Домашнее задание 1. Последовательный порт, прерывания и таймер.

Я

February 19, 2016

Contents

1 Основное задание

Это домашнее задание является вводным, в нем вам необходимо настроить последовательный порт, контроллер прерываний и интервальный таймер.

1. Инициализировать контроллер последовательного порта и создать функцию записи в последовательный порт. Мы будем использовать последовательный порт вместо экрана.

2. Настроить программируемый контроллер прерываний i8259 [INT, b] (такой раньше использовался в персональных компьютерах, но сейчас его не используют, ему на замену пришел APIC, который поддерживает обратную совместимость).

3. Настроить программируемый интервальный таймер [INT, a] на прерывания через равные интервалы времени. В обработчике прерывания вам нужно выводить на последовательный порт сообщение (текст сообщения не существенен, но перед тем как расходиться, помните, что у проверяющего может отсутствовать чувство юмора и шутка может плохо закончится).

2 Дополнительные задания

- В целях отладки очень часто полезно видеть backtrace, т. е. как программа пришла к той или иной строчке кода. Задание заключается в том чтобы написать функцию, которая выводит backtrace в последовательный порт в виде набора адресов. Используйте эту функцию в обработчике исключений. При этом запрещается инструментировать функции, т. е. нельзя модифицировать функции, чтобы они самостоятельно добавляли себя в backtrace. Backtrace не должен быть совершенно точным. Построение backtrace не должно приводить к ошибкам (будет странно, если при выводе сообщения об ошибке при выводе backtrace-а вы полезите не в свою память, что в свою очередь может привести к другой ошибке).

- Реализуйте функции семейства *printf (а именно printf, vprintf, snprintf и vsnprintf). Функция должна поддерживать как минимум следующие спецификаторы формата: d, i, u, o, x, c, s, p, и следующие модификаторы размера: hh, h, l, ll, z [CPP,]. При

реализации нельзя ограничивать размер вывода одного вызова printf/vprintf каким-то фиксированным размером, т. е. нельзя просто реализовать {v}snprintf и вызвать ее из {v}printf передав буффер фиксированного размера.

3 Программируемый контроллер прерываний

В рамках курса мы будем работать со старым семейством программируемых контроллеров прерываний представленных чипами Intel из серии 8259 (8259, 8259a и 8259b). Эти чипы уже давно не используются в реальном оборудовании поэтому мы будем ссылаться на них как Legacy PIC. К счастью для нас, современные контроллеры прерываний [WIK,] поддерживают интерфейс Legacy PIC, поэтому все что вы напишите можно будет запустить и на реальном железе, а не только в эмуляторе.

Почему мы будем пользоваться Legacy PIC вместо более современных APIC? Причин две:

- Legacy PIC проще (потому что менее функциональный), соответственно быстрее можно получить работающее решение;
- чтобы использовать APIC вам все равно сначала придется настроить Legacy PIC, т. е. работу с Legacy PIC все равно пропустить нельзя.

Как и для любой другой электронной схемы для Legacy PIC есть спецификация: [INT, 1988]. Однако полезность этой спецификации очень ограничена. Она описывает контакты контроллера прерываний и их назначение, но так как этот контроллер прерываний можно подсоединить к любому процессору[1] она не описывает интерфейс для обращения к контактам контроллера. Другими словами, прочитав эту спецификацию вы буде знать какие команды передавать контроллеру прерываний, но не будете знать как именно это сделать. Таким образом чтобы работать с Legacy PIC вам нужно знать как он подключается и какой интерфейс для доступа к его выходам.

[1] по крайней мере не только к процессорам семейства x86

3.1 Подключение

На самом деле, в персональных компьютерах использовался не один чип, а два, соединенных в каскад (Legacy PIC можно было соединять в каскады и из большего числа контроллеров). Один из двух чипов называется Master, а другой, соответственно Slave. Выход Slave чипа подсоединяется к второму входы (считая с нуля) Master чипа. Master чип соединен напрямую с процессором.

Всего у каждого чипа по 8 входов для устройств, один из входов Master чипа занят выходом Slave чипа, т. е. остается 15 свободных входов для подключения внешних устройств к контроллерам прерываний. 15 внешних устройств это, на самом деле, очень мало для современных компьютерных систем, поэтому некоторые входы контроллеров прерываний разделяются между несколькими устройствами, но мы не будем касаться этой темы.

3.2 Сигнализация и обработка аппаратных прерываний

Теперь когда мы знаем как Legacy PIC подключен к процессору рассмотрим чуть подробнее что происходит, когда устройству требуется внимание и забота со стороны процессора и оно генерирует прерывание.

Допустим для определенности, что, например, программируемый интервальный таймер (PIT, о котором мы еще поговорим дальше) генерирует сигнал на входе IRQ0, к которому он подключен (мы используем сквозную нумерацию входов: IRQ0-IRQ15, где IRQ0-IRQ7 входы Master чипа, а IRQ8-IRQ15 входы Slave чипа). Контроллер прерываний устанавливает соответствующий бит во внутреннем регистре Interrupt Request Register в 1. После чего контроллер прерываний проверяет другой внутренний регистр Interrupt Mask Register, чтобы проверить, что прерывание не было замаскировано[1].

Если прерывание не было замаскировано, то нужно убедиться что нет более приоритетных прерываний, которые ждут своей очереди[2]. Если более приоритетных прерываний нет, то сигнал об ожидающем обработки прерывании передается процессору.

Что должен делать процессор чтобы принять прерывание? В первую очередь, процессор завершает выполнение текущей

[1] замаскированные прерывания ждут, не замаскированные можно передавать процессору

[2] контроллер прерываний среди прочего выполняет арбитраж прерываний, т. е. определяет очередность обработки прерываний

инструкции[1]. Затем процессор проверяет состояние бита разрешения прерываний специального флагового регистра процессора[2]. Если бит разрешения прерываний установлен, то процессор подтверждает получение прерывания подав сигнал на специальный вход контроллера прерываний. После этого контроллер прерываний выставляет на своих выходах номер вектора обработчика сработавшего прерывания.

Важно отметить, что номер вектора прерывания определяет какой обработчик должен быть вызван, чтобы обработать прерывание. И этот номер вектора может не совпадать[3] с номером прерывания. Как номера прерываний отображаются на номера векторов прерываний определяется во время инициализации контроллера.

Далее контроллер прерываний выставляет бит в еще одном внутреннем регистре: In Service Register[4]. Этот бит будет сброшен, когда обработчик прерывания передаст специальную команду контроллеру прерываний. У процессора к этому времени есть вся информация, которая нужна, чтобы обработать прерывание. К тому как имеенно обрабатывается прерывание на стороне процессора мы вернемся после того, как поговорим как инициализировать контроллер прерываний.

3.3 Настройка контроллера прерываний

Настройка контроллера прерываний осуществляется с помощью внутренних регистров контроллер прерываний. Доступ к этим регистра осуществляется через порты ввода/вывода.

Я уже упоминал выше несколько регистров:

- Interrupt Request Register - установленный бит в этом регистре говорит, что есть прерывание ожидающее подтверждения со стороны процессора;
- In Service Register - установленный бит в этом регистре говорит, что прерывание было подтверждено процессором и ожидается специальная команда от обработчика прерывания, до тех пор пока эта команда не будет послана контроллеру прерываний все прерывания того же или меньшего приоритета не будут сигналиться процессору;

[1] это не нужно воспринимать буквально, например, что будет текущей инструкцией, если процессор использует конвейер?

[2] его называют FLAGS, EFLAGS или RFLAGS в зависимости от режима

[3] и скорее всего не будет совпадать

[4] не трудно догадаться по названию регистра, что значит этот бит

- Interrupt Mask Register - позволяет "замаскировать" некоторые прерывания, чтобы контроллер прерываний не сигнализировал о них, т. е. другими словами мы можем используя этот регистр выключать отдельные аппаратные прерывания;

Кроме упомянутых регистров есть еще несколько:

- Command Register - этот регистр описывает команду, которую вы хотите передать контроллеру прерываний (сколько в ней будет байт, например);
- Status Register - этот регистр только для чтения, так что не нужен при настройке контроллер прерываний, поэтому мы его опустим;
- Data Register - в этот регистр мы записываем данные (например, данные нужные согласно команде, которую мы записали в Command Register).

Как уже было отмечено для обращения к этим регистрам используются порты ввода/вывода, в частности, для обращения к Command Register и Status Register Master контроллера используется порт 0x20[1]. Interrupt Maske Register и Data Register для Master контроллера отображены на порт 0x21[2]. Для Slave контроллера используются порты 0xA0 и 0xA1.

Что мы должны сделать, чтобы сконфигурировать контроллер прерываний? Мы должны указать ему в какой конфигурации они работают: кто Master, а кто Slave и как они соединены[3], кроме того мы должны указать как отображать номера прерываний на номера векторов прерываний.

Конфигурация происходит посредством записи во внутренние регистры "командных слов"[4]. Первое командное слово является главным, записывается в Command Register (т. е. в порты 0x20 и 0xA0) и определяет количество и смысл дальнейших слов, которые будут записаны в Data Register (т. е. в порты 0x21 и 0xA1).

Типичное первое командное слово при инициализации контроллера состоит из следующих бит (наименее значимый справа): 0b00010001. Смысл интересных нам бит следующий:

[1]когда вы читаете из него - вы читаете значение Status Register, а когда пишите - вы пишите в Command Register

[2]тут все несколько сложнее, потому что в оба регистра можно писать

[3]это необходимо, так как Legacy PIC может работать в разных конфигурациях, хотя нас интересует только одна из множества возможных

[4]часто когда используют термин "слово" подразумевают 16 бит, в данном случае это не так, под словом подразумевается 1 8-ми битный байт

- bit 0 - единица в этом бите означает, что команда состоит из 4 слов (первое слово + 3 слова данных);
- bit 1 - единица в это бите значит, что в системе только один контроллер прерываний, а ноль, что несколько контроллеров прерываний соединены в каскад (конфигурация этого каскада определяется 3-им командным словом);
- bit 4 - должен быть установлен в 1 для команды инициализации контроллера.

Второе командное слово определеяет отображение номеров прерываний на номера векторов прерываний. Legacy PIC не очень умен, поэтом отображение осуществляется очень просто: $VECTOR_no = IRQ_no + ICW_2$, где ICW_2 это значение второго командного слова. В архитектуре x86 (точнее в некоторых режимах работы процессоров данной архитектуры) первые 32 вектора прерывания заняты исключениями или зарезервированы, т. е. для нормальной работы ICW_2 должен быть 32 или более. Типичным значением ICW_2 для Master является 0x20, а для Slave 0x28.

Следующее командное слово определяет конфигурацию каскада. Master контроллеру необходимо передать в в этом слове байт, в котором установлены биты соответствующие входам контроллера соединенным со Slave контроллерами, так как у нас Slave контроллер подключен ко второму входу Master (считая с нуля) вы должны передать туда 0b00000100.

Для Slave контроллера это слово интерпретируется как номер входа Master контроллера, к которому он подключен, то есть в нашем случае это должно быть значение 0b00000010. [1]

Наконец, последнее командное слово определяет режим работы контроллера прерываний. Я не упоминал об этом ранее, но он может работать в нескольких режимах, из которых нас опять же интересует только один, не вдаваясь в смысл всех бит, командное слово должно быть следующим: 0b00000001, где нулевой установленный бит значит, что контроллер прерываний работает в режим 8086 (режим влияет на взаимодействие контроллер и процессора на низком уровне, который нам не особо интересен).

[1] Вопрос на засыпку, а как контроллер узнает каким образом интерпретировать это слово? Мы ведь не сообщаем нигде кто из них Master, а кто Slave? Это для тех кто жаловался на AVR...

3.4 Маскировка прерываний на контроллере

Как уже упоминалось ранее у Legacy PIC есть специальный внутренний регистр, который позволяет избирательно замаскировать прерывания - Interrupt Mask Register. Кроме того я упоминал, что Interrupt Mask Register как и Data Register использует порты 0x21 и 0xa1 Master и Slave контроллеров. Возникает вопрос, а как же тогда писать в Interrupt Mask Register? Тут все довольно просто, если запись в порт была частью команды[1], то контроллер интерпретирует запись согласно ожижаниям соответствующей команды, а во всех остальных случаях это будет считаться записью значения в Interrupt Mask Register.

3.5 Команда подтверждения прерывания (EOI)

Вы уже знаете, что обработчик прерывания должен послать специальную команду контроллеру прерываний чтобы сбросить бит в In Service Register и разрешить обработку прерываний с тем же или меньшим приоритетом. Эту команду обычно называют EOI - End Of Interrupt.

EOI существует в двух вариантах: один указывает конкретный номер прерывания на, которое нужно отправить подтверждение, а второй подтверждает последнее (и значит самое приоритетное) просигналенное процессору прерывание. И вы можете использовать любой вариант из них. Но вы должны помнить, что контроллеры прерываний соединены в каскад. Т. е. если устройство подсоединенное к Slave контроллеру сигналит о прерывании, то Slave контроллер сообщает об этом мастеру, через один из его входов и таким образом устанавливается сразу два бита: один в In Service Register Slave контроллера, а один в In Service Register Master контроллера и вы должны сбросить оба[2].

Собственно сама команда записывается в Command Register соответствующего контроллера прерываний и формат этой команды вы можете найти в спецификации под именем OCW2 (Operation Control Word 2).

[1] например, команды инициализации, которую я описывал чуть раньше

[2] другими словами если прерывание пришло на Slave, вы должны отправить подтверждение и на Slave и на Master.

4 Обработка прерываний и таблица векторов прерываний

До сих пор мы касались, в основном, аппаратной части обработки прерываний - контроллера прерываний, теперь мы будем говорить о прерываниях с программной стороны[1].

Вы уже должны знать, что чтобы процессор мог принимать и обрабатывать прерывания вы должны настроить контроллер прерываний, разрешить прерывания на процессоре установив соответствующий бит в флаговом регистре и не забыть в обработчике прерывания выполнить команду EOI. Кроме того вы должны смутно подразумевать, что нужна настроить/создать "вектора" прерываний, о которых я упоминал, но не говорил ничего конкретного. Этот раздел как раз освещает эти моменты, но начну я издалека - с типов прерываний.

4.1 Типы прерываний

Вы уже знаете о прерываниях генерируемых внешними устройствами[2], но кроме них существуют и другие - программные прерывания. Программные прерывания не соответствуют никаким внешним устройствам и соответсвенно никак не связаны с контроллером прерываний.

Программные прерывания можно разделить на два класса[3]:

- прерывания сгенерированные специальной инструкцией (например, int или into), такие прерывания мы будем называть синхронными[4];
- прерывания сгенерированные процессором в случае какой-то исключительной ситуации, например, при делении на ноль, или при попытке обратиться к недоступной памяти или использовать некорректный дескриптор;

Мы оставим первый класс на конец курса, а сейчас поговорим о втором классе. Я упоминал, что архитектура x86 резервирует первые 32

[1] или, если хотите, со стороны процессора

[2] это прерывания, о которых нам сообщает контроллер прерываний

[3] Деление довольно условное, потому что грубо говоря специальной инструкцией мы можем сгенерировать прерывание с любым номером.

[4] потому что они не являются неожиданными для нас, мы сами их вызвали

вектора прерываний, так вот резервирует она их как раз для прерываний второго класса[1].

Прерывания второго класса также делятся на группы:

- ловушки (traps) - генерируются после выполнения какой-то инструкции и позволяют перехватить управление после какого-то события, сделать грязное дело, а потом вернуть управление следующей инструкции;

- ошибки (faults) - соответствуют ошибкам, которые можно поправить, что, в принципе, тоже можно отнести к "грязному делу", но разница между ловушками и ошибками в том, что после обработки ошибки прерванная инструкция перезапускается[2];

- критические ошибки (aborts) - суровая версия ошибки, при которое перезапуск прерванной инструкции может быть невозможен, например, потому что адрес возврата может быть некорректен.

Примерами ошибок являются: деление на ноль, выход за границы[3] или неправильный код операции[4].

Примерами ловушек являются: breakpoint или переполнение.

Примеров критических ошибок немного, наиболее типичный из них носит название double fault, и как не трудно догадаться соответствует возникновению ошибки при попытке вызвать обработчик для другой ошибки (т. е. две ошибки одновременно).

Детальная классификация прерывания доступна в [INT, c] в разделе 6.5, а раздел 6.15 содержит справочную информацию о всех зарезервированных прерываниях.

4.2 Таблица дескрипторов прерываний

Теперь вы знаете какие бывают прерывания кроме аппаратных перейдем ближе к делу и коснемся векторов прерываний. Вектор прерываний

[1]Это не совсем так, архитектура x86 резервирует эти прерывания под специальные нужды или на будущее, просто так получается, что большинство из этих специальных нужд это как раз прерывания из второго класса.

[2]Например, если была сгенерирована ошибка деления на 0, вы можете в обработчике прерывания подправить аргументы команды деления и устранить ошибку, после чего деление будет перезапущено

[3]Это штука позволят проверять что обращение к памяти вышло за границы массива.

[4]Вы передали управление чему-то, что процессор не может декодировать как инструкцию.

это, грубо говоря, информация об обработчике прерывания - другими словами дескриптор обработчика прерывания.

Этот дескриптор по сути содержит адрес функции обработчика прерывания, причем адрес это не просто указатель на функцию, мы также включает в понятие адреса селектор сегмента кода (подробнее про дескрипторы можно прочитать в [INT, c] особенно разделы 2.1.1, 2.1.2, 2.4.1 и 3.4) и необходимый уровень привилегий, который, впрочем, не будет интересовать нас на данном этапе.

Информация о структуре дескриптора обработчика прерывания доступна в [INT, c] в разделах 6.11 и 6.14.1.

Все дескрипторы обработчиков прерываний хранятся в таблица дескрипторов обработчиков прерываний (IDT). Информация о местонахождении этой таблицы хранится в специальном регистре IDTR. Эта информация состоит из двух частей:

- базовый адрес (IDT Base Address) - просто виртуальный адрес начала таблицы[1];
- лимит - 16 битное число, размер таблицы в байтах минус один [2];

Тот самый номер вектора прерывания это ничто иное как номер дескриптора в этой таблице, таким образом если вы отобразили аппаратные прерывания на номера векторов прерываний начиная с 0x20, то в соответствующе дескрипторы IDT вы и должны сохранить информацию о соответствующих обработчиках прерываний.

В домашнем задании вам не нужно настраивать GDT, она уже есть в готовом виде в коде, который вам выдан вместе с заданием, а вот создать IDT вам нужно будет самостоятельно. Для того чтобы настроить IDT вам нужно разобраться, по сути, с двумя вещами:

- формат дескриптора IDT, особенно обратите внимание на типы дескрипторов IDT и чем они друг от друга отличаются;
- как записать в регистр IDTR нужное значение - для этого используется специальная инструкция lidt, вам нужно разобраться как она работает;

[1] учтите что в 64-битном режиме этот адрес 64-битное число, хотя в [INT, c] об этом явно не сказано

[2] Таким образом не обязательно создавать таблицу на все 256 возможных векторов прерываний если они вам не нужны, с другой стороны вы можете создать таблицу в которой поместится больше дескрипторов прерываний, но использованы они не будут

4.3 Обработчик прерывания

Вы уже знаете, что обработчик аппаратного прерывания должен посылать команду EOI контроллеру прерываний, сейчас же мы поговорим о том, что должны делать все обработчики прерываний.

Начнем с простого, вызов обработчика прерывания очень похож на вызов функции, разница лишь в деталях. Вызов функции в архитектуре x86 должен заканчивать специальной инструкцией ret или какой-то из ее вариаций[1]. В этом для вас не должно быть ничего удивительного. Обработчик прерываний же должен заканчиваться инструкцией iret, или в случае 64-битного кода инструкцией iretq.

Кроме того, для вас не должно стать неожиданностью, что обработчик прерывания должен сохранять и восстанавливать регистры, которые он собирается испортить, причем в отличие от обычных функций это касается почти всех регистров [2].

4.4 Включение и выключение прерываний

Вы уже знаете как включать и выключать аппаратные прерывания на контроллере прерываний, но контроллер прерываний это внешнее устройство, несколько более медленное чем процессор, а необходимость включать и выключать прерывания появляется довольно часто[3]. Поэтому если вы хотите выключить все прерывания, то легче запретить прерывания сбросив флаг в флаговом регистре, о котором я упоминал ранее. Для сброса флага разрешения прерываний используется команда cli, а чтобы установить флаг разрешения прерываний (и тем самым включить прерывания на процессоре) используется команда sti.

5 Программируемый интервальный таймер (PIT)

Как и для контроллера прерываний для интервального таймера есть спецификация, которая обладает тем же недостатком для нас, что и спецификация контроллера прерываний [INT, 1993]. Поэтому в данном разделе я расскажу про PIT подробнее.

[1]Не то чтобы прям совсем обязательно, но если вы не делаете совсем что-то изощренное, то скорее всего используете ret или leave.

[2]Для тех кто забыл: обработчик прерывания прерывает основной код не давая ему подготовиться и сохранить состояние нужных ему регистров, а значит это должен сделать сам обработчик прерывания.

[3]Мы вернемся к этому в задании про многопточность

PIT внутри содержит генератор тактов который генерирует сигналы с частотой 1193180 Гц. Выбрав нужный режим работы и коэффициент деления вы можете с его помощью генерировать прерывания через равные небольшие интервалы времени.

Подключен PIT обычно к IRQ0 (т. е. нулевой ноге Master контроллера), а значит, если вы отобразили аппаратные прерывания на вектора начиная с 0x20, то номер вектора прерывания PIT будет 0x20.

К счастью PIT не требует какой-то сложной настройки. Для настройки PIT так же как и для Legacy PIC используются порты ввода/вывода. В частности Control Port имеет номер 0x43 и отвечает за команду и выбор "канала" и Data Port, который для каждого "канала" свой, в частности для нулевого (который вам и предлагается использовать) это порт 0x40.

Упомянутые выше "каналы" вы можете рассматривать как различные таймеры внутри PIT. Всего каналов 3, но два из них условно заняты, а для свободного использования остается только нулевой канал, с которым мы и будем работать.

Вы уже должны примерно представлять как работают различные таймеры в общем, PIT мало чем от них отличается, так что большая часть из того что будет расписано дальше может выглядеть подозрительно знакомой.

Начнем с режимов работы, их 6, как всегда нам нужен только один - Rate Generator [1]. Как не трудно догадаться (по названию и из задания, которое вам нужно сделать) в этом режиме PIT генерирует прерывания через равные интервалы времени.

Коэффициент деления это 16-битное число. И это все что можно сказать про коэффициент деления. Чтобы установить режим работы и коэффициент деления нужно записать в Control Port команду в специальном формате:

- bit 0 - отвечает за выбор формата числа, в котором мы передаем делитель частоты, 0 значит, что используется обычное двоичное кодирование, а 1 значит, что используется BCD[2];
- bits [3:1] - эти 3 бита содержат номер режима работы, для Rate Generator используется значение 2;
- bits [5:4] - определяют какую часть коэффициента деления мы хотим записать, 1 значит, что мы хотим записать только наименее

[1] Вообще говоря нам подойдут два режима из 6 - можно также использовать Square Wave Mode.

[2] Не уверен, что нужно говорить, но вы наверняка не хотите использовать BCD

значимый байт из двух, 2 значит, что мы хотим записать наиболее значимый из двух, ну и наконец 3 значит, что первая операция записи запишет наименее значимый байт, а вторая операция записи запишет наиболее значимый байт [1];

- bits[7:6] - выбирают какой из каналов вы настраиваете, так как мы работаем только с нулевым, то значение этих бит, очевидно, 0;

Далее необходимо записать в Data Port значение коэффициента деления. В зависимости от значения bits [5:4] вам потребуется сделать либо одну запись либо две записи подряд.

6 Последовательный порт

Последний кусочек аппаратной части этого задания - последовательный порт (UART). В персональных компьютерах использовалось несколько разновидностей контроллеров последовательных портов, собирательное название для них "семейство 8250", но на этот раз уже не от компании Intel, а от National Semiconductor. Как и для других интегральных схем для нее имеется документация: [NS:, 1990] о проблемах этой документации применительно к нашим задачам говорить уже излишне.

Последовательный порт может работать в двух режимах: с использованием прерываний и опрашивая контроллер. Вам рекомендуется выбрать второй вариант по нескольким причинам:

- он проще;
- неизвестно в каком контексте вам понадобится выводить что-то в последовательный порт[2];
- нам не нужно (пока) чтение из последовательного порта;
- вы всегда сможете перейти к использованию прерываний в дальнейшем;

[1] Так как нам надо инициализировать PIT только один раз, вы скорее всего должны использовать значение 3.

[2] Возможно вы захотите выводить какие-то логи в контексте, в котором прерывания отключены.

6.1 Конфигурация последовательного порта

По количеству разных внутренних регистров и опций конфигурации последовательный порт заткнет за пояс не один Legacy PIC, а целый каскад из 9 Legacy PIC-ов [1].

Вы уже должны знать из чего, примерно, состоит конфигурация последовательного интерфейса:

- символьная скорость, обычно определяется коэффициентом деления некоторой базовой частоты;
- формат кадра, который включает количество бит данных, стоп бит и проверку четности;
- режим работы контроллера последовательного интерфейса (использовать прерывания или нет, читать или писать и тд) - зависимая от контроллера часть;

Как обычно настройка осуществляется через порты ввода/вывода, которые соответствуют каким-то внутренним регистрам контроллера. В нашем конкретном случае для конфигурации последовательного порта используются порты ввода/вывода начиная с 0x3f8 и до 0x3ff включительно[2].

Пройдемся по порядку по портам и регистрам, на которые они отображены. Почти самый простой регистр это регистр данных. Вы пишите в этот регистр то, что хотите передать и читаете из него то, что получили. Этот регистр использует порт 0x3f8 + 0 (я буду использовать такую нотацию для обозначение отдельных портов ввода/вывода последовательно порта). Этот же порт используется для записи и чтения значения младшего байта коэффициента деления, как определяется на что этот порт указывает в данный момент я расскажу дальше.

За прерывания отвечает Interrupt Enable Register, и для доступа к нему используется порт 0x3f8 + 1. Отдельные биты этого регистра отвечают за генерацию прерываний при тех или иных событиях. Если записать в регистр 0, то контроллер не будет генерировать прерывания. И опять же, тот же самый порт используется для доступа к старшему байту коэффициента деления.

[1] Максимальное количество PIC-ов в каскаде 9: 1 Master и 8 Slave.

[2] На самом деле порты ввода/вывода последовательного порта могут находится и по другим адресам, но не будем усложнять.

Следующий важный для нас регистр это Line Control Register и он использует порт 0x3f8 + 3. Этот регистр выполняет две задачи[1]:

- определяет куда в данный момент указывают порты 0x3f8 + 0 и 0x3f8 + 1;
- определяет формат кадра;

За первую функцию отвечает 7 бит (начиная с 0) регистра LCR, также известный как DLAB (Divisor Latch Access Bit). Если этот бит установлен в 1, то порты 0x3f8 + 0 и 0x3f8 + 1 указывают на младший и старший байты коэффициента деления. В противном случае они указывают на регистр данных и Interrupt Enable Register, соответственно.

Биты 0 и 1 Line Control Register отвечают за количество бит данных в формате кадра. Чтобы использовать 8 бит данных вам нужно чтобы оба бита были выставлены в 1.

Бит 2 отвечает за количество стоп бит, если вам нужен 1 стоп бит, то используйте значение 0.

Биты 3-5 отвечают за проверку четности, если вам не нужна проверка четности, то установите их значение в 0[2].

Наконец Line Status Register позволяет проверить закончилась ли предыдущая передача, или еще нужно подождать перед тем как писать в регистр данных следующий байт. Этот регистр использует порт 0x3f8 + 5. Нас интересует в этом регистре бит 5. Если этот бит установлен, значит можно записать следующий байт, если этот бит сброшен, то нужно ждать.

Гораздо более подробная информация о последовательных портах в персональных компьютерах их различных версиях, особенностях и разных режимах работы доступна в [SER,].

7 Предоставляемый код

Вместе с заданием мы предоставляем вам начальный код и билд скрипт. Вы можете вносить в него любые изменения, пытать его как вам угодно или вообще выкинуть и не использовать. Назначение этого кода упростить вам жизнь, чтобы вам не пришлось сразу и неожиданно писать целую кучу[3] кода на языке ассемблера, с которым вы не знакомы,

[1] абсолютно друг с другом не связанные

[2] Вообще, достаточно установить в 0 только значение 3 бита.

[3] в этом задании вам, скорее всего придется написать какое-то количество ассемблерного кода, но это будет уже не целая куча.

но это не значит, что вы можете вообще не разбираться в этом коде, потому что я не всегда буду таким добрым и когда-нибудь вам придется столкнуться с суровой реальностью.

Пройдемся по списку:

- Makefile - тут трудно что-то добавить, он довольно примитивный, добавлять в него свои файлы не проблема;
- videomem.S - код для работы с видеопамятью, он нужен только для bootstrap.S пока вы не настроили последовательный порт, после этого работа с видеопамятью нам не понадобится;
- bootstrap.S - в этом файле находится точка входа в программу, он отвечает за настройку начальной GDT, инициализацию paging-а (о которой вам не нужно волноваться на данном этапе) и перевод процессора в так называемый long mode (64-битный режим работы), после чего он просто вызывает функцию main (с которой вы и начнете);
- kernel.ld - линкер скрипт, из этого файла вы можете увидеть, что ядро загружается начиная с 1M физической памяти, но рассчитано на работу в "верхах" виртуального адресного пространства (а именно с адреса 0xffffffff80000000), таким образом все нижнее виртуальное адресное пространствто будет свободно для userspace приложений;
- memory.h - файл, который содержит информацию о конфигурации памяти, в частности он содержит используемые начальной GDT селекторы кода и селекторы данных (KERNEL_CODE и KERNEL_DATA, соответственно);
- ioport.h - функции работы с портами ввода/вывода, которые наверняка вам пригодятся в домашнем задании;
- interrupt.h - содержит описание структуры-указателя на IDT и функцию для установки значения IDTR, они тоже могут оказаться полезными;
- kernel_config.h - предполагается, что вы сюда будете добавлять различные опции конфигурации, на данный момент там есть только одна опция, о которой будет рассказано в разделе про использование QEMU;
- main.c - содержит функцию main, которая не делает ничего полезного.

8 Использование QEMU

Проще всего проверять ваше ядро не на реальной машине, а в эмуляторе. Тут у нас есть некоторый выбор опций. ИМХО, самый удобный вариант QEMU. Так что я покажу как пользоваться QEMU для запуска и отладки ядра.

8.1 Запуск ядра в QEMU

Чтобы просто запустить скомпилированное ядро в qemu нужно выполнить следующую команду:

```
qemu-system-x86_64 -kernel kernel
```

- qemu-system-x86_64 - system в названии значит, что мы эмалируем полную систему[1], а x86_64 это наша целевая архитектура;
- опция kernel позволяет указать бинарный файл ядра, бинарный файл должен быть multiboot или linux boot protocol совместимым (в примере выше используется файл с именем kernel из текущего каталога).

Если когда-нибудь вам не будет хватать производительности QEMU, то вы можете использовать опцию –enable-kvm, если у вас достаточно современный процессор (а это почти всегда так) и вы не пользуетесь каким-нибудь изотерическим дистрибутивом linux, то QEMU будет использовать средства аппаратной виртуализации архитектуры x86, что может заметно ускорить выполнение.

Иногда будет полезно указать QEMU какое количество памяти доступно виртуализуемой системе. Это, как минимум, полезно чтобы проверить, что ваше ядро адекватно реагирует на ситуации когда в системе не достаточно памяти или когда ее слишком много[2]. Для этого можно использовать опцию -m. Например:

```
qemu-system-x86_64 -kernel kernel -m 256
```

команда выше говорит, что в системе должно быть 256 мегабайт виртуально памяти. Можно также указывать суффиксы, например:

[1] QEMU позволяет осуществлять эмуляцию на уровне процессов, т. е. вы например можете запустить Linux приложение собранное под ARM под QEMU на x86 и это будет даже условно сносно работать

[2] впрочем слишком много в QEMU все равно не передать.

```
qemu-system-x86_64 -kernel kernel -m 1g
```

теперь мы требует 1 гигабайт памяти.

Я уже упоминал, что мы не будем использовать экран, а вместо него используем последовательный порт. QEMU позволяет соединить последовательный порт виртуальной со стандартными потоками ввода/вывода самого QEMU. Другими словами мы можем направить последовательный порт прямо к нам в терминал. Для этого можно использовать опцию serial следующим образом:

```
qemu-system-x86_64 -kernel kernel -serial stdio
```

Ну и наконец, раз нам не нужен экран, то мы можем попросить QEMU не создавать графическое окно "виртуального экрана", это делается с помощью опции -nographic. Впрочем, после этой опции пользоваться терминалом становится проблематично, так что я бы не советовал ее вам использовать.

8.2 Отладка с помощью GDB

Самый главный плюс использования QEMU это возможность подключить отладчик. Но в случае с QEMU, x86_64 и GDB есть некоторая тонкость. Для начала, чтобы можно было подключить отладчик нужно указать QEMU опцию -s:

```
qemu-system-x86_64 -kernel kernel -s
```

После этого к QEMU можно подключиться с помощью GDB. Для этого запустите GDB и введите в нем следующие команды:

```
set architecture i386:x86-64
target remote localhost:1234
```

После этого QEMU остановится и вы можете делать все что вы можете делать с обычными userspace программами используя дебагер. Однако, если вы хотите остановить выполнение QEMU в самом начале загрузки ядра, перед стартом функции main[1], а не в случайном месте до которого дошло ваше ядро пока вы набирали команды в GDB, то тут есть небольшая проблема.

Обычно в QEMU для этого используется опция -S, но в данном случае она нам не поможет. Причина в том, что ядро запускается изначально в

[1] а обычно вы этого и хотите

32 битном режиме и опция -S остановит ядро в самом начале 32 битном режиме, после чего ядро переводит процессор в 64 битный режим и в этом переходе из одного режима в другой заключается проблема: QEMU или GDB не могут адекватно обработать этот переход[1].

В связи с тем, что вам вряд ли придется отлаживать 32 битную часть кода или сам переход из 32 битного кода в 64 битный, то вам может помочь следующий хак. Я упоминал про файл kernel_config.h и про одну единственную опцию, которая там в данный момент есть: CONFIG_QEMU_GDB_HANG. Если вы поищете использование этой опции в bootstrap.S то вы увидите, что она включает/выключает бесконечный цикл перед самым вызовом функции main в bootstrap.S. Таким образом, если вы раскоменнтируете define в kernel_config.h ядро зависнет после перехода в 64 битный режим перед вызовом функции main.

Пока ядро крутится в бесконечном цикле вы можете неторопясь подключиться к нему с помощью GDB, настроить все нужные вам breakpoint-ы или загрузить дебажную информацию, а после чего вам нужно выйти из бесконечного цикла, например, используя команду jump GDB. Таким образом вы без проблем подключитесь к QEMU после перехода в 64 битный режим, но до вызова main.

9 Вопросы и ответы

Студент Л спрашивает: *Вопрос общего вида – что именно происходит в предложенном bootstrap.S и kernel.ld? Выставляется GPT и long-mode, это видно, но что именно там записано, и что именно происходит с сегментами кода? Какая-то часть уходит – как мне кажется – по маленькому адресу памяти (из контекста он физический), какая-то по большому (виртуальному), как всё оказывается именно там, где ожидается?*

GPT это я так понимаю GDT (Global Descriptor Table), а не GUID Partition Table? Изначально все загружается в 32 битном защищенном режиме, это нам гарантирует multiboot. Для того чтобы перейти в long mode нужно настроить paging и включить этот самый long mode, ну и создать GDT для long mode (хоть он и довольно ущербную).

Paging настраивается следующим образом. Первые 2 GB физической памяти отображаются на 2 GB виртуальной памяти в трех местах:

- начиная с 0 - для включения paging-а нам нужен identity mapping;

[1]QEMU винят GDB-шников, GDB-шникам много раз присылали разные патчи на исправление этого дела, но ниодин им не угодил, а мне без разницы кто виноват - я хочу чтобы работало, поэтому приходится придумывать обходные пути.

- на участок сразу за дырой в каноническом адресном пространстве - этому можно найти применение, понадобится ли это вам или нет, зависит от вас, так что подробнее эту часть описывать не буду;
- на 2 GB в самом верху виртуальной памяти - это собственно рабочее место нашего ядра (kernel.ld написан таким образом, что ядро должно жить там), эта часть описана в ABI.

Соответсвенно, после включения этого отображения у нас в памяти находится три копии первых двух гигабайт физической памяти. Когда ваш код получает управление все уже работает в верхних 2GB памяти, identity mapping при этом уже не предполгается использовать, а использование средней части up to you.

Студент Л продолжает: *И в целом, если мы дальше где-то будем оперировать регистрами сегментов (тот же bootstrap оперирует сегментами уже казалось бы в Long Mode), то зачем, у нас указатель уже накрывает размер памяти с запасом?. Что в них хранится?*

Ничего не понял.

Студента Л не остановить: *Соотносятся ли это с GPT, или оно просто формирует логический адрес?*

В long mode сегментные регистры за исключением FS и GS никаким образом не участвуют в трансляции адресов (и FS и GS участвуют в ней очень особенным образом). Дескрипторы же вообще никак не участвую в трансляции адресов. Но сегментные регистры и дескрипторы влияют на проверку привелегий.

Студент Л таки раскрывает тайну магического сокращения GPT: *В прошлом семестре мы разбирали, как работает GPT, но только в теории, и там мы брали готовые виртуальные адреса, смотрели по старшим битам в таблице страниц и так далее – всё так же?*

Вот он тот неловкий момент, когда понимаешь, что GPT - это на самом деле PT (т. е. page table). Теперь когда я понял, что такое GPT у меня есть два сообщения. Во-первых, не используйте не общепринятые обозначения - я очень плохо угадываю, что имеется в этом случае ввиду (а лучше вообще избегать сокращений или описывать их заранее, как в научных статьях по математике).

Во-вторых, принцип работы таблиц страниц одинаков на всех архитектурах, меняются только детали: используемые флаги,

количество уровней, распределение бит адреса по уровням. В данном конкретном случае в 64-битной версии x86 используется 4 уровня, вместо 2 в 32-битной версии (на сколько я понимаю, именно на ее примере вам и объясняли таблицы страниц).

Студент Л о страницах разного размера: *И ещё в этой GPT есть, как я понял, переменная длина вхождений, то есть страницы не одного размера – это всё ещё дружит между собой?*

Страницы разного размера можно было создавать и в 32-битной версии x86 - тут нет ничего нового, опять же, разница только в деталях. Например, в 64-битной версии можно использовать страницы трех размеров: 4KB, 2MB и, иногда, 1GB - не трудно увидеть, что эти размеры соответсвуют размерам памяти покрываемой одной записью таблицы страниц разных уровней.

Принцип действия очень простой - в записи таблицы страниц присутсвует специльный бит размера. Если этот бит сброшен, то запись указывает на таблицу страниц следующего уровня (если конечно текущий уровень вообще поддерживет этот бит). Если бит установлен то текущая запись содержит физический адрес начала "большой" страницы, а размер страницы определяется уровнем, на котором эта запись находится.

Задача bootstrap кода создать хоть какую-нибудь таблицу, которая покроет весь код и данные ядра, чтобы можно было дальше работать. Пользоваться большими страницами проще и отображение занимает меньше памяти, поэтому в bootstrap коде используются страницы по 2MB.

Почему не 1GB страницы, ведь с ними все станет еще проще? От процессора не требуется обязательная поддержка страниц в 1GB[1], поэтому я и не использую 1GB страницы.

Студент Т спрашивает: *Скажите, при инициализации serial порта нам не нужно задавать коэффициент деления (если я правильно понимаю, мы его нигде не используем)?*

Хитрый вопрос, короткий ответ на него - вам нужно задавать коээфициент деления. На что у вас должен возникнуть следующий вопрос - а какой коэффициент деления нужно использовать? И вот тут мне простым вариантом ответа не обойтись, поэтому далее длинный вариант.

[1] хотя она, скорее всего, присутсвует

Пока вы используете QEMU коэффициент деления не важен, потому что другой "виртуальный" конец соединения, с которым ваша ОС "общается" через последовательный порт, будет работать при любом коэффициенте деления, просто потому что QEMU без разницы этот коэффициент. Но если речь идет о реальном оборудовании на другом конце соединения, то оно, скорее всего, не будет работать с любым коэффициентом деления как QEMU, а будет ожидать какой-то конкретной скорости передачи по последовательному интерфейсу и вам придется задать какое-то конкретное значение.

Короче говоря, в домашнем задании задайте какой-нибудь валидный коэффициент деления и этого будет достаточно.

Студент Т задает другой вопрос: *В Interrupt Enable Register все, кроме reserved, биты должны быть единичками?*

На сколько я помню, там там наоборот должны все быть 0, потому что по заданию вам не нужно использовать прерывания для общения с последовательным портом, а вместо этого использовать так называемый polling mode. Т. е. вы должны "опрашивать" контроллер последовательного порта перед тем как что-то в него записать (или прочитать, хотя чтение из последовательного порта не требуется по заданию).

Вопрос задает студент Л. из Санкт-Петербурга: *Ещё раз, что именно записано в предложенной Global Descriptor Table, и как в зависимости от этого считать сдвиг для записи прерывания? Потому что кажется логичным использовать селектор сегмента KERNEL_CODE, который имеет значение 0x18, по документации означающий запись номер 3 (считая с нуля) в GDT, а в нашем Bootstrap записано в этом месте 0x00a09a0000000000, и эта запись по той же документации означает сегмент выполняемого кода с базовым адресом 0 и лимитом 0, измеряемом в 4K-страницах (то есть сам сегмент имеет длину примерно 4 килобайта), что как-то совсем не похоже на верхние 2GB кода. Отсюда собственно вопрос – если всё так и планировалось, то зачем, почему и как это использовать? А если нет, то как должно быть? Всё это ради того, чтобы понять, какую пару сдвиг-сегмент класть в описание прерывания в будущем.*

В long mode (64-битный режим работы x86, который мы и используем) поля Limit и Base дескриптора сегмента игнорируются, т. е. то что там записаны нули ни на что не влияет. Более того, с 20-битным полем Limit и максимальной гранулярностью в 4KB просто не

получится использовать сегменты больше 4GB - это стало бы сильным ограничением.

Касательно того, что нужно использовать в качестве селектора кода - используйте KERNEL_CODE. Вы можете проверить, что именно это значение хранится в CS

Студент Л. продолжает: *А как тогда работают обращения в память? Пусть есть честный 64-битный указатель, и я хочу обратиться туда. Я могу указать любой селектор, если они не проверяются?*

Limit - описывает размер, а Base - нужен для трансляции адресов, как они по вашему должны влиять на проверку привелегий?

Проверка привелегий происходит, когда вы записываете что-то в сегментный регистр, и для проверки используются поля DPL дескриптора, поле RPL селектора, который мы записываем и который указывает на дескриптор, и поле CPL в CS. За подробностями прошу обратиться в разделы 5.5, 5.6 и 5.7 Intel®64 and IA-32 Architectures Software Developer's Manual. Volume 3 (3A, 3B & 3C): System Programming Guide.

И снова студент Л.: *Или они на 0 уровне привилегий только не проверяются? Или что-то где-то захардкожено?*

На уровне 0 действительно почти нет смысла проверять привилегии. Но есть особые правила для регистра SS, но важны, только если вы используете более 2 различных уровней привелегий - чего, скорее всего, не будет.

References

[WIK,] Advanced programmable interrupt controller. `https://en.wikipedia.org/wiki/Advanced_Programmable_Interrupt_Controller`. Accessed: 2016-02-08.

[INT, a] Intel 8253. `https://en.wikipedia.org/wiki/Intel_8253`. Accessed: 2016-02-08.

[INT, b] Intel 8259. `https://en.wikipedia.org/wiki/Intel_8259`. Accessed: 2016-02-08.

[INT, c] Intel® 64 and IA-32 Architectures Software Developer's Manual, Volume3: System Programming Guide.

http://www.intel.com/content/www/us/en/processors/architectures-software-developer-manuals.html.

[SER,] Interfacing the serial / rs-232 port. http://retired.beyondlogic.org/serial/serial.htm. Accessed: 2016-02-08.

[CPP,] printf - c++ reference. http://www.cplusplus.com/reference/cstdio/printf/. Accessed: 2016-02-08.

[INT, 1988] (1988). 8259a. programmable interrupt controller (8259a/8259a-2). https://pdos.csail.mit.edu/6.828/2010/readings/hardware/8259A.pdf.

[NS:, 1990] (1990). Pc16450c/ns16450, pc8250a/ins8250a universal asynchronous receiver/transmitter. http://pdf.datasheetcatalog.com/datasheets/700/255656_DS.pdf.

[INT, 1993] (1993). 8254. programmable interval timer. http://www.scs.stanford.edu/10wi-cs140/pintos/specs/8254.pdf.