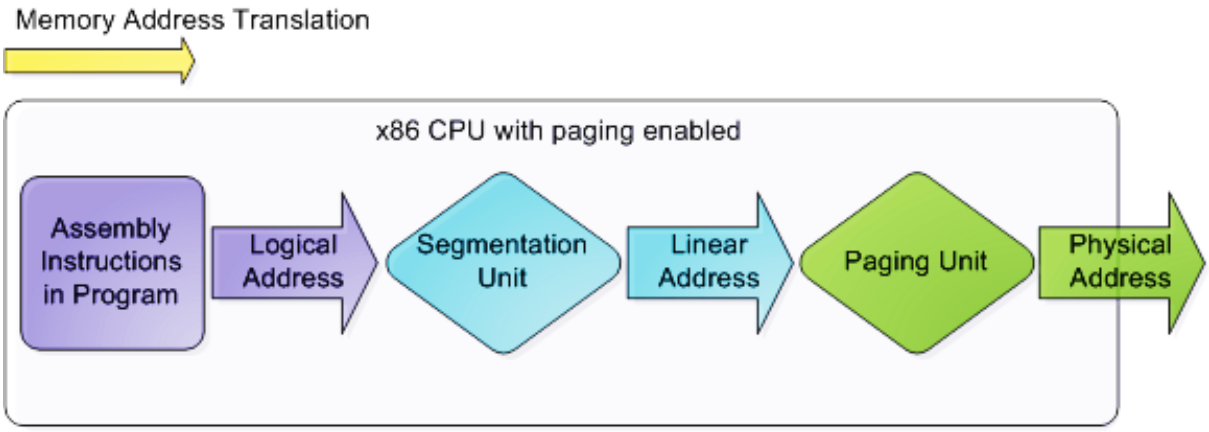
Обратите внимание на комментарии в исходном коде JOS, который Вам нужно модифицировать. Они часто содержат подсказки и описание того, что нужно сделать.

Виртуальная память

Для выполнения лабораторной работы необходимо изучить, как происходит управление памятью в защищенном режиме работы процессора x86. Для этого можно воспользоваться справочным материалом из Intel 80386 Reference Manual (главы 5 и 6, особенно разделы 5.2 и 6.4).

Виртуальный, линейный и физический адреса

Виртуальный адрес состоит из селектора сегмента и смещения внутри сегмента. *Линейный адрес* – это адрес, получаемый после сегментной трансляции, но до страничной трансляции. *Физический адрес* – это результирующий адрес, который получается после сегментной и страничной трансляций. Именно этот адрес выдается на адресную шину для доступа к памяти.



В файле `boot/boot.S` вводится глобальная таблица дескрипторов (Global Descriptor Table (GDT)) и тем самым отключается сегментная трансляция. Значение всех базовых сегментных адресов устанавливается в `0`, а границы сегментов – в `0xffffffff`. В результате значение селектора сегмента не играет никакой роли, а линейный адрес всегда равен смещению виртуального адреса. Таким образом, при трансляции адресов памяти можно игнорировать сегментацию и сфокусироваться полностью на страничной трансляции.

При выполнении лабораторной работы №2 была настроена простая таблица страниц так, чтобы ядро могло запускаться со своего связующего адреса `0xf0100000`. Эта таблица страниц отображала только 4 Мб памяти. В текущей лабораторной работе нужно расширить таблицу страниц так, чтобы она отображала первые 256 Мб физической памяти, начиная с виртуального адреса `0xf0000000`.

GDB имеет доступ к памяти QEMU только по виртуальным адресам. Для того чтобы исследовать физическую память при настройке виртуальных адресов, можно воспользоваться командами встроенного монитора QEMU[1]. Например, командой `xp`, которая позволяет просматривать физическую память. Для входа в режим монитора нажмите последовательность клавиш `Ctrl-a c` в терминале (повторное нажатие этой последовательности клавиш позволит выйти из режима мониторинга).

В версии QEMU, используемой в этой лабораторной работе реализована команда `info pg`. С помощью этой команды можно получить компактное и в то же время детальное представление таблиц текущих страниц, включая все отображенные диапазоны адресов памяти, разрешения и флаги.

Ядру JOS часто бывает необходимо работать с адресами как с непрозрачными (абстрактными, opaque) или как с целыми, не разыменовывая их, например, при распределении физической памяти. Иногда это виртуальные адреса, а иногда физические. Тип данных `uintptr_t` используется для представления виртуальных адресов, а `physaddr_t` – физических. Оба эти типа являются просто синонимами типа данных `uint32_t`.

Ядру JOS иногда необходимо читать или изменять области памяти, для которых оно знает только физические адреса. Например, добавление отображения в таблицу страниц может потребовать выделения физической памяти для хранения каталога страниц, а затем

[1] В Приложении 2 приведены некоторые команды встроенного в QEMU монитора.

инициализации этой памяти. Тем не менее, ядро, как и любое другое ПО, не может избежать трансляции виртуальной памяти, а значит не может напрямую загружать и сохранять данные по физическим адресам в памяти. Одна из причин, по которой JOS отображает всю физическую память, начиная с физического адреса `0`, на виртуальный адрес `0xf0000000`, состоит в том, чтобы помочь ядру читать и писать в области памяти, для которых оно знает только физические адреса. Для того чтобы транслировать физический адрес в виртуальный, с которым ядро сможет работать, ядро должно добавить `0xf0000000` к физическому адресу. Для выполнения этого сложения нужно использовать `KADDR(pa)`.

Также иногда ядру нужно иметь возможность найти физический адрес по данному виртуальному адресу, по которому хранится в памяти какая-нибудь структура данных ядра. Глобальные переменные ядра и память, выделенная функцией `boot_alloc()`, находятся в области, где загружено ядро, начиная с `0xf0000000`, как раз в той области, куда отображается вся физическая память. Поэтому для преобразования виртуального адреса этой области в физический, ядро просто должно вычесть `0xf0000000`. Для выполнения этого вычитания используйте `PADDR(va)`.

Подсчет ссылок на страницы памяти

Количество ссылок на каждую физическую страницу хранится в поле `pp_ref` структуры `struct PageInfo`, соответствующей этой физической странице. Когда это число становится равным 0 для некоторой физической страницы, эта страница может быть освобождена, так как более не используется. Обычно это число должно быть равно количеству раз, сколько физическая страница появляется *ниже* `UTOP` во всех таблицах страниц (отображения выше `UTOP` в основном выполняются ядром во время загрузки системы и никогда не должны освобождаться, поэтому нет необходимости подсчитывать ссылки на них).

При использовании `page_alloc` нужно иметь в виду, что страница, возвращаемая этой функцией, будет всегда иметь количество ссылок, равное 0. Поэтому `pp_ref` нужно инкрементировать каждый раз, когда выполняются какие-либо действия с возвращаемой функцией `page_alloc` страницей (например, вставка ее в таблицу страниц). Иногда инкремент выполняется другими функциями (например, `page_insert`), а иногда функция, вызывающая `page_alloc`, выполняет это действие сама.

Управление таблицами страниц

Задание 2

Теперь нужно написать набор функций для управления таблицами страниц: вставка и удаление отображений линейного адреса в физический и при необходимости создание страниц для таблицы страниц.

В файле `kern/pmap.c` нужно реализовать следующие функции:

- `pgdir_walk()`
- `boot_map_region()`
- `page_lookup()`

- `page_remove()`
- `page_insert()`

С помощью функции `check_page()`, вызываемой из `mem_init()`, можно протестировать функции управления таблицей страниц.

Адресное пространство ядра

JOS разделяет 32-разрядное линейное адресное пространство процессора на две части. Пользовательские процессы будут управлять распределением и содержанием нижней части этого пространства, а ядро всегда осуществляет полный контроль над верхней частью. Граница определена произвольно и хранится в `ULIM` (файл `inc/memlayout.h`). Для виртуального адресного пространства ядра зарезервировано 256 Мб.

Для справки можно использовать схему распределения памяти JOS, приведенную в файле `inc/memlayout.h`.

Права доступа и локализация ошибок

Для обеспечения доступа пользовательского кода только к пользовательской части адресного пространства будем использовать биты прав доступа в таблицах страниц. Иначе ошибки в пользовательском коде могут привести к перезаписи данных ядра, что в свою очередь повлечет за собой сбой системы или менее заметную неисправность.

Пользовательские процессы не будут иметь прав доступа к области памяти, расположенной выше значения `ULIM`, а ядро будет иметь доступ на чтение и запись в эту область памяти. Для диапазона адресов `[UTOP,ULIM]` у ядра и пользовательских процессов будут одинаковые права доступа: только на чтение. Этот диапазон адресов используется для того, чтобы предоставить пользовательским процессам возможность чтения некоторых структур данных ядра. И наконец адресное пространство, расположенное ниже `UTOP`, предназначено для использования пользовательскими процессами. Сами процессы будут устанавливать права доступа к этой области памяти.

Инициализация адресного пространства ядра

Теперь настроим адресное пространство выше `UTOP`: адресное пространство ядра. В файле `inc/memlayout.h` показано распределение памяти, которое нужно использовать. Для осуществления соответствующих преобразований линейных адресов в физические нужно использовать функции, написанные Вами ранее.

Задание 3

1. Добавьте недостающий код в функцию `mem_init()` после вызова `check_page()`. Используйте вызовы `check_kern_pgdir()` и `check_page_installed_pgdir()` для проверки.
2. Мы разместили области памяти ядра и пользовательского процесса в одном и том же адресное пространство. Почему пользовательские процессы не могут читать или писать в область памяти ядра? Какие специальные механизмы защищают адресного пространство ядра?

3. Каков максимальный размер физической памяти, с которым может работать операционная система JOS? Почему?

Приложение 1. Установка QEMU для выполнения 3-ей лабораторной работы

1. Скачайте архив qemu_lab3.tar.gz с сайта кафедры, разархивируйте в свой домашний каталог.
2. Сконфигурируйте исходный код:

```
Linux: ./configure --disable-kvm [--prefix=PFX]
OS X: ./configure --disable-kvm --disable-sdl [--prefix=PFX]
```

   Опция `prefix` указывает, куда будет установлен QEMU, по умолчанию будет выполнена установка в каталог `/usr/local`.

3. Выполните команду `make && make install`

Приложение 2. Команды монитора QEMU

QEMU включает в себя монитор для отслеживания и модификации состояния машины. Для входа в режим мониторинга нажмите `Ctrl-a c` в терминале с запущенным QEMU. Для выхода из режима мониторинга нажмите `Ctrl-a c` еще раз.

Полный список команд монитора приведен в QEMU manual. Ниже перечислены только некоторые команды:

`xp/Nx paddr`

Выводит шестнадцатеричные значения `N` слов, начиная с физического адреса `paddr`. По умолчанию `N` равно 1. Это аналог команды `x` в GDB.

`info registers`

Вывод полного содержимого внутренних регистров машины, включая скрытое состояние селекторов сегментов, локальную (LDT) и глобальную (GDT) таблицы дескрипторов и регистр задачи. Виртуальный ЦП считывает скрытое состояние из GDT/LDT, когда загружает селектор сегмента. Ниже приведено содержимое `CS` при работе ядра JOS и значения каждого поля:

```
CS =0008 10000000 ffffffff 10cf9a00 DPL=0 CS32 [-R-]
```

```
CS =0008
```

Видимая часть селектора сегмента. Значение селектора сегмента равно `0x8`. Это означает, что мы ссылаемся на глобальную таблицу дескрипторов (`0x8&4=0`), текущий уровень привилегий (CPL, current privilege level) равен `0x8&3=0`.

```
10000000
```

Начало этого сегмента. `Linear address = logical address + 0x10000000`.

```
ffffffff
```

Граница этого сегмента. Линейные адреса более `0xffffffff` вызовут исключение «нарушение границы сегмента».

`10cf9a00`

Флаги для этого сегмента, значения которых QEMU декодирует в следующих нескольких полях.

`DPL=0`

Уровень привилегий для этого сегмента. Только процесс с уровнем 0 может загрузить этот сегмент.

`CS32`

Это 32-разрядный сегмент кода. Другие возможные значения: `DS` (сегмент данных) и `LDT` (локальная таблица дескрипторов).

`[-R-]`

Сегмент доступен только для чтения.

`info mem`

Выводит отображенные области виртуальной памяти и права доступа к ним. Например,

```
ef7c0000-ef800000 00040000 urw
efbf8000-efc00000 00008000 -rw
```

показывает, что `0x00040000` байт памяти с адреса `0xef7c0000` до `0xef800000` отображены с правами доступа на чтение и запись для пользовательских процессов, а область памяти с адреса `0xefbf8000` до `0xefc00000` отображена с правами доступа на чтение и запись для ядра.

`info pg`

Выводит структуру текущей таблицы страниц. Вывод идентичен команде `info mem`, но различает записи в каталоге страниц и таблице страниц и выводит сведения о правах доступа к ним отдельно. Повторяющиеся записи в таблице страниц и таблицы страниц в целом «сворачиваются» в одну строку. Например,

```
VPN range     Entry         Flags        Physical page
[00000-003ff] PDE[000]      -------UWP
[00200-00233] PTE[200-233] -------U-P 00380 0037e 0037d 0037c 0037b 0037a ...
[00800-00bff] PDE[002]      ----A--UWP
[00800-00801] PTE[000-001] ----A--U-P 0034b 00349
[00802-00802] PTE[002]      -------U-P 00348
```

Здесь показаны две записи в каталоге страниц (PDE), охватывающие виртуальные адреса с `0x00000000` до `0x003fffff` и с `0x00800000` до `0x00bfffff` соответственно. Обе записи присутствуют в памяти (флаг `P`), доступны для записи (флаг `W`) и имеют пользовательский уровень привилегий (флаг `U`). Ко второй записи был осуществлен доступ (флаг `A`). Одна из таблиц страниц (PTE) охватывает виртуальные адреса с `0x00800000` по `0x00802fff`. Первые две страницы этой таблицы присутствуют в памяти, имеют пользовательский уровень привилегий и к

ним был осуществлен доступ. Третья страница присутствует в памяти и имеет пользовательский уровень привилегий.

QEMU также поддерживает аргументы командной строки, которые могут быть переданы в Makefile учебной операционной системы JOS с помощью переменной QEMUEXTRA:

```
make QEMUEXTRA='-d int' ...
```