Семинар 1

Защищённый режим (2 лаба)

Режимы

Компьютере на базе процессора intel работают в трёх режимах

1. Реальный:

   Это 16-разрядный режим с 20 разрядной шиной адреса.

   В этом режиме работали процессору 8086.

   2^20 – позволяет адресовать не более 1ого Мб (1024 Кб).

   Работали эти компьютеры под DOS (дисковая операционная система)

   DOS – однозадачная ОС (в один момент времени, в памяти могла находится только одна программа, и выполнялась она с начала до конца)
2. Защищённый (Protected)

   32-разрядный режим

   процессоры 3086

   32ух разрядные регистры, 32ух разрядная шина адреса

   Эти процессоры поддерживали две независимые… схемы… памяти…

   a. …Виртуально По запросу (виртуально – означает что фактически такой памяти нет, она поддерживается аппаратно – существуют аппартаные схемы которые поддерживают это память)
   b. …По запросу
   c. Сегментирован по запросу (взять лучше от 2ух предыдущих)

   В защищённом режиме есть специальный режим V86(virtual) – (в нём...)

   Как задачи в этом режиме запускаются операционные системы реального режима.

   Фактически запускается виртуальная машина и в этой среде может выполнятся одна программа реального режима. Intel8086

   Этот режим многозадачный, но каждая такая запущенная виртуальная машина является виртуальной машинной 86ой, со всем вытекающими отсюда последствиями (1 программа, 1Мб памяти, операторы 16-

разрядные)

Почему режим называется защищённый (потому что microsfot наглые)

Винды являются персональным разделением данных. Вся теория операционных систем уже существовала к 4ому поколению. **Адресные пространства процесса должны быть защищены!**

Естественно адресные процессы ОС тоже защищены (как только появились многопроцессорные (многопроцессорность))

3. Длинный (режим long)

   64-разрядные регистры и 64-разрядные операнды

   Многопроцессорность

   Виртуальная память только страничная

   В этом режиме поддерживается специальный режим (…), в котором могут выполняться 32ух разрядные процессы.

Принято делить регистры на группы (см. тетрадь):

1) РОНы (Регистры общего назначения)
2) Индексные регистры и регистры-указатели
3) Сегментные
4) Регистры системных адресов
5) Управляющие
6) PAE?
7) EIP
8) EFLAGS

(Минимально адресованной ед. памяти является 1 байт.)

Перейдём к GDTR и …

Во 2ой лабе мы пишем управляющую программу.

1ое что делает эта программа – переводит из защ. режима в реал.

Необходимо написать 2 обработчика прерываний.

После этого, программа должна вернуть в защ. Режим

Можно разбить на 2 шага:

1) Написать программу, которая переводит из защ. в реал. и обратно (Рудаков-Финогеннов – костыль, будем расширять эту программу, надо найти неправильное и исправить)
2) Что-то?

Процессор может выполнять только те программы, которые находятся в памяти.

Операционная система лежит с определённого адреса в таблице (когда комп. выключен).

Минимум действий, для загрузки ОС в память, но этого мало. Необходимо обеспечить адресацию. Все команды выполняются последовательно.

Таблица глобальных дескрипторов содержит дескрипторы сегментов физической памяти.

В SMP архитектуре равноправные процессоры, которые работают с общей памятью. Один процессор назначается главным только для того, чтобы он выполнял обработчик прерываний от системного таймера (это может делать только один).

В системе JDT память будет одна, и таблица глобальных дескрипторов тоже будет одна.

Эта таблица должна находится в физической памяти, причём в памяти ядра системы (см. тетрадь)

В защищённом режиме доступно 4 Гб

Локальные дескр. таблицы описывают виртуальные адресные пространства процессов – каждая такая таблица явл. системной. Будет занимать сегмент. Сегменты физ. памяти описывается…???

LDTR – является селектором к …???

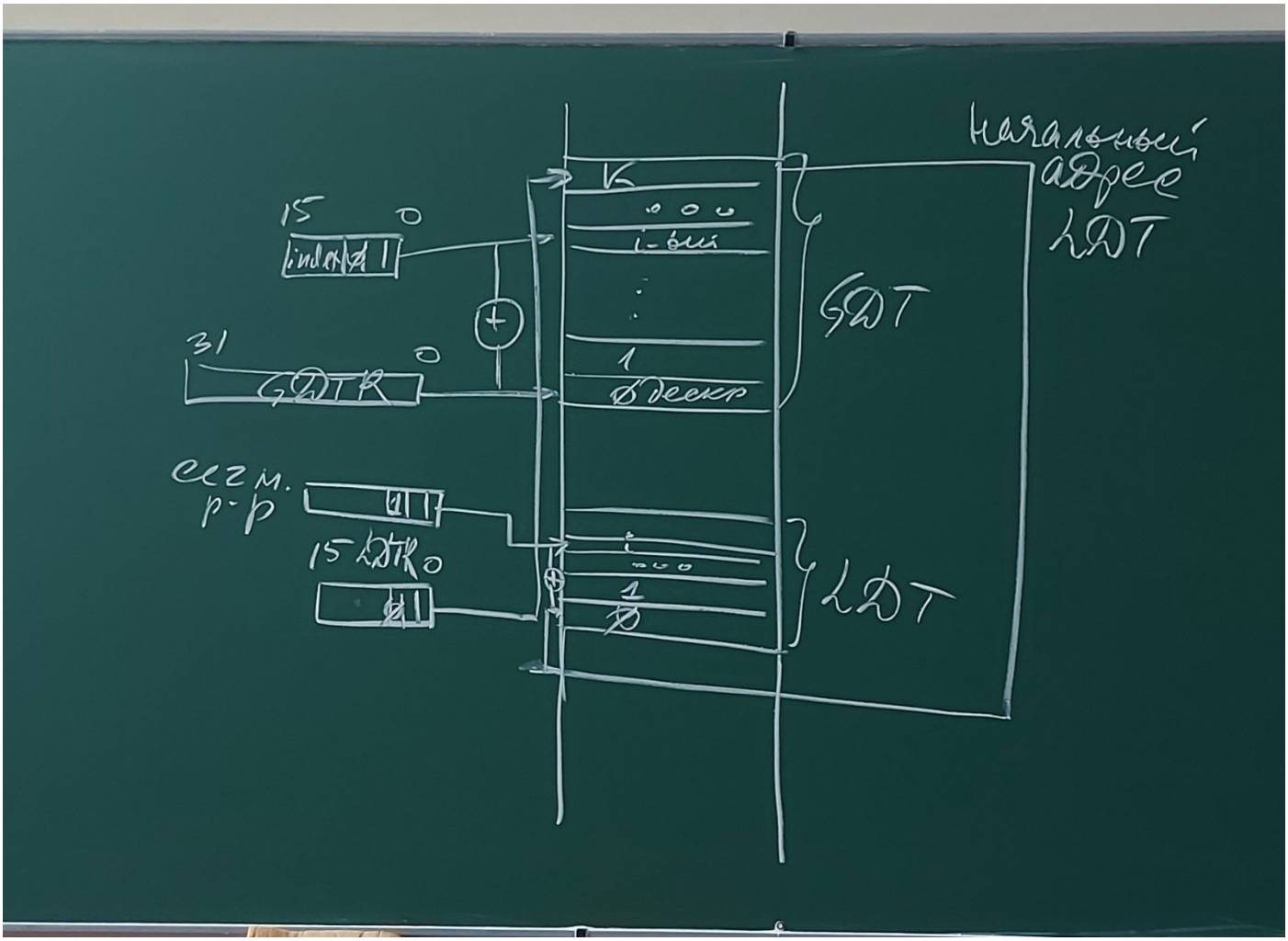
В защищённом режиме используется модель памяти FLAT.

В реальном режиме – таблица векторов прерываний, а в защищённом режиме – таблица дескрипторов прерываний (вектор — это непосредственно сам адрес прерывания, а дескриптор — это другая структура, позволяющая получить адрес)

DOS не защищён, но есть уровень привилегий.

Сохранение аппаратного контекста (pusha – сохр. все регистры)

Косвенный вызов – мощны механизм адресации (например, делали меню)



Семинар 2

ОС – это программа, GDT используется для адресации программы.

В GDT описывается … (?для физического адреса?)

!*!GDT содержит дескрипторы сегментов физической памяти.!*!

То-есть глобальная таблица дескрипторов, позволяет ?адресовать память?

Команды и данные которые находятся в оперативной памяти, находятся (доступ к ним) по адресу. Но такой адрес надо получить

Две схемы управления памятью:

1. Сегментами по запросу
2. Страницами по запросу

Предполагает выполнение преобразования памяти.

Сегментированный адрес…

Сегментный регистр смещается на 4 разряда и добавляется еще 4 байта.

В З-Р у нас есть начальный адрес начала сегмента (в дескрипторе из глобальной таблицы дескрипторов), а смещение мы берём из команды.

Обычно говорят: любая программа считает, что она начинается с нулевого адреса. Соответственно, отсюда получается, что мы получаем смещение из команды. Но мы должны понимать что это счётчик команд, или индексные регистры/указатели, или непосредственно …

Это смещение добавляется к … и получается линейный физический адрес по которому совершается обращение к памяти.

Процессоры Intel, также поддерживают локальные таблицы дескрипторов (LDTR – 16 разрядов => этот регистр, не может содержать полный адрес. В LDTR может лежать селектор) Локальная таблица описывает виртуальное пространство, локального процесса. Будет столько локальных таблиц, сколько процессов.

Во 2ой лабе:

- Работу с таймером и клавиатурой
- Определение

Наши JDT и IDTR – системные таблицы

Убрать сброс процессора.

В этой программе

Описаны 16-разрядные сегменты, а защищённый режим 32-разрядный, соответственно

Задание: расписать дескрипторы сегментов на двоичные коды и какие это сегменты.

Мы анализируем код из Рудакова-Финагенова.

В этой программе для всех 4ёх регистров установлена граница (lim) = FFFFh = 2^16=64 Кб. Потому что регистре в реальном режиме 16-разрядные.

2^20 = FFFFFh – 1 Мб –

Р-Ф ?Забыли про еще 4-е разряда в атрибутах?

Теневые регистры

С каждым сегментным регистром в процессоре сопоставлен теневой регистр. В котором при обращении к сегментному регистру, записывается информацию из …. Сделано это для того, чтобы исключить обращение к таблице JDT (которая находится в оперативной памяти). Дело в том, что О.П. отстаёт по производительности на порядок, обращение к ОП – трудозатратное действие (сущ. Понятие цикл обращение к памяти – требует определённое кол-ва тактов.

(Как часто обращается к ОП – на каждой команде а то и больше. + преобразование адреса)

Чтобы этого избежать, информация с дескриптора, записывается в теневой регистр. Теневые регистры находятся, непосредственно в процессоре (нету обращение к ОП – исключает обращение к таблице JDT)

Программа в Р-Ф кривая! Адресация — это важно!

Поскольку в реальном режиме, смещение не может превышать FFFF устанавливается граница, но в этом нет смысла, т.к. описанные сегменты и так не выходят за эту границу. (закомментировать эти 4 строки)

В нашей программе - Определить объём доступного адресного пространства.

Что для этого нужно сделать? (в З-Р можем адресовать 4 Гб) Для того чтобы адресовать нужно объявить дескриптор. 1ый Мб пропускается, со 2ого Мб-та … записываем туда сигнатуру, сравниваем со своей сигнатурой. Если сигнатуры совпали, то это память. И естественно инкрементировать счётчик. Нужно объявить дескриптор сегмента памяти 4Гб (бит гранулярности = 1, для чтения-записи)

В этих дескрипторах должны быть

- 16-разрядный сегмент кода
- 16-разрядный сегмент данных
- 32-разрядный сегмент кода
- 32-разрядный сегмент данных (для определение доступного объёма памяти)
- 32-разрядный сегмент стека (когда возвращаемся в реальный режим, нужно вернуться к стеку для реального режима)

Мы должны написать 2 обработчика:

- Для таймера
- Для клавиатуры

Прерывания в защищённом режиме

Есть IDTR (32-разрядный) – в этом регистре находится начальный адрес таблицы дескрипторов прерываний.

Все дескрипторы по 8 байт.

Первые 32 дескриптора IDT отведены под исключения.

Определены следующие исключения (некоторые):

- Нулевое - 0 – Деление на ноль (div error)

- Восьмое - 8 – Double Fault (Происходит если выполняется исключение и/или маскируемое прерывание и происходит ошибка – паника, компьютер выключается)
- 13-е – General Protection (Общая защита – должно обрабатываться специальным образом. В нашей программе на все исключения мы пишем заглушки. На 13-е пишем специальную заглушку)
- 11-е – Segment not present (Возникает когда сегмент отсутствует. Нашей программы это не касается, это касается управления памятью)
- 14-е – Page Fault (Страничное прерывание. Возникает, когда процессор обращается к странице или данным, которые отсутствуют. Обрабатывая это исключение, нужно загрузить …)

Всего определенно 19 исключений.

Исключения с 20-ого по 31-е зарезервированы.

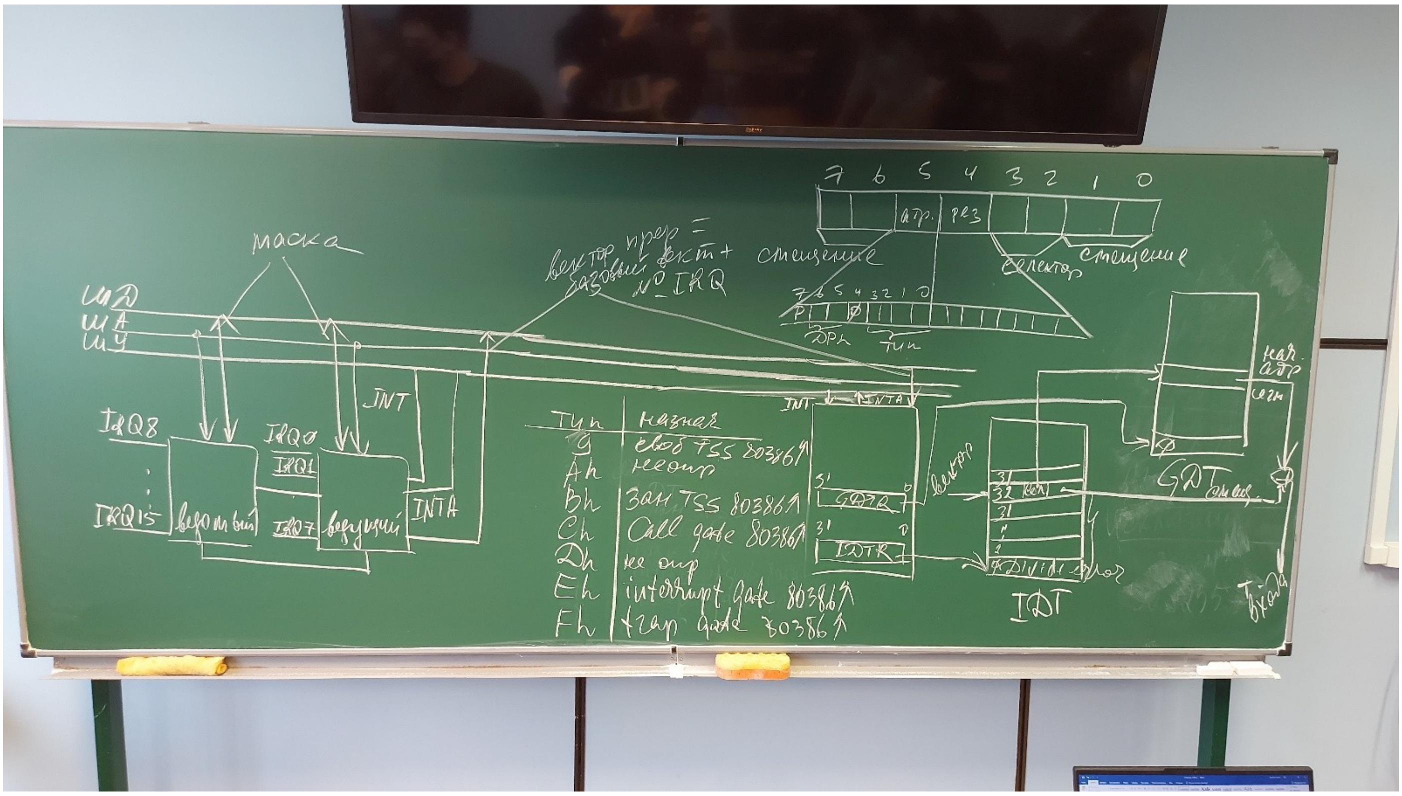
И с 32 по 255 определяются пользователем (User defined)

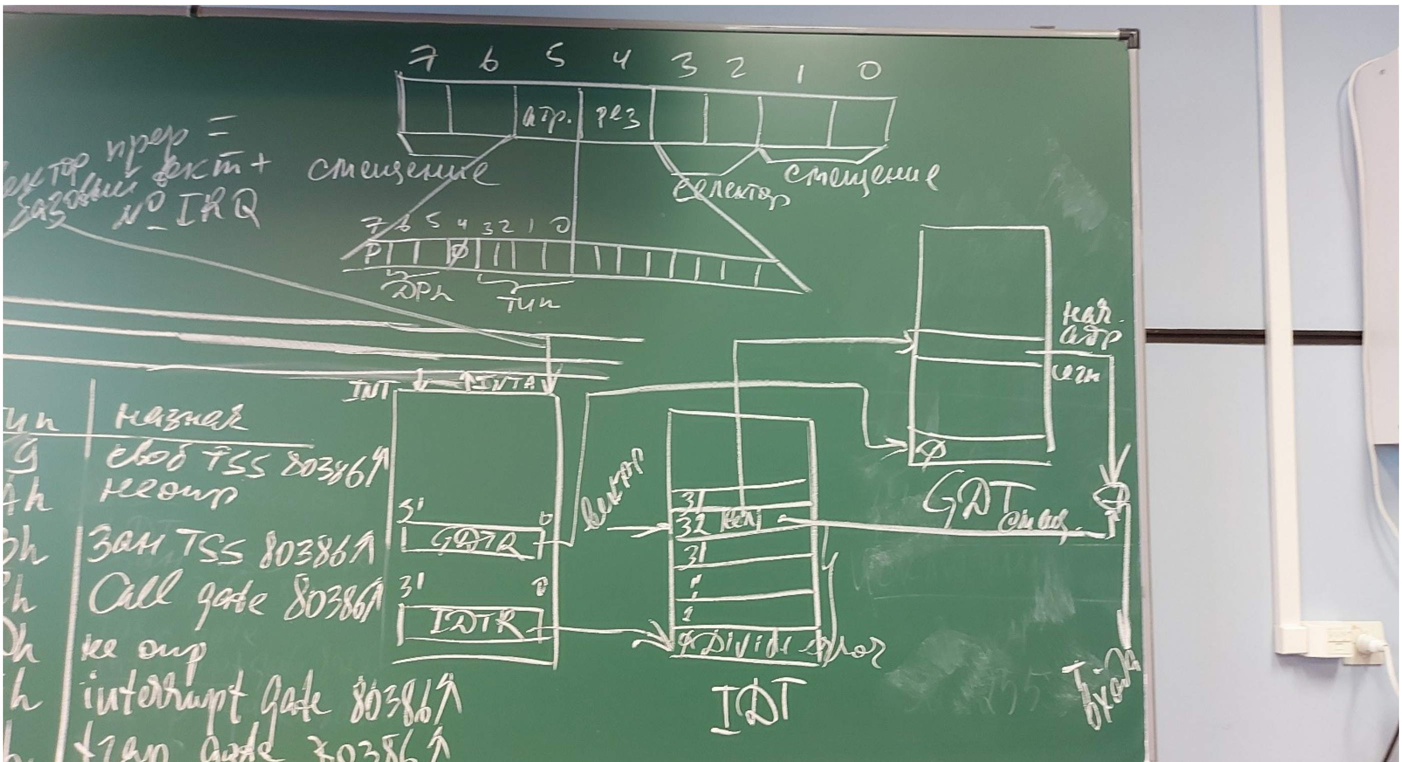
Также начинается с нулевого дескриптора (Он не пустой - деление на ноль).

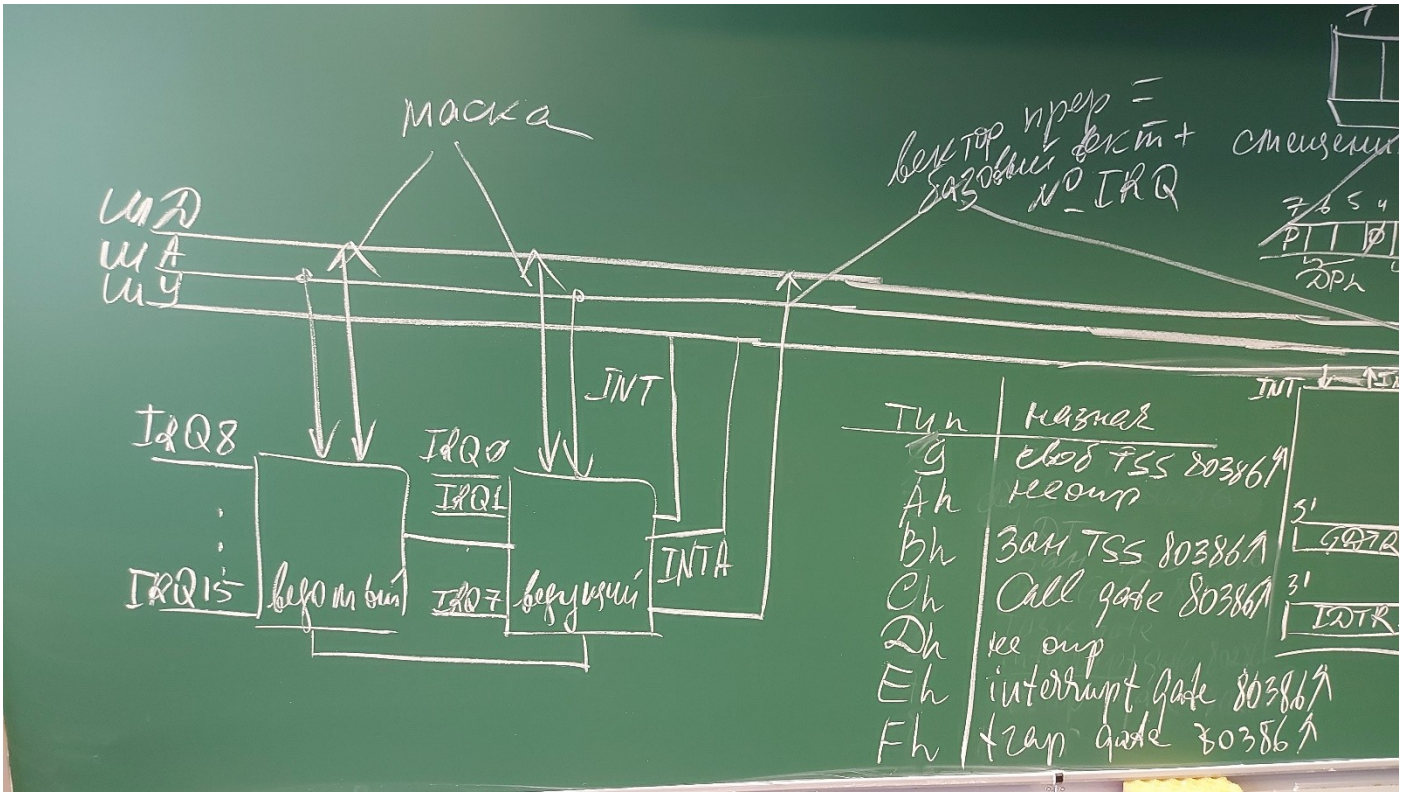
Номер исключение умножается на 8.

Нам нужно написать обработчики для прерываний таймера и клавиатуры (это аппаратные прерывания). Самое простое – через программируемый контроллер

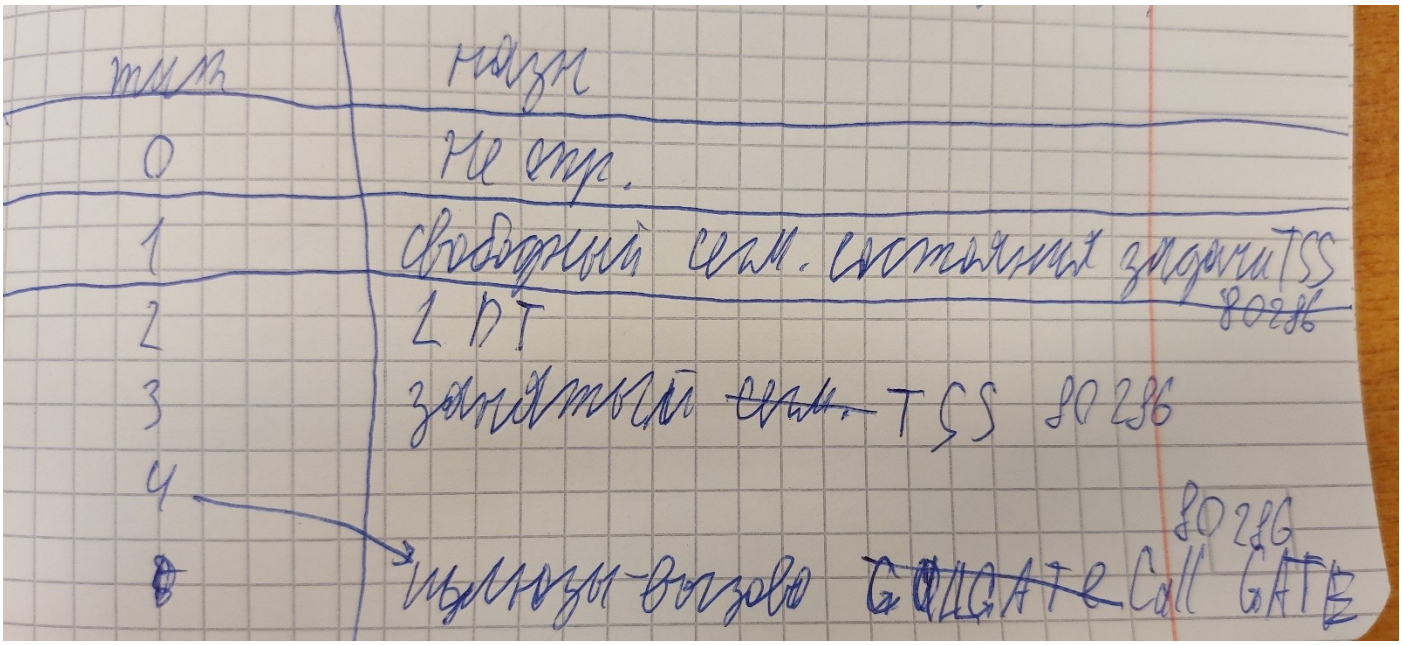
прерываний.

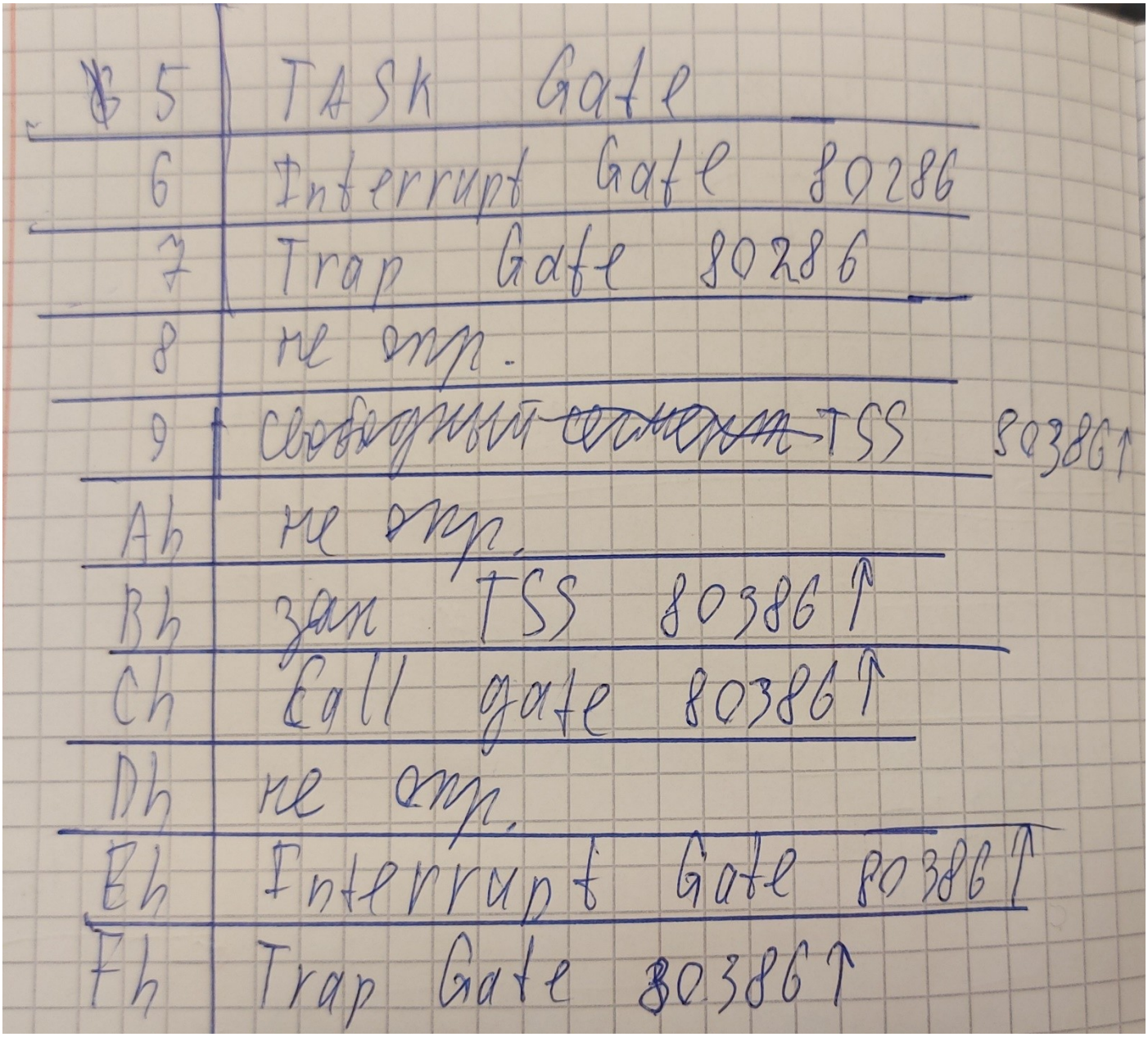

маска
IRQ8
IRQ15
IRQ7
ведомый
ведущий
INT
INTA
GDTR
IDTR
IDT
GDT


смещение
селектор
GDTR
IDTR
IDT
GDT



Нам нужно перепрограммировать контроллер прерываний.





Таким образом обработка аппаратных и программных прерываний.

Любимый вопрос: почему мы пропускаем 1ый Мб (когда определяем …)?

Вопросы по защищенному режиму:

1) Какую программу вы написали?

   Написали управляющую программу с 0 уровнем привилегий.
2) Что пришлось создать в этой программе?

   Две системные таблицы.
3) Где явно установлен соответствующий уровень привилегий?

   2 бита (DPL) в первом атрибуте дескриптора сегмента.
4) Какие сегменты описали в глобальной таблице и почему, для чего?
   - 16-разрядный сегмент кода для реального режима;
   - 16-разрядный сегмент данных, для хранения данных (строки для вывода, например);
   - 32-разрядный сегмент кода для защищенного режима;
   - 32-разрядный сегмент данных, чтобы посчитать доступную память;
   - 32-разрядный сегмент стека.
5) Охарактеризуйте дескриптор сегмента дополнительной памяти, который вы описали.

   ?.
6) Почему ваша таблица дескрипторов прерываний имеет такую структуру?

   ?.
7) Что вы написали для исключений?

   IDT, обработки для исключений (заглушки) и обработчики для прерываний клавиатуры и системного таймера.
8) Как адресуются аппаратные прерывания в защищенном режиме? (самый "ударный" вопрос)

   см. рисунок из семинара 2.
9) Когда вызывается ваш обработчик от клавиатуры / системного таймера?

   Для клавиатуры: вызывается при нажатии или отжатии кнопки на клавиатуре.

   Для таймера: 18.2 раз в секунду.

10) Какие действия необходимо выполнить для корректного возвращения компьютера в реальный режим?

По коду:

- Заперт MI;
- *far jmp;
- Обнулить флаг pe в CR0;
- Обновить сегментные регистры;
- Перенастроить базовый вектор контроллера прерываний;
- Восстановить маски контроллеров прерываний;
- Восстановить состояние IDTR реального режима;
- Закрыть линию A20 (Не обязательно?);
- Разрешить MI и NMI.

11) Что такое теневые регистры? Для чего они включены в процессор? Какую информацию содержат?

«Теневые регистры находятся в процессоре. Нужны чтоб исключить постоянные обращения к физической памяти. Тк мы обращаемся к физ. памяти на каждой команде, а то и несколько раз, особенно если адресация косвенная.» (отвечать, наверное, лучше по-другому)

Из семинара 3: Задача теневых регистров – хранить информацию о текущем сегменте в самом процессоре, чтобы не обращаться в памяти (затратное действие). (чтобы избежать обращение к таблице дескрипторов)

12) Линия A20. Что произойдет с памятью, если при переходе в защищенный режим забудем открыть линию A20?

В реальном режиме линия A20 закрыта. Когда компьютер переводится из реального режима в защищённый, она должна быть принудительно обнулена. Иначе адреса с 1 в 20ой линии будут недоступны.

13) Что произойдет, если при возвращении в реальный режим забудем закрыть линию A20?

Если мы в реальном режиме откроем линию A20, нам станет доступно еще 64 Кб памяти (HMA)

ТРЕБУЕТСЯ ПОЛНОЕ ПОНИМАНИЕ

/

Привилегии != приоритеты

Привилегии - 4 кольца защиты

Приоритеты - приоритет процессов, потоков, который назначается и может быть пересчитан.

Процессы выстраиваются в очередь в соответствии с приоритетом.

**/*

Семинар 3

Наша программа в защищённом режиме имеет нулевой уровень привилегий. В реальном режиме нету уровней привилегий.

Нет никакой защиты в реальном режиме (можем поменять вектор прерывания).

Привилегированные команды могу вызваться на определённом уровне привилегий. (То, что мы можем вызвать их в реальном режиме еще раз доказывает, что нету никакой защиты – нету уровней привилегий)

Таблица векторов прерываний – таблица реального режима. IDT – таблица защищённого режима.

Если первые 32 дескриптора (gate'a) отведены под исключения. С 32 по 255 определяются пользователем (User defined)

Нужно реализовать аппаратные прерывания (от таймера и клавиатуры).

Вектор прерывания = базовый вектор + номер IRQ. => нужно перепрограммировать базовый вектор ведущего контроллера на новый = 32. При этом мы больше никаких прерываний не обрабатываем. => на ведущий контроллер приходит только 2 сигнала (импульса): от клавиатуры и таймера. Соответственно больше никаких прерываний. На ведомый вообще никаких прерываний не приходит. Надо замаскировать все прерывания на ведомом контроллере (правила, следующие: 1-заперт, 0 - разрешение) => На ведущем FCh, на ведомом FFh. … обращается через порты (значит по адресу) значит через команды in и out. Порт 21h - ведущего, A1h - ведомого.

Написать сохранение масок. (размер маски байт):

```
    mask_master db 0
    mask_slave  db 0

; Сохранение масок (Чтобы смогли их восстановить)
    in  al, 21h
    mov mask_master, al ; Ведущий.
    in  al, 0A1h
    mov mask_slave, al  ; Ведомый.
; Перепрограммирование ведущего контроллера
    ; Т.к. (в з-р) первые 32 вектора зарезервированны для ообработки
    ; Исключений, аппаратным прерываниям нужно назначить другие векторы
    mov al, 11h
```

```
    out 20h, al
    mov al, 32 ; это новый базовый вектор (был до этого 8)
    out 21h, al
    mov al, 4
    out 21h, al
    mov al, 1
    out 21h, al

    ; Маска для ведущего контроллера
    mov al, 0FCh ; 1111 1100 - разрешаем только IRQ0 И IRQ1 (Interruption Request -
Запрос прерывания)
    out 21h, al

    ; Маска для ведомого контроллера (запрещаем прерывания)
    mov al, 0FFh ; 1111 1111 - запрещаем все!
    out 0A1h, al
```

Восстановление масок:

```
; восстанавливаем маски контроллеров прерываний
    mov al, mask_master
    out 21h, al
    mov al, mask_slave
    out 0A1h, al
```

Важно запретить все прерывание: маскируемые и немаскируемые.

Маскируемые:

```
cli; Запрет аппаратных прерываний. (Маскируемых)
```

Немескируемые:

```
; Запрет немаскируемых прерываний. NMI
    mov al, 80h
    out 70h, al
```

Разрешение:

```
    sti ; Резрешаем (аппаратные) прерывания
    xor al,al
    out 70h,al
```

Посылаются несколько команд, которые называется слово команды исполнения.

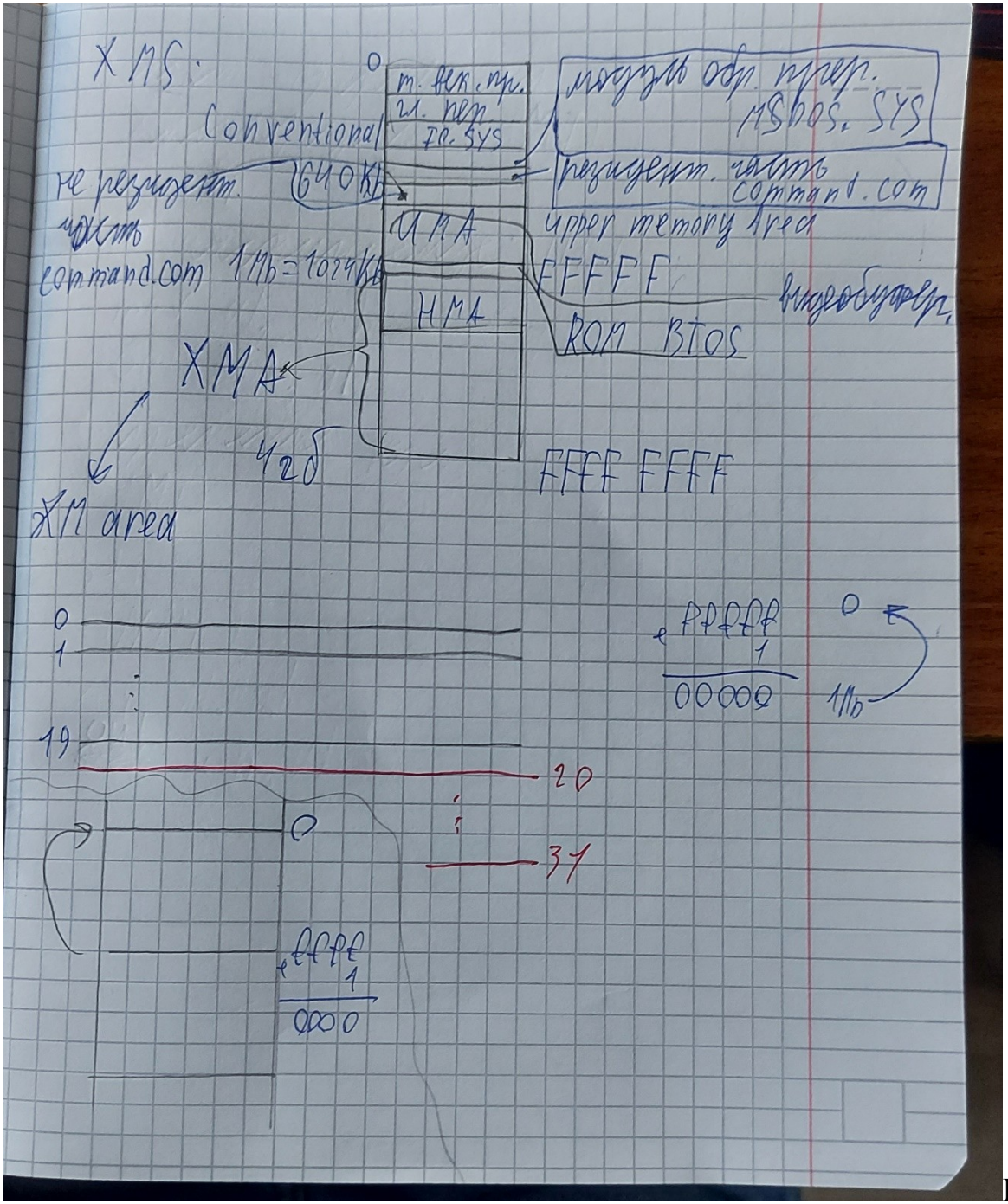
```
    ; Перепрограммирование ведущего контроллера
    mov al, 11h ; СКИ1
    out 20h, al
    mov al, base_vec ; СКИ2
    out 21h, al
    mov al, 4 ; СКИ3-IRQ2
    out 21h, al
    mov al, 1 ; СКИ4 требует EOI
    out 21h, al
```

Линия A20

Спецификации XMS и EMS (XMS - extended memory specification, EMS - expanded memory specification)

XMS – дополнительная память

EMS – расширенная память (фактически – страничное преобразование)



В реальном режиме было 20 адресных линий. Было 2 типа заворачивания (огибания).

(Аппаратно поддерживает обратную совместимость)

В реальном режиме линия A20 закрыта. Когда компьютер переводится из реального режима в защищённый, она должна быть принудительно обнулена. Иначе адреса с 1 в 20ой линии будут недоступны.

Если мы в реальном режиме откроем линию A20, нам станет доступно еще 64 Кб памяти (HMA)

Задача теневых регистров – хранить информацию о текущем сегменте в самом процессоре, чтобы не обращаться в памяти (затратное действие). (чтобы избежать обращение к таблице дескрипторов)

Существует 2 кода (линия A20):

Открыть:

```
mov al,0D1h ; команда управления
out 64h,al
mov al,0DFh ; код открытия
out 60h,al


; открытие линии A20 (Быстрый)
in  al, 92h
or  al, 2
out 92h, al
```

В большинстве новых компьютеров, начиная с … (PS/2) имеется быстрый вариант открытия линии A20. Быстрый вариант исключает опрос. Быстрый вариант, считается не вполне надёжным. Поддерживается не всем платформами. Убедится заранее поддерживается или нет – невозможно => пользоваться надо 1ым вариантом.

Закрыть линию A20:

```
; закрытие линии A20
mov al,0D1h ; команда управления
out 04h,al
mov al,0DDh ; код закрытия
out 60h,al
```

Семинар 4 (25.10.2021)

Unix

Лабораторные мы выполняем на Linux, Linux это UNIX-подобная ОС, поэтому она исполняет все

В UNIX основной абстракцией является процесс. С точки зрения UNIX – процесс часть времени выполняется в режиме задача (или в режиме пользователя) и тогда он выполняет собственный код, а часть времени он выполняется в режиме ядра и тогда он выполняет реентерабельный код операционной системы. В режиме ядра, процесс выполняет повторно входимые процедуры (или процедуры чистого кода). Повторная входимость означает что одну и ту же процедуру могут использовать несколько процессов, при этом они могут находиться … . Такое возможно только с процедурами чистого кода, то есть которые не модифицирует сами себя. Что можно изменить в коде? (Переменную), поэтому вынесены все данные. В ОС-ах существуют специальные системные таблицы (те самые таблицы к которым можно обращаться, какие-то функции ядра могут редактировать, но они не находятся в коде) – это структуры … ядра. То есть в самих чистых процедурах никаких переменных быть не должно.

Потоки могут разделять экземпляры структур (они глобальные – Системные таблицы). Ядро и функции ядра довольно плохо описаны. В UNIX любой процесс создаётся системным вызовом fork (Есть в ядре функция clone). При этом создаётся новый процесс – процесс потомок, который находится в отношениях с процессом, вызвавшим процесс, как потомок к предку. Это отношение не математическое, так просто говорят. Это отношение поддерживается соответствующими указателями в системе. А именно процесс вызвавший новый процесс … потомок получает указатель на своего предка. В Linux процесс описывается структурой, UNIX – struct proc{…} (выдадут в методичке), в Linux – struct task-struct{…} (Найти в интернете, распечатать, несколько страниц, называется дескрипторами процесса, большинство полей – указатели на другие структуры, и эта структура содержит всю необходимую информацию чтобы система могла управлять процессом и выделять ему ресурсы). Естественно, это структура ядра, они не доступны в User’e.

Как мы уже сказали, после fork создается новый процесс – процесс-потомок, который является копией процесса предка в том смысле, что потомок наследует код предка, дескрипторы открытых файлов, сигнальную маску, окружение. В старых системах код предка копировался в адресное пространство потомка, то есть – потомку создавалось собственное адресное пространство. (Что значит создать защищённое виртуальное пространство? – надо создать соответствующие таблицы – в старом UNIX – таблицы сегментов, в новой системе – таблицы страниц). Таблицы страниц – образуют адресное пространство процесса. Очевидно, что это крайне неэффективно, то есть в системе могут одновременно существовать какое-то кол-во копий одной и той же программы. Поэтому в современных системах существует два способа оптимизации задачи создания нового процесса. (можно встретить такое название - оптимизация fork)

Первый способ называется копирование при записи (copy on write). Когда вызывается системный вызов fork и создаётся новый процесс, для него создаются (карты таблицы адресов), таблицы страниц и они ссылаются на страницы адресного пространства предка, при этом для страниц данных из стека предка, права меняются на only read и устанавливается флаг copy on write. Если предок или потомок пытаются изменить какую-либо страницу, возникает исключение по правам доступа. Обрабатывая это исключение, система обнаружит установленный флаг copy on write и создаст копию нужной страницы в адресном пространстве того процесса, который пытался её изменить. Дескриптор этой страницы должен быть добавлен в таблицу страниц этого процесса – сначала дескриптор, потом всё остальное. ОС – программа, не может работать … . В результате будут созданы копии только нужных страниц. Как только было найдено copy on write решило проблему коллективного … и страничное преобразование стало ….

?Такая ситуация с изм правами доступа к? .. и установлен флаг copy on write существует пока ?...потомок? не вызовет или системный вызов exit() или системный вызов exec(). Exit() – системный вызов завершения процесса.

Второй способ. Реализован способ с системным вызовом vfork(). В этом случае для потомка не создаются собственные карты трансляции адресов, а предок

предоставляет потомку свои, при этом предок блокируется до того момента пока потомок не вызовет exit или exec.

В результате fork создаётся иерархия процессов. Эта иерархия поддерживается указателями, в UNIX есть дерево, но в UNIX есть двусвязные списки. (Таблица — это массив структур, проблема массивов в доп. накладных расходах при удалении и вставке) (Двусвязный список – для ускорения поиска. Сортировки для использования специальных методов поиска)

Процессы в списке находятся в соответствии с приоритетом.

В системе всегда есть процесс с идентификатором 0 и 1. Все идентификаторы в UNIX – это целые положительные числа. Процесс 0 – это процесс, запустивший систему. Процесс 1 – это процесс открывший терминал. (Терминал – это клавиатура и дисплей. Какая идеология ОС UNIX? – разделение времени (sharing time)).

В системе может быть создано большое кол-во терминалов и каждого будет процесс 1. Процесс 1 – предок всех процессов запущенных на данном терминале.

Рисунок дерева

Процесс у которого завершается предок – процесс сирота. При завершении процесса система проверяет не остались ли у этого процесса не завершившиеся потомки, если да, то запускается усыновление. Этого сироту усыновит процесс 1. При этом предок получит указатель на нового потомка, а предо указатель на нового предка. Fork на русском значит - вилка.

```
#include <stdio.h>
# include <unistd.h>
int main(void)
{
   int childpid;
   if ((childpid =fork())==-1)
   {
      perror("Can't fork")
      exit(1)
   }
```

```
    else if (childpid == 0)
    {
        exec(); //условная запись
        printf("Child;pid=%d,ppid=%d\n",getpid(),getppid());
        return 0; // Возвращает управление предку
    }
    else
    {
        wait(&status)
        print("parent: childpid =%d, pid=%d\n", childpid, getpid());
    }
    return 0;
}
```

Parent получает идентификатор потомка

Соглашение UNIX – если завершается с ошибкой, возвращается -1.

Для того чтобы не появлялись сироту в UNIX есть системный вызов wait(&status). Система не контролирует, где мы вызвали wait, в предке или в потомке (Любой процесс может стать предком). Также система не контролирует, где мы вызываем exec. Это дополнительные проверки, которые системы не интересны. Правильная логика, следующая: предок должен ждать завершения своих потомков. То есть в предке надо писать wait(&status) – вернёт статус завершения потомка.

Но с wait'ом связана проблема.

Рисунок 2 (вечный wait)

Это решается с помощью состояние зомби. В UNIX все процессы проходят через состояние зомби. Зомби — это процесс у которого отобраны все ресурсы кроме строки в таблице процессов (дескриптор).

Системный вызов Exec().

(Нужно выучить про fork и exec?)

Exec создаёт низкоуровневый процесс (На самом деле процесс не создаётся, процесс создаётся fork'ом). Для программы, которая передаётся exec в качестве параметра, создаётся адресное пространство, то есть таблица страниц. Exec проверяет права доступа этого процесса (потомка) к файлу, проверяется путь к файлу и является ли он исполняемым. Три типа файлов в системе: исполняемый, объектный, исходник. Exec может быть передан только исполняемый файл. После этого создаётся адресное пространство, но у child'а уже оно есть, надо убрать (уничтожить) эти страницы. Очевидно, что в дескрипторе процесса должна быть строка, содержащая адрес таблиц страниц, и надо поменять его, указав адрес новых таблиц страниц. Передать управление на точку вход (передать адрес точки входа в instruction pointer). То есть системный вызов exec() переводит процесс на новое адресное пространство.

Fork-бомба (почему называется fork бомба? :D)

Любая программа в два этапа:

1. Создаётся процесс
2. Переход в новое адресное пространство

У exec() есть несколько вариантов.

$$execl() \{ \begin{matrix} execlp() \\ execle() \end{matrix}$$

$$exec\,r() \{ \begin{matrix} exec\,r\,p() \\ exec\,r\,e() \end{matrix}$$

execl("/bin/ls", "ls", "-l", 0);
execl(char *name, char arg[0], …, char arg[n], 0);

Семинар 5 (08.11.2021)

Лабораторная 4

4 лаба – 5 файлов:

1. Sleep?
2. Убирается sleep, добавляется wait. (расширение 1ой)
3. Exec? (Но потомки должны выполнять совершенно разные коды – должна быть большая по функциональности программа)
4. (Совершенно разные сообщения!!!! (НЕ потомок 1 и потомок 2))
5. Добавление в 4ую программу своего обработчика сигналов.

+ переписанные от руки действия при системных вызовах fork и exec (материал из Вахалии, 10-11 пунктов)

Семинар 7 (29.11.2021)

```
void* shmat(int shmid, const mid *shmaddr, int shmflg);
const mid *shmaddr = 0 SHM_RND
```

На лекции рассматривали проблемы параллельных процессов.

Средства взаимодействия процессов

IPC System V

IPC – inter process communication – меж процессорное взаимодействие.

В System V используются следующие средства меж процессорного взаимодействия:

1. Сигналы;
2. Семафоры;
3. Программные каналы (именованные и не именованные)
4. Сегменты разделяемой памяти
5. Очереди сообщений. (В unix bsd это сокеты – след. семестр)

Unix – это linux подобная ОС.

Выделяются файлы отображаемые в память (mapping file) – но мы их рассматривать не будем.

Для 5ой ЛР: Симафоры UNIX и Сегменты разделяемой памяти

N1 Семафоры не имеют хозяина => mutix может освободить…

Семафор может освободить любой процесс, который знает его идентификатор.

N2 команды … и на эти команды непроизводительно тратится процессорное время. Введение семафоров … в режим ядра. … Это же привело к крайне негативной ситуации – если процессы используют большое кол-во семафоров, то очень сложно уследить за их выполнением. Поэтому всегда было стремление структурировать это… Использование набора семафоров является некоторым решением этой проблемы. Unix поддерживают наборы семафоров.

В ядре системе имеется таблица семафоров. В этой таблице отслеживаются все созданные в системе семафоры. (struct semid_ds <sys/sem.h>) Фактически это таблица дескрипторов семафоров, каждая строка описывает отдельный набор. О каждом наборе известно:

1. Имя (целое число, в Unix все идентефикаторы целые числа) – присваивается процессом который создал набор. Другие процессы, по этому имени, могут ?открыть…? и получить дескриптор для доступа к набору.
   Одной неделимой операцией можно изменить все или часть набора семафоров.
2. U_ID и G_ID (Создателя и группы)
   Процесс U_ID которого совпадает с U_ID ?процесс может изменять его и …?
3. Права доступа (Users, Group, Others, Read/Write/Execute)
4. Кол-во семафоров в наборе
5. Время изменения одного или нескольких значений семафоров последним процессом
   Это важно в отношении: случилось до, случилось после.
6. Время последнего изменения управляющих … набора, управляющим процессом. (По той же причине)
7. Указатель на набор (или массив) семафоров.
   Индексы начинаются с 0.

О каждом семафоре набора имеются след. данные:

1. Значение семафора
2. Идентификатор процесса, который оперировал семафором в последний раз.
3. Число процессов, заблокированных в текущий момент времени на семафоре.

Для семафоров определены следующие системные вызовы:

semget();

?semctl();

semop();

```
void* shmat(int shmid, const mid *shmaddr, int shmflg);

int semget(key_t key, int num_sem, int flg); //верн1т дескриптор

//sem control - управлять
int semgctl(int semfd, int num, int cmd, union semun arg); //fd - file descriptor

int semop(int semfd, struct semluf *opsptr, int lim);
```

Если системный вызов не выполнен, то он возвращает -1 => их проверяют на -1.

```
struct semluf
{
    nshort sem_num; //
    short sem_op; //Операция на семафоре
    short sem_fl;  //Флаги определённые на семафоре
}
```

Три типа операций:

1) sem_op > 0 – освобождение семафора
2) sem_op == 0 – процесс выполнивший такое, переводится в сост. Ожидания до момента освобождения ресурса. (Блокируется)
3) sem_op < 0 – захват семафора (⇔ P на S)

Флаги:

- IPC_NOWAIT – флаг информирует ядро о нежелании процесса переходить в состояние ожидания. Наличие этого флага объясняется желание избежать блокировки всех процессов, которые находятся в очереди к семафору. В случае если, захвативший семафор процесс завершился аварийно или получил сигнал kill. В силу того что сигнал kill нельзя перехватить, процесс не сможет освободить семафор. И все процессы которые находятся в очереди к данному семафору будут заблокированы на вечно (что очень негативно).
- SEM_UNDO – этот флаг указывает ядру, что необходимо отслеживать изменения значений семафора в результате вызова semop() для того чтобы

при завершении процесса ядро могло ликвидировать сделанные процессом изменения. (Для семафора это важно – у него нет хозяина)

Пример:

```
#include <sys/types.h>
#include <sys/ipc.h>
#include <sys/sem.h>

struct sem_but sbuf[2]={{0, -1, SEM_UNDO|IPC_NOWAIT},{1, 0, 1}}; //Массив структур
int main()
{
    int perms = S_IRWXV|S_IRWXG|S_IRWX_0; //Полные права доступа
    int fd segment(100, 2, IPC_CREATE|perms); //Создаёт набор из двух семафоров с
идентификатором 100, возвращает файловый дескриптор
    if (fd == -1) //Набор создать не удалось
    {
        perror("semop");
        exit(1);
    }
    //Если удачно, то выполняем semop()
    if (semop(fd, sbuf, 2) == -1) //передаём
        perror("semop");
    //Если всё благополучно то
    //То есть семафор был захвачен, не освобождён, процесс завершился
    //Но система отменит изменения т.к. установлены флаги
    return 0;
}
```

Сегменты разделяемой памяти

Разделяемая память – один процесс может туда записать значение, а другой считать значение оттуда.

В ядре имеется таблица сегментов разделяемой памяти. (Таблица разделяемых сегментов) (struct shmid_ds <sys/shm.h>)

Ни один процесс не может залезть в чужое адресное пространство, поэтому для общения нужно третье. Разделяемые сегменты подключаются к виртуальному адресному пространству, то есть получает ссылку на разделяемый сегмент.

На разделемых сегментах определены следующие системные вызовы:

Shmget();

Shmctl();

Shmat(); // touch? - подключить

Shmdt(); //detouch – отключить

После создания разделяемого сегмента, любой процесс может присоединить его к своему виртуальному адресному пространству (исп. с.в. Shmat()) и работать с ним к с собственным. По завершении процесса, разделяемый сегмент сохраняется. То есть разделяемые сегменты не удаляются даже если завершаются процессы, которые их создали. У каждой разделяемой памяти (или разделяемого сегмента) есть назначенный владелец и удалять эту область из ядра или корректировать её управляющие параметры могут процессы, которые имеют привилегированные права, создателя или назначенного владельца.

Пример:

```c
#include <sys/types.h>
#include <sys/ipc.h>
#include <sys/shm.h>
#include <string.h>

int main()
{
    int peerms = S_IRWXU|S_IRWXG|S_IRWXO;
    //Создаётся роазделяемый сегменты с идентефикатором 100, 1024 байт
    //С полными правами
    int fd = shmget(100, 1024, IPC_CREATE|perms);
    if (fd == -1)
    {
        perror("shmset");
        exit(1);
    }
    //Если создан, процесс пытается подключить разд. сегмент
    cahr *addr = (char*)shmat(fd, 0, 0);
    if (addr == (char*)-1)
    {
        perror("shmat");
        exit(1);
    }
    //Если получилось подключить, он записывает туда "Hello"
    strcpy(addr, "Hello");
    //Затем процесс отсоединяется от разделяемого сегмента
    if (shmdt(addr)==-1)
        perror("shmdt");
    return 0;
}

void *shmat(int shmid, const void *shmaddr, int shmflg);
```

В системе на (?Разделённых сегментах?)… определены следующие системные ограничения:

Системные ограничения	Знач.
SHMMNI	Максимально возможное кол-во сегментов, которые могут существовать одновременно
SHMMIN	минимально возможный размер сегмента в байтах
SHMMAX	максимально возможный размер сегмента в байтах

SHMMNI – кол-во сегментов … могут существовать одновременно

SHMMIN – минимально возможный размер сегмента в байтах

SHMMAX – максимально возможный размер сегмента в байтах

Причём, если процесс пытается создать сегмент, когда их уже максимально возможное кол-во, то процесс блокируется.

Процесс пытающийся создать сегмент больше максимально возможного размера выполнен не будет.

Семинар 8 (06.12.2021)

?System V?

Сигналы

Механизм сигналов позволяет процессам реагировать на события, которые могут произойти внутри процесса или вне его. Как правило, получение некоторым процессом сигнала, указывает ему на необходимость завершить свою работу. Вместе с тем реакция процесса на сигнал зависит от того, как сам сигнал определяет своё поведение в случае приёма какого-то определённого сигнала.

Процесс может реагировать на сигнал стандартным образом, то есть в соответствии с существующим в системе обработчиком сигнала, может игнорировать сигнал или вызвать на выполнение свой обработчик сигнала.

Начиная с классического Unix сигналы имеют числовой идентификатор, а также мнемоническую форму записи идентификатора сигнала, которые хранятся в библиотеке <signal.h>. (Мнемоника – запоминания, мнемоническая форма – форма удобная для запоминания)

В классическом UNIX было определенно 20 сигналов.

Укажем некоторые, которые встречаются в наших работах:

```
#define NSIG 20
#define SIGHUB 1 //разрыв связи с терминалом
#define SIGINT 2 //сигнал завершения программы Ctrl+C
#define SIGUIT 3 // Ctrl+слэш
//...
#define SIGKILL 9
//...
#define SIGSEGV 11 // Нарушение сегментации, выход за пределы сегмента
//...
#define SIGPIPE 13 // ?Запись в канал есть, чтения нет? - недопустимая операция с
каналом
#define SIGALARM 14 // Прерывание от таймера
#define SIG 15 // команда kill которая выполняется в ... режиме
#define SIGUSR1 16
#define SIGUSR2 17
#define SIGCHL 18 //сигнал который получает предок при завершении процесса потомка


#define SIG_DFL(int(*)()) 0
#define SIG_IGN(int(*)()) 1;
```

Средством посылки и приёма сигналов служат два системных вызова: signal и kill.

```
int kill(int pid, int sig);
//вызов
kill(pid, sig);
//сигнал sig будет послан процессу с pid и всем потомкам процесса
```

В UNIX процессы объеденяются в группы. Например, может быть такая ситуация когда в первом параметра указывается pid<= 1: в этом случае сигнал будет послан группе процессов. При pid == 0 сигнал будет послан всем процессам с группой = процессу вызвавшему kill.

```
//Примеры:
kill(37, SIGKILL); // Приказывает процессу с pid 37 безусловно завершиться
kill(getpid(), SIGALARM); //Процесс вызвавший kill, получит сигнал побудки
//getpid() - получает собственный идентификатор
```

signal() не является стандартным в POSIX 1. Но он определён во всех Си, соответственно во всех Unix подобных системах. Поскольку он не portable, что такое POSIX? POSIX 1– аббревиатура: Portable Operating System Interface.

POSIX 1 FIPS (Federal Information Processing Standard)

Portable Operating System Interface

В настоящее время существует POSIX 2 (Он включает в себя 1 и еще кое-что…)

(Антимонопольный закон… В Европе создан стандарт X/OPEN)

X/OPEN основан на … POSIX1, POSIX2.

Поскольку signal() не входит в POSIX – его не рекомендуется использовать в переносимом ПО. … поведение отличается от поведения в BSD

signal() – возвращает указатель на предыдущий обработчик данного сигнала, и его можно использовать для восстановления данного сигнала. Еще можно восстановить обработчик сигнала с помощью DFL.

```
#include <signal.h>
int main()
{
    void(*old_handler)(int) = signal(SIGINT, SIG_IGN);
    /*действия*/
    signal(SIGINT, old_handler);
    return 0;
}
```

Поскольку системный вызов signal() не входит в POSI, есть с.в. который входит в него. Это sigaction

```
int sigaction(int sig_num, struct sigaction *action, struct sigaction *old_action);
//struct sigaction определена в <signal.h>
```

Важнейшей особенностью сигналов, является возможность определить процессу собственную реакцию на сигнал. => можно изменить ход программы.
sigsetjmp() – устанавливает одну или несколько позиций в программе
siglongjmp() – осуществляет переход на одну из выделенных позиций

Pipe (Пайпы – букв. перевод труба)

Изначально были созданы неименованные программные каналы, потом именованные. В отличии от разделяемой памяти, для которой в системе существует таблица разделяемых сегментов, программные каналы поддерживаются файловой подсистемой (Это значит, что у программных каналов есть дескриптор файла). То есть каналы, являются специальными файлами.

Существует два типа каналов:

1. Неименованные – только ?I_NODE? дескриптор
2. Именованные – имеют имя и ?I_NODE?

Неименованные каналы создаются системным вызовом pipe(), который возвращает его файловый дескриптор, если удалось его создать. Но поскольку нету имени, а только дескриптор, то пользоваться им могут только процессы родственники, потому что процессы потомки в результате системного вызова fork() наследуют от процесса предка дескрипторы открытых файлов. Потоковая … (как вода в трубах), это симплексная (односторонняя) связь. Для двухсторонней (дуплексной) связи необходимо как минимум две трубы.

Программы используют встроенные средства взаимоисключения. То есть в канал нельзя писать, если из него читают. И из канала нельзя читать, если в него пишут.

(Только пользовательские процессы могут иметь динамический пересчёт)

Именованные каналы создаются командой mknod (mknod <pipe> p)

Соответствующий системный вызов можно вызывать из программы.

```
mknode(<имя>, IFIFO|ACCESS,0);
//формальные параметры - с типами
```

Программные каналы определены в системе для взаимодействия процессов. Программные каналы могут создаваться, только в адресном пространстве ядра системы. Программный канал/pipe буферизуется на 3ёх уровнях:

На 1ом уровне буферизуется в области данных ядра системы
Обычно размер канала не превышает размера одной страницы (4096 б). При переполнении системной памяти программные каналы (буфера), имеющие наибольшее время существования, переписываются во вторичную память. Эта процедура использует стандартные функции управления или работы с файлами. Если процесс пытается записать в трубу >4096 б, то труба буферизуется во времени, приостанавливая процесс, до тех пор, пока все данные не будут прочитаны.

3 уровня:

1. ?.../В области данных ядра системы?
2. На диске
3. Во времени

Ограничение чтобы повысить эффективность обмена т.к. при это не используется обращения к внешней памяти. (которые явл. медленными)

Программный канал обеспечивает потоковую передачу данных. Кроме этого, канал имеет встроенные средства взаимоисключения, чего не имеет разделяемая память. Поэтому разделяемые сегменты используют с семафорами (всё ложится на программиста). Всё что положили в трубу, так там лежать и будет.

Очереди сообщений

В ядре системы существует таблица очередей сообщений.

Рисунок… (см. тетрадь)

<sys/msg.h>

Struct msgid_ds

msg_spot – указатель на выделенную область память в которой находится сообщение.

На очередях сообщений определены следующие системные вызовы:

```
msgget()
msgctl()
msgsnd()
msgrcv()

struct msg_buf
{
    long mytype; // тип сообщения
    char mytext[MSGMAX]; //то что мы пишем в этом сообщении
}
```

Важные особенности:

Когда процесс передаёт сообщения в очередь, ядро создаёт для него новую запись и помещает её в конец связного списка записей, соответствующих сообщениям указанной очереди. В каждой такой записи указываются:

- тип сообщения
- длина сообщения (в байтах)
- указатель на область данных ядра системы, в которую копируется сообщение, и в которой он будет фактически находиться

Ядро копирует сообщение из адресного пространства процесса отправителя в область данных ?ядра? для того чтобы процесс отправитель мог завершиться. При этом сообщение остаётся доступным для чтения другими процессами.

Когда какой-то процесс выбирает сообщение из очереди, ядро копирует это сообщение в адресное пространство этого процесса, после этого, сообщение удаляется.

Процесс может выбрать сообщение из очереди следующими способами:

1. Взять самое старое сообщение, независимо от его типа
2. Взять сообщение, если идентификатор сообщения, совпадает с идентификатором, который указал процесс. Если существует несколько сообщений с этим идентификатор, то берётся самое старое из них.

3. Процесс может взять сообщение, числовое значение типа которого является наименьшим из меньших или равным значению типа, указанного процессом. Если таких сообщений несколько, то берётся самое.

Таким образом мы видим, что процессы могут не блокироваться в ожидании сообщения и при отправке, и при получении. (*Вспомнить диаграмму состояний процесса при передаче сообщений)

Пример:

```
//Пример
//#include ...

#ifndef MSGMAX
#define MSGMAX 1024
#endif

struct mbuf
{
    long mtype;
    char mtext[MSGMAX];
}mobj={15, "Hello"};

int main ()
{
    int fd = msgget(100, IPC_CREATE|IPC_EXCL|0642);
    if (fd == -1 || msgsnd(fd, &mobj, strlen(mobj.mtext))+1, IPC_NOWAIT))
        perror("message");
    return 0;
}
```

В примере создаётся новая очередь с иденетфикатором 100 и устанавливаются следующие права доступа, чтение/запись – 6 для user, только чтение для группы и только запись для остальных. Если msgget успешно, то отправляется “Hello”, с типом 15, при этом этот вызов не блокирующий (IPC_NOWAIT).

Как мы видим, в отличии от разделяемых сегментов, при передаче сообщений, выполняется копирование. При посылке – из программы в область данных ядра, и при чтении – из области данных ядра в буфер программы. Но при этом, если процессу нужно прочитать сообщение, он имеет гибкие возможности и не будет блокирован при чтении. => очереди сообщений позволяют избежать лишних блокировок.

На очередях сообщений определено довольно много флагов. В частности IPC_EXCL (прочитать самостоятельно, что этот флаг означает в примере?)