Билет 1

1. Процесс - как единица декомпозиции. Потоки и их типы. Далее нужно было нарисовать схему fork (которая максимально подробная), расписать как это работает.

Процесс - программа в стадии выполнения. Является единицей декомпозиции системы. Является потребителем системных ресурсов.

Процесс имеет две характерные черты

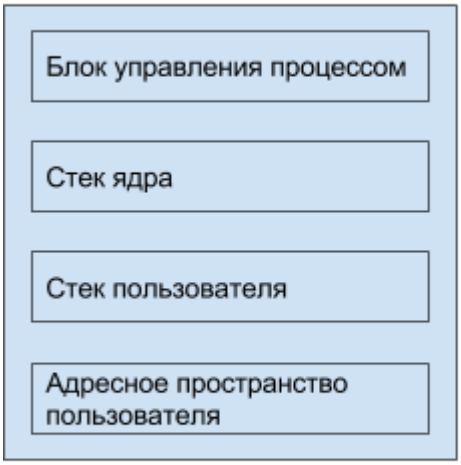
1. Владение ресурсами. Процесс имеет виртуальное защищенное адресное пространство все время его жизни, и время от времени процесс владеет такими ресурсами, как: 1) физическая память 2) устройство ввода вывода 3) (открытые) файлы 4) объекты ядра
2. Планирование и выполнение: scheduling and execution. За время своей жизни процесс изменяет свои состояния в соответствии с диаграммой состояний процесса, однако вторая характеристика процесса может быть отделена от процесса и передана его части – потоку.

Поток – это часть кода программы, выполняемая последовательно, но параллельно с другими частями кода. Это динамический объект, представленный в процессе точкой начала и последовательностью команд, отсчитываемой от точки начала. Он не имеет собственного адресного пространства, выполняется в адресном пространстве процесса, при этом процесс является владельцем ресурсов.

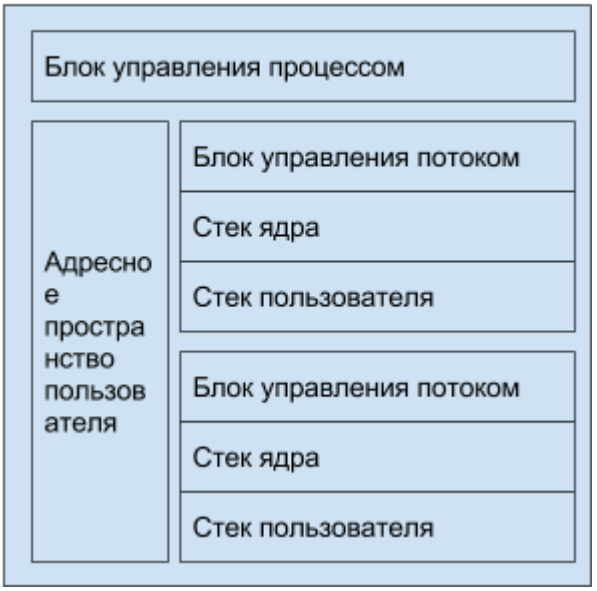
Чем владеет поток? Поток владеет счетчиком команд, т. е аппаратным контекстом. То есть, при переключении потоков в рамках одного процесса переключается только аппаратный контекст.

Модели процессов.

Однопоточная модель процесса



Многопоточная модель процесса

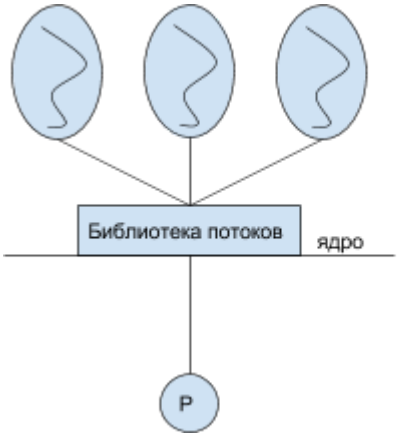


Почему в системе может быть только одна таблица процессов?

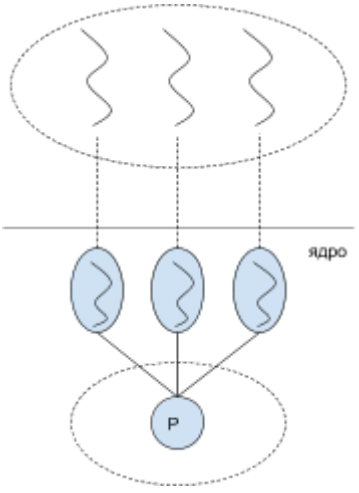
Почему надо иметь стек ядра и стек пользователя? Потому что мы с вами говорили, когда рассматривали диаграмму состояний: часть времени процесс выполняется в режиме пользователя (задачи), и тогда процесс выполняет собственный код. А часть времени процесс выполняет реэнтирабельный код ОС. Когда процесс переключается в режим ядра, с ним происходят всякие метаморфозы, при этом он запрашивает дополнительные ресурсы – например, запрашиваются устройства ввода-вывода. После этого возникает аппаратное прерывания и вызывается соответствующий обработчик прерывания.

Типы потоков.

Глобально различаются два типа:



- Потоки уровня пользователя или прикладные потоки. Об этих потоках ядро ничего не знает. Управление такими потоками осуществляется через библиотеку управления потоков. Она должна содержать программы работы с потоками, а именно: создание, удаление, обмен сообщениями, планирование выполнения потоков, переключение контекста. Все хорошо, пока не выполняется никаких системных вызовов. Если поток выполняет системный вызов, например, запрос ввода-вывода – блокируется весь процесс, с сохранением полного контекста. Тоже самое происходит с истечением кванта процесса.



- Потоки на уровне ядра. Они создаются в результате системного вызова, ядро о них полностью извещено. При переключении таких потоков сохраняется только аппаратный контекст, но в очереди к процессору стоят процессы, то есть, такого не бывает, что процесс мы запустили и потоки этого процесса аккуратно стоят в очереди один за другим. Система не учитывает, какому процессу принадлежит поток.

Примечание: системы Linux и Windows – многопоточные ОС.

Light-weight process – легковесный процесс – это и есть нити, поток.

Процессы Unix

В Unix концепция процессов является базовой. С точки зрения Unix процесс часть времени выполняет собственный код в режиме пользователя, а часть времени – реэнтирабельный код ОС в режиме ядра.

Процесс **создается** с помощью единственного системного вызова **fork**(). Ему выделяется строка в таблице процессов. Если памяти нет, то он переводится в состояние «готов к запуску, выгружен» (только система подкачки). Если у процесса

есть возможность выделить RAM, то такой процесс будет переведен в состояние «готов к запуску, в памяти».

В начальный момент времени процесс получает минимальное число страниц.

Разработчики системы договариваются, какие состояния процесса в системе они будут выделять. Это их произвол.

«Зарезервирован» – возникает, если в системе запущено очень много процессов, и процессу не хватает ресурсов.

В современных системах процессы создаются по мере необходимости, а ресурсы выделяются по мере надобности. То есть, фактически, в системе может быть создано V количество процессов, но в Unix/Linux это лимит, выраженный числом типа integer, хоть она и очень большая.

2. Прерывания и их виды (спрашивала про то, какие из них синхронные, какие асинхронны и что это значит). Потом нужно было нарисовать диаграмму ввода/вывода (ту, где есть узел vfs).

Система прерываний

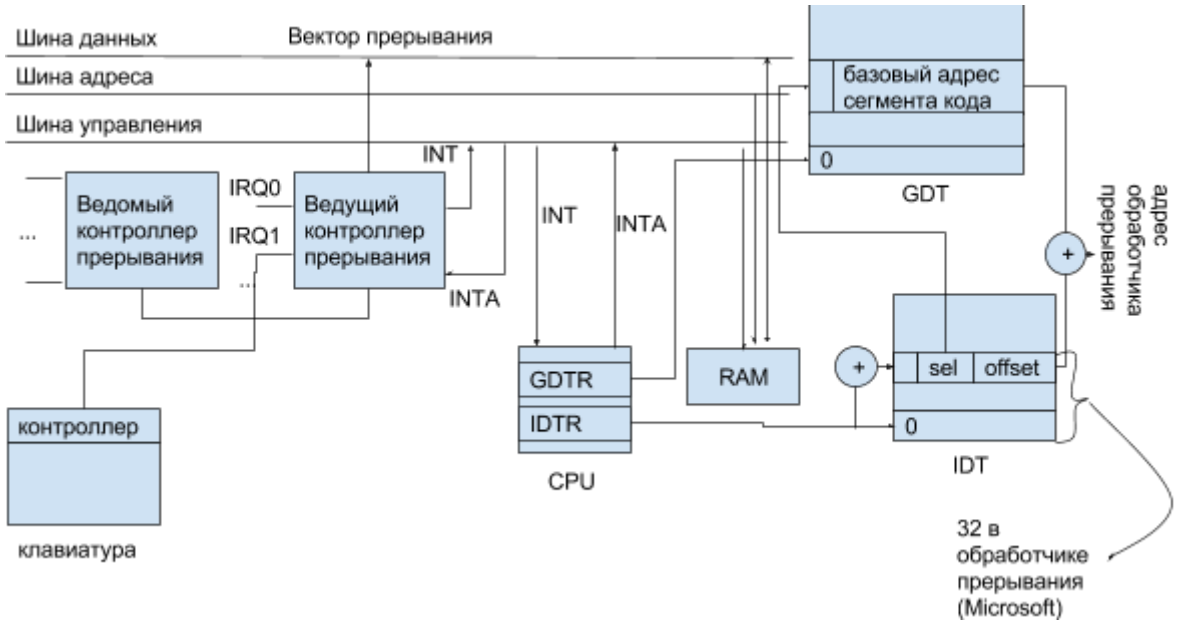
Типы прерываний:

1. системные вызовы
2. исключения (устранимые / неустранимые). Пример устранимого исключения - страничные прерывания
3. аппаратные прерывания:
    a. системный таймер. Главная функция - декремент счетчика времени
    b. прерывание от действия оператора
    c. инициируемые устройствами ввода/вывода по завершению операции ввода/вывода

1 и 2 - синхронные по отношению к выполняющемуся процессу (вызваны непосредственно выполнением или попыткой выполнения команды)

Аппаратные прерывания являются асинхронными (вызваны событиями, которые не зависят от выполняемых команд).

Механизм реализации аппаратных прерываний



Когда устройство заканчивает свою работу, оно инициирует прерывание (если они разрешены ОС). Для этого устройство посылает сигнал на выделенную этому устройству специальную линию шины. Этот сигнал распознается контроллером прерываний. При отсутствии других необработанных запросов прерывания контроллер обрабатывает его сразу. Если при обработке прерывания поступает запрос от устройства с более низким приоритетом, то новый запрос игнорируется, а устройство будет удерживать сигнал прерывания на шине, пока он не обработается.

Контроллер прерываний посылает по шине вектор прерывания, который формируется как сумма базового вектора и No линии IRQ (в реальном режиме базовый вектор = 8h, в защищенном – первые 32 строки IDT отведены под исключения => базовый вектор = 20h). С помощью вектора прерывания дает нам смещении в IDT, из которой мы получаем точку входа в обработчик. Вскоре после начала своей работы процедура обработки прерываний подтверждает получение прерывания, записывая определенное значение в порт контроллера прерываний. Это подтвержд. разреш. контроллеру издавать новые прерывания.

Быстрые и медленные прерывания.

Linux различает быстрые и медленные прерывания. Быстрые прерывания должны выполняться быстро(обычно устанавливается flag выполнения прерывания так быстро как это возможно) Быстрые прерывания выполняются при запрете прерываний на локальном процессоре, что не маловажно для быстрого завершения. Если флаг сброшен то это не является быстрым прерыванием,прерывания разрешены, за исключением маскированных на всех процессорах.В линуксе для этих отложенных действий есть три основных средства: softirq,tasklet,work_queue.

Билет 6

1. Взаимоисключение и синхронизация процессов и потоков. Семафоры: определение, виды, примеры; семафор, как средство синхронизации и передачи сообщений. Операции на семафоре в Linux. Семафоры в решении Дейкстры задачи “производство-потребление”.

2. Подсистема ввода-выводу: программные принципы управления устройствами. Специальные файлы устройств — идентификация устройств в ОС Linux, основные точки входа драйверов устройств. Адресация обработчиков в защитном режиме.

Билет 9

1. Процессы: алгоритм планирования. Системы приоритетов в ОС Windows и ОС Linux. Пересчет приоритетов в ОС Windows и ОС Linux.

Выделение процессора, т.е. процессорного времени, процессу выполняет диспетчер. Постановка процессов в очередь в соответствии с выбранной дисциплиной планирования выполняет планировщик.

Планирование процессов - это управление распределением ресурсов между конкурирующими процессами путем передачи им управления согласно некоторой стратегии планирования. По-английски планировщик - scheduler.

Алгоритмы планирования классифицируются следующим образом:

A. алгоритмы без переключения и с переключением
B. планирование бесприоритетное и с приоритетами
C. если у нас приоритетное планирование, то алгоритмы могут быть с вытеснением и без вытеснения
D. приоритеты могут быть статистические и динамические, абсолютными и относительными.

На планирование, прежде всего, оказывает влияние тип ОС, т.е. в разных ОС реализовываются разные принципы планирования.

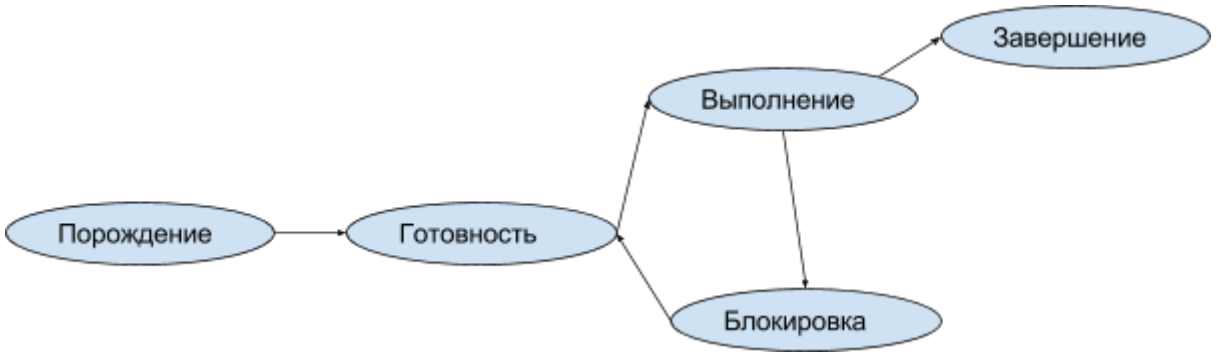
Для систем пакетной обработки рассмотрим следующие алгоритмы и классифицируем их:

1. **fifo - first in first out -** без переключения, без приоритетов. В соответствии с таким алгоритмом процесс будет выполняться от начала до конца.
2. **shortest job first -** наикратчайшее задание выполняется первым. это приоритетное планирование без переключения и вытеснения. Этот алгоритм имеет очень негативное свойство - бесконечное откладывание. Длинные

задания, требующие много процессорного времени, могут постоянно откладываться.

3. **shortest remaining time -** задание с наименьшим временем оставшегося выполнения выполняется в первую очередь. Выполняющийся процесс может быть прерван, если в очередь поступит процесс с меньшим оценочным временем выполнения, чем то время, которое процессу осталось до его завершения. Это алгоритм с вытеснением без переключения. Для реализации этого алгоритма надо следить за текущим временем обслуживания. Очевидно, что такой алгоритм требует дополнения в диаграмму состояний - выполнение может быть прервано вытеснением в состояние готовности.

// Если вдруг попросит диаграмму состояний

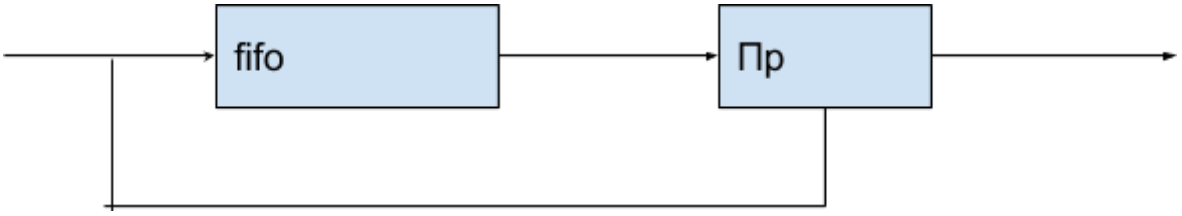


4. **HRN - highest response ratio next -** наибольшее относительное время ответа. Учитывается время ожидания готовых процессов. Чем выше время ожидания, тем выше приоритет процесса, т.е. это динамические процессы. *p=(tw-ts)/ts,* где *tw* - время ожидания, *ts* - запрошенное время.

Рассмотренные алгоритмы являются алгоритмами, используемые в системах пакетной обработки.

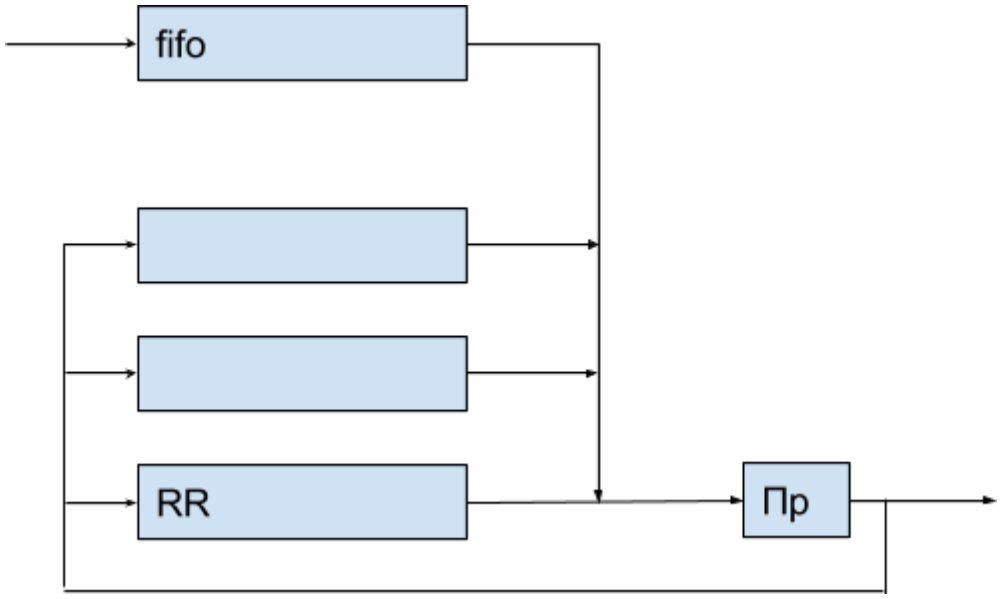
Алгоритмы систем разделения времени

1. RR. С переключением, без вытеснения, без переключения.



2. Адаптированное планирование. На каждом уровне приоритета может находиться одновременно большое количество процессов. В нее попадают вновь созданные процессы и процессы после ожидания операций ввода-вывода. Квант для этой очереди выбирается таким образом, чтобы наибольшее количество процессов успело или завершить или выдать запрос ввода-вывода. Если за выделенный квант процесс не сделал ничего, то он

помещается в низкий уровень.

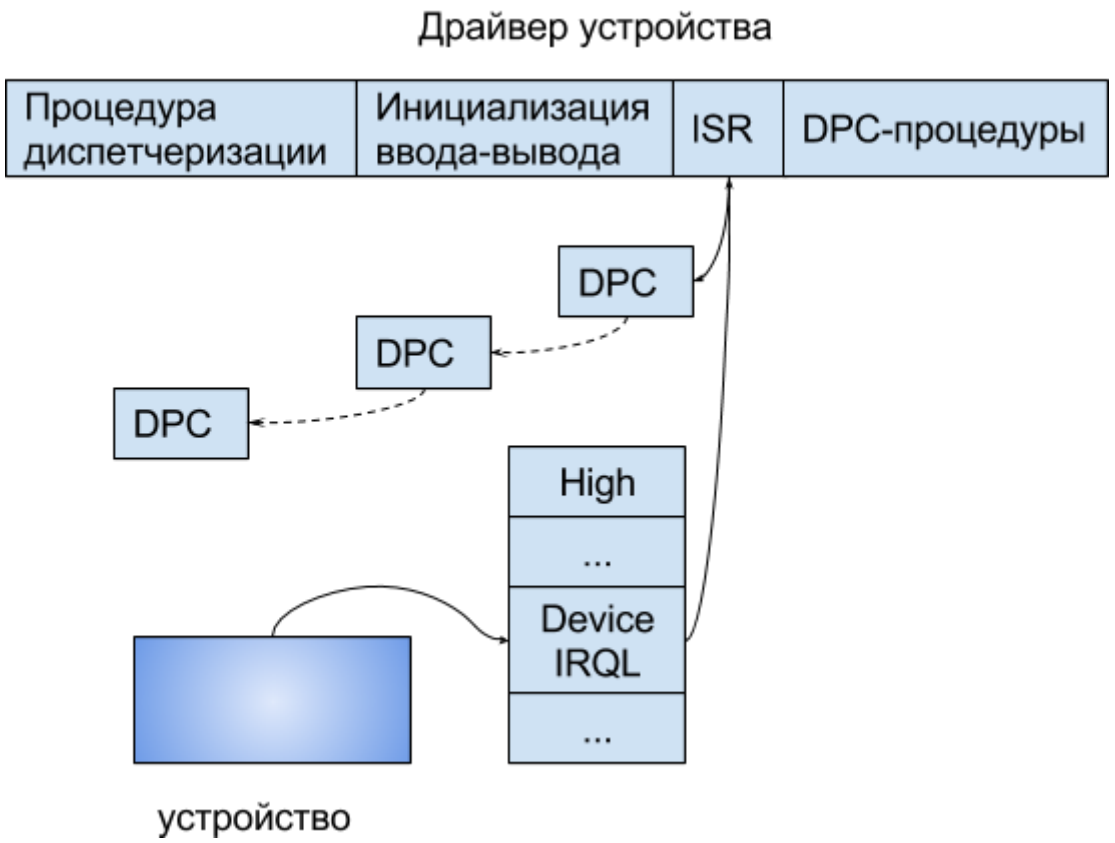


Windows и Unix - операционные системы разделения времени с динамическими приоритетами и вытиснением, т.е. более приоритетный процесс вытесняет менее приоритетный. В Windows происходит путем явного выявления...

2. Программные принципы вводом-выводом. Аппаратные прерывания. Обработка запроса ввода-вывода. Обслуживание прерываний в ОС Windows и DPC

НЕ ДОПИСАНО про приницпы вводом выводом???

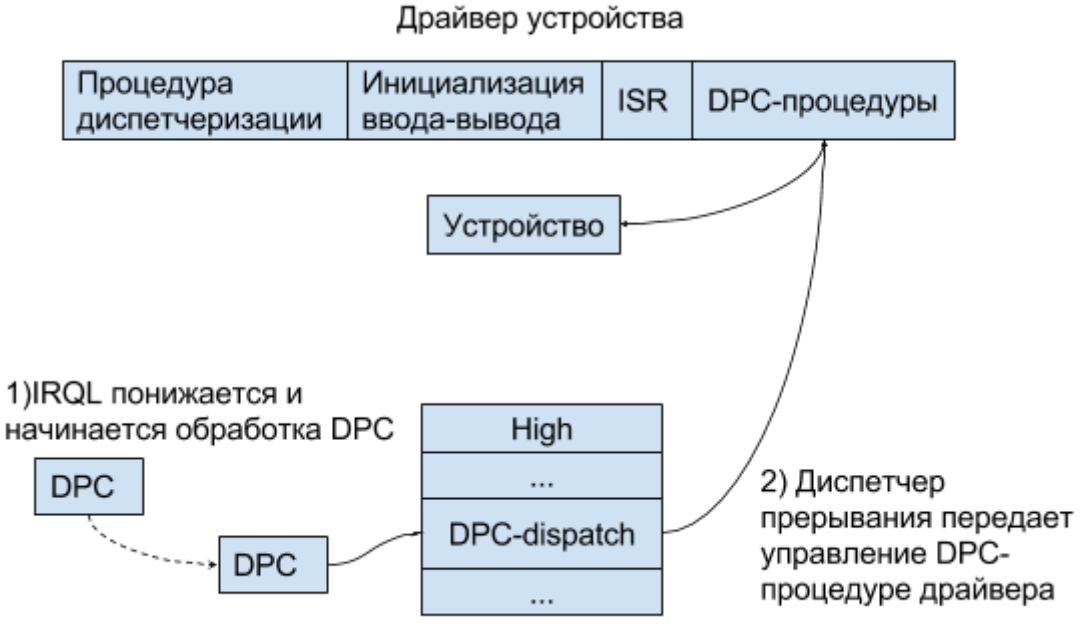
Аппаратное прерывание возникает в системе **при завершении** операции ввода-вывода. Операциями IO управляют контроллеры – программно-управляемыми устройствами. От драйвера они получают команду, выполняя которую они и управляют устройством. Задача устройства – передача данных. Завершая передачу данных, контроллер устройства генерирует сигнал, который поступает в x86 в контроллер прерываний.


Драйвер устройства
Процедура диспетчеризации
Инициализация ввода-вывода
ISR
DPC-процедуры
DPC
DPC
DPC
High
...
Device IRQL
...
устройство

Все прерывания от внешний устройств имеют уровень выше, чем DPC_dispatch. Это говорит о том, что прерывания от внешнего устройства может прервать любую работу на процессоре. Interrupt Service Routine должна выполнять минимум действий по обслуживанию прерывания от своего устройства – а именно – сохранять информацию о состоянии прерывания и отложить передачу данных и выполнение других некритичных во времени операций до понижения уровня IRQL до DPC_dispatch.

Все заканчивается постановкой в очередь DPC-объекта, содержащей адрес функции, которая должна вызвать код ядра для завершения обработки прерывания. ... По умолчанию ядро помещает DPC-объект в конец очереди того процессора, в котором выполнялось соответствующее ISR (interrupt service routine), но драйвер устройства может изменить приоритет процесса (по умолчанию DPC имеет средний приоритет), а также направить DPC к конкретному процессору.

DPC – Deferred Procedure Call. Напоминает механизм tophalf-bottomhalf call.



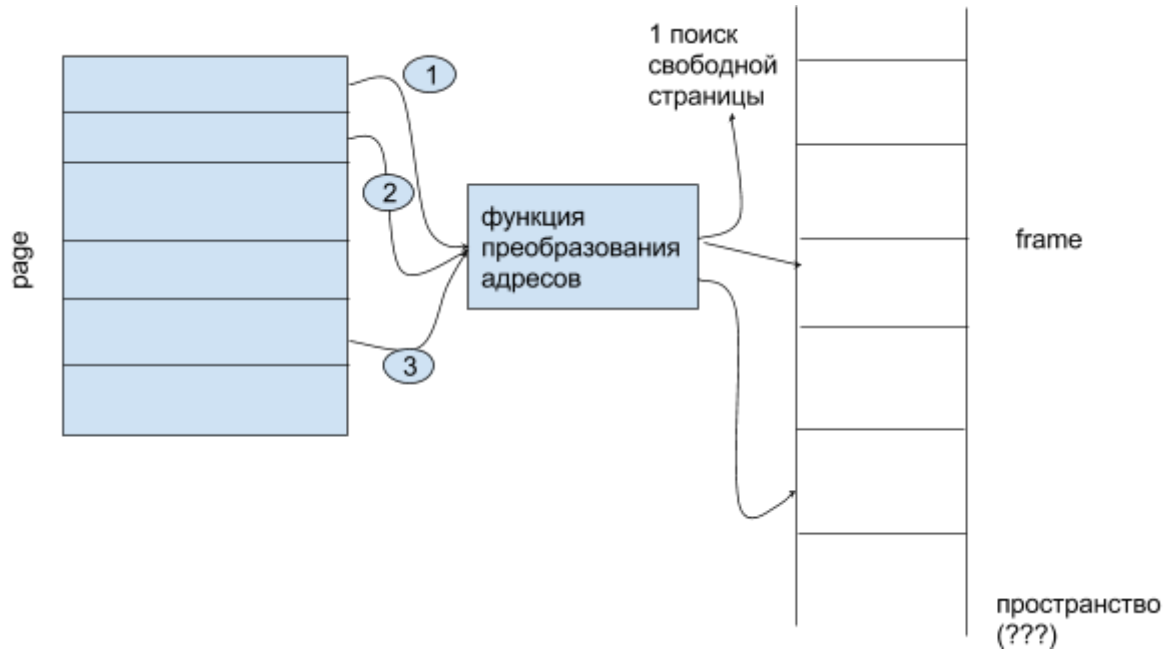
После выполнения ISR уровень IRQ(?) понижается до DPC_dispatch и диспетчер передает управление DPC-процедуре драйвера. DPC-процедура начинает обработку … после чего заканчивает обработку прерывания.

DPC – ничто иное, как отложенное действие по обработке прерывания, выполняющийся на более низком уровне привилегий. Все рассуждения, выполняемые для аппаратных прерываний Linux, экстраполируются на Windows.

Билет 12

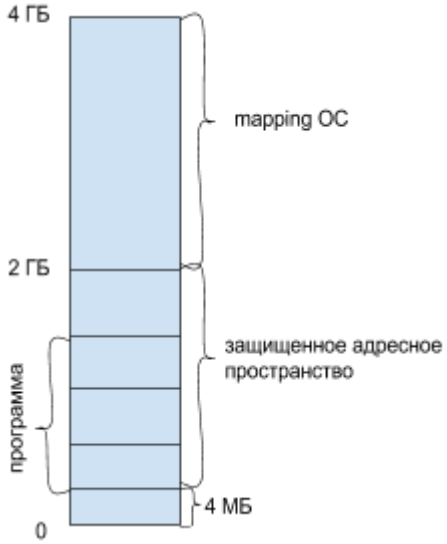
1. Виртуальная память: распределение памяти страницами по запросам, схема с гипер страницами, свойство локальности, анализ страничного поведения процессов, рабочее множество. Управление памятью страницами по запросам в архитектурах x86 - расширенное преобразование (PAE).

Виртуальная память - память, размер которой значительно превосходит размер физического адресного пространства.



3 метода преобразования адресов

1. Прямое отображение

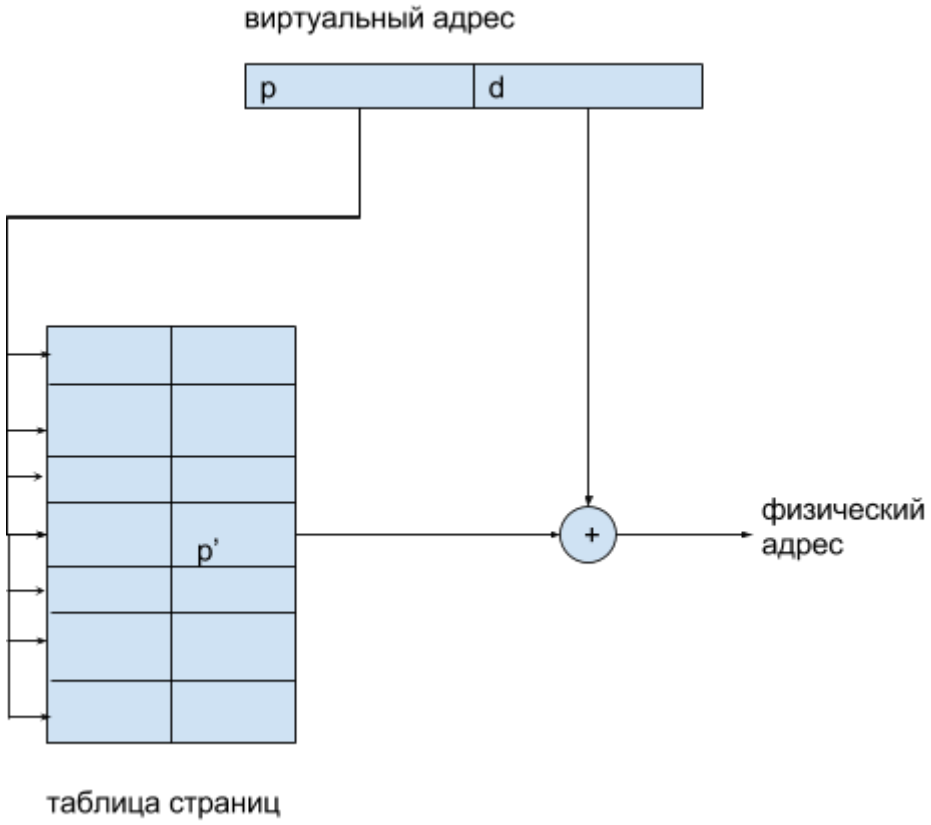


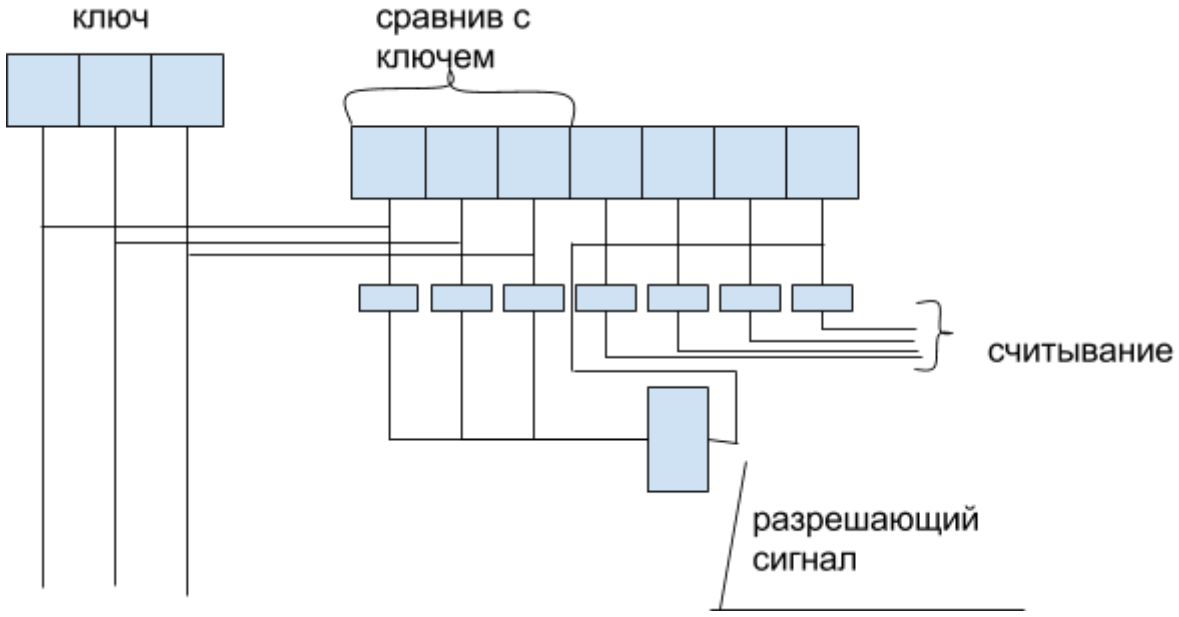
4МБ не используются, они зарезервированы для реального режима. Таблица страниц находится в ОЗУ.

2. Ассоциативное отображение

Ассоциативная память: в системе должна существовать память другого типа. В данном случае номер страницы играет роль ключа. Делится на номер страницы P и смещение D.

Выборка из параллельной ассоциативной памяти осуществляется за 1 такт, по ключу. В данном случае ключ - номер страницы. Он сравнивается со всеми полями одновременно, и при совпадении осуществляется выборка



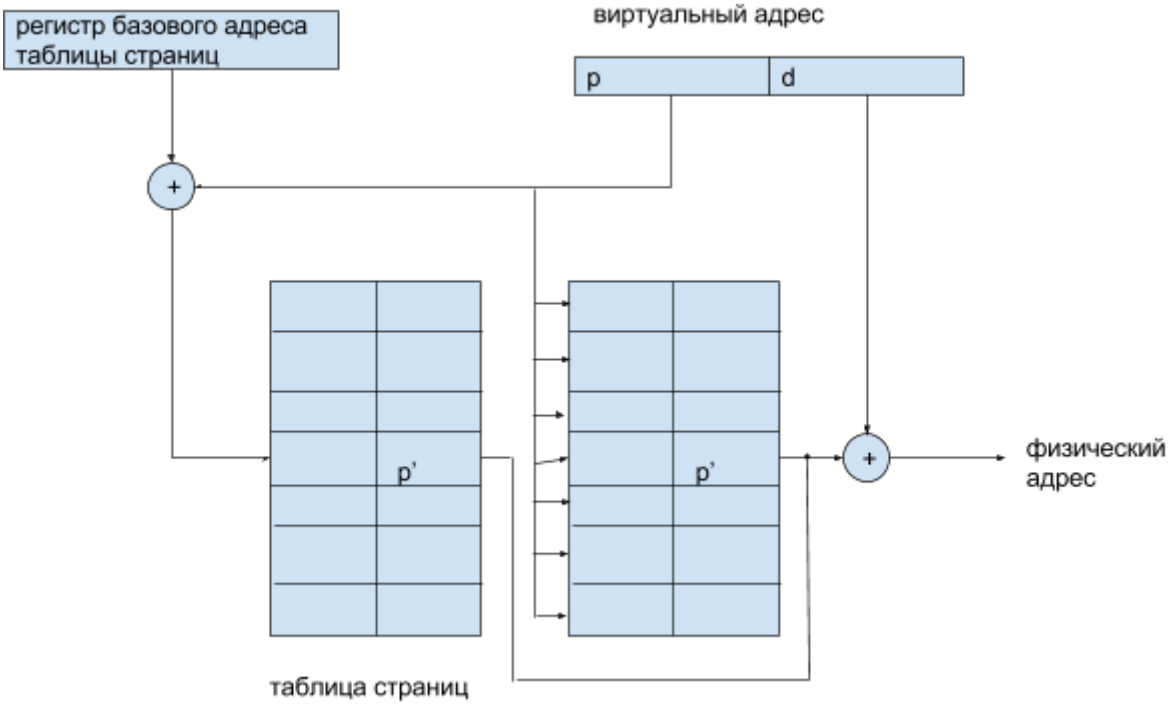


3. Ассоциативно-прямое отображение

В системе будет столько таблиц страниц, сколько процессов. Они описывают адресное пространство (виртуальное), и отображается факт загрузки страницы в физическую память.

Для такого отображения должен присутствовать регистр базового адреса таблиц страниц. Сначала страница ищется в ассоциативной памяти, и только потом в таблице страниц.

Ассоциативный кеш - небольшого размера: 8-16 строк, но он позволяет получить примерно 90% скоростных показателей, из-за того, что в кеше хранятся адреса страниц.

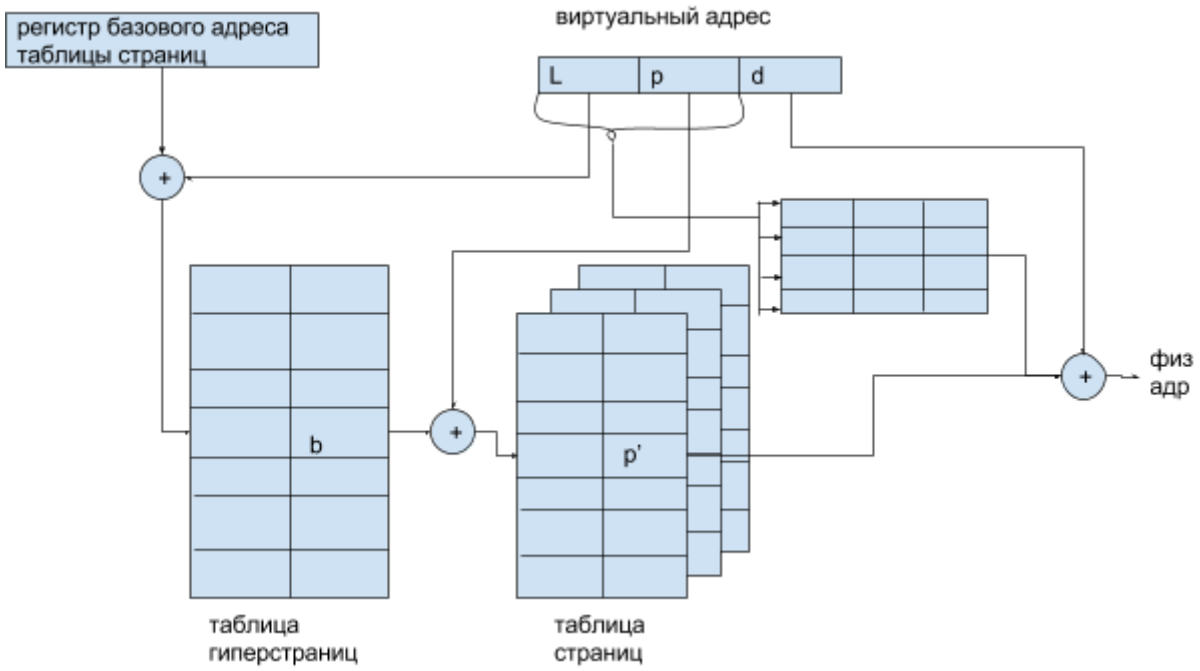


В процессорах Intel реализована именно эта модель. Там есть TBL - Translation Look-Aside Buffer. Это частично ассоциативный кеш, который больше, чем тот, что был обсужден.

Двухуровневая страничная организация

При увеличении размера виртуальной памяти пропорционально увеличиваются таблицы страниц. Это приводит к загромождению оперативной памяти таблицами страниц. Таблиц страниц столько, сколько процессов в памяти. Чтобы сократить расходы на это хранение, в IBM/360 была предложена двухуровневая страничная организация. Их стали делить на гиперстраницы, а гиперстраницы делятся на страницы.

Таблица гиперстраниц на каждый процесс одна, но таблиц страниц столько же , сколько и страниц. Из вектора гиперстраницы мы берем адрес, соответсвующий адресу страницы. Очевидно, что есть ассоциативный кеш.



В таблице гиперстраниц загружаются только актуальные таблицы страниц. Время увеличивается, т.е. за эффективное хранение данных мы платим увеличением времени преобразования. Всегда мы должны отвечать на вопрос: почему это сделано и на что тратятся ресурсы.

Проблема коллективного использования страниц была решена

введением флага Copy On Write. Очевидно, что площадкой инновационных

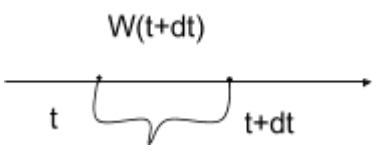
вещей является Unix / Linux.

Страничное поведение процессов

Оно исследовалось с точки зрения производительности системы, то есть каким образом страничное поведение процессов влияет на нее и от чего СПП зависит.

На СПП, то есть на количество страничных прерываний, влияет объем памяти, выделяемой процессу. Очевидно, что при выполнении процесса такое увеличение является негативным фактором, на фоне которого падает производительность системы. В результате исследований было определено, что для процесса могут быть определены критические значения объема выделяемой процессу памяти такие, что небольшое увеличение объема существенно не меняет числа СП, а даже небольшое уменьшение наоборот приводит к их резкому возрастанию.

Теория рабочего множества

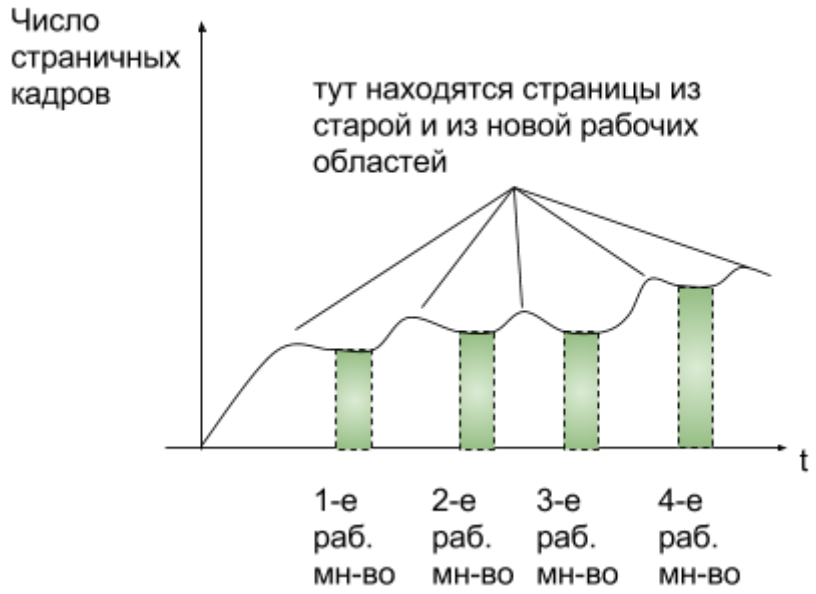


В 1968 году Дэннинг предложил в качестве локальной меры производительности взять число страниц, к которым программа обращается за время dt. Размер рабочего мн-ва W (working set) (t,dt) является монотонной функцией от dt, т. е. При увеличении интервала t,dt W будет стремиться к некоторому пределу l, который и определяет кол-во страниц, необходимых процессу для эффективного выполнения.

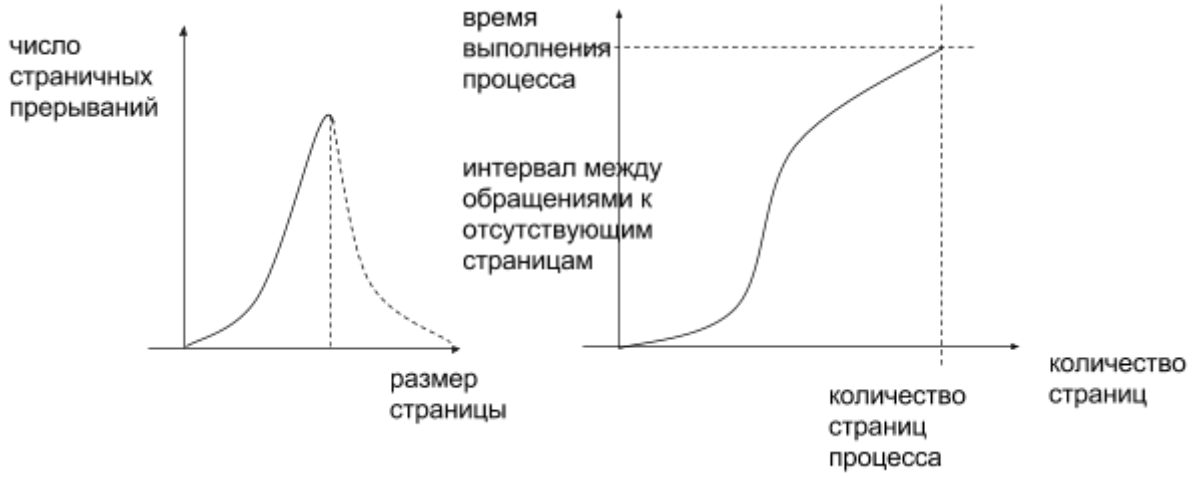
Под эффективным выполнением понимается выполнение программы без

страничных прерываний. Другими словами, если процессу удалось загрузить все свое рабочее мн-во, то некоторое достаточно небольшое время процесс будет

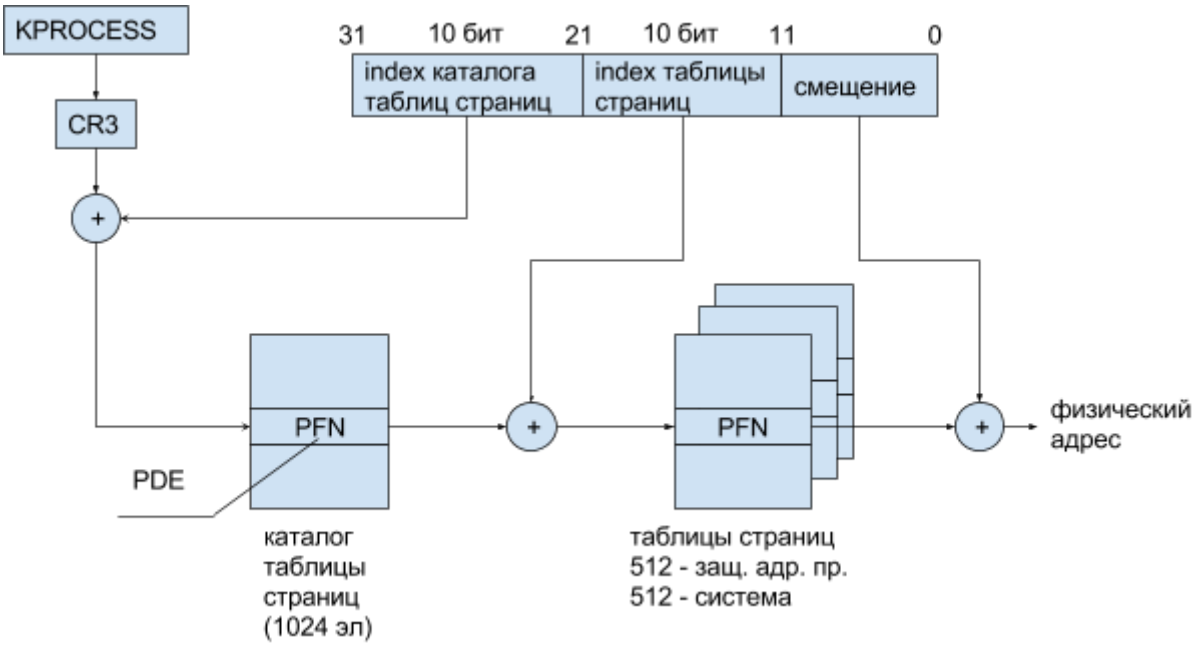
выполняться без страничных прерываний. Если же у процесса нет такой возможности, то он будет постоянно запрашивать нужные страницы, т. е. Будет осуществляться подгрузка и загрузка одних и тех же страниц – явление трешинга (thrashing).



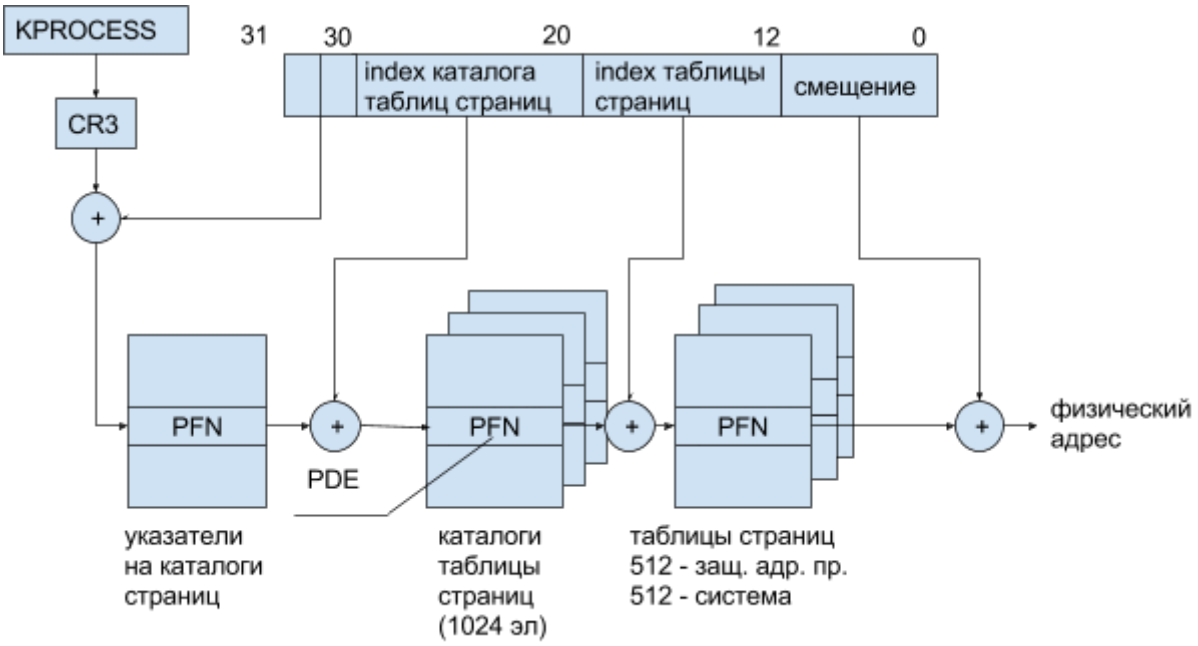
Рабочее мн-во – это набор страниц, исп. CPU в текущий момент времени. Можно считать, что процесс, выполняющийся в среде виртуальной памяти реально имеет возможность использования в течение некоторого времени произвольного короткой природы только определенной подобласти своего адресного пространства. Задача менеджера памяти в том, чтобы оставлять те страницы памяти, которые используются. Благодаря этому максимизируется процент страничных удач – hit rate. В Microsoft под этим понимается понятие квоты – и при резком увеличении числа страничных прерываний увеличивает ее, скажем, на 5 страниц.



PAE



Давала она такую, но на консультации говорила что адрес разбивается на 4 части, т.е. возможно потребует такое



2. Отложенные действия: очереди работ, особенности выполнения, системные вызовы для создания очереди работ и постановки ее в очередь на выполнение.

workqueue – их сравнивают с тасклетами. Естественно, у них есть общие черты, но различий больше.

Они также, как и тасклеты, позволяют выполнять действия по обработке прерываний в некоторое будущее время в виде отложенных действий, но главным отличием является то, что очереди работ могут блокироваться, в отличие от атомарного тасклета. Возможность эта связана с тем, что очереди работ выполняются

не в контексте прерывания, как тасклет, а в контексте спецпроцесса ядра – worker. Мало того, что очереди работ могут блокироваться, код ядра могут запросить, что выполнение функции очереди работ было отложено на заданный интервал времени. Коротко можно сказать – тасклеты выполняются быстро и в режиме неделимости, а очереди работ могут иметь большие задержки и не должны быть атомарными.

Очереди описываются workqueue_struct в linux/workqueue.h. Начиная с 2.6.36 был внесены очень большие изменения в очереди работ, но интерфейс был сохранен. В ядрах до этой версии workqueue создавалась с помощью функций

struct workqueue_struct *create_workqueue(const char *name);

struct workqueue_struct *create_singlethread_workqueue(const char *name);

В результате создаваемая очередь не имеет потоков и содержит только контекст для выполнения задач. Как уже сказано, эти API считаются устаревшими и правильно создавать очереди работ, используя функцию

int alloc_workqueue(char *name, unsigned int flags, int max_active);

Параметры:

1) name – имя очереди. В отличие от старых реализаций не создается потоком с таким именем.

2) flags – возможная комбинация флагов, которая определяет, как очереди работ будут выполняться.

3) max_active ограничивает число задач, которые могут выполняться одновременно.

#cat /proc/interrupts

#cat /proc/stat

ps -ef | grep kworker

Как пишут в более свежих статьях, предлагается полностью create_workqueue на alloc_workqueue, т. е. Выполнить … Существует потребность в старых API.

```
static int _init synthesizer_init(void)
{
	// регистрация обработчика прерываний
	int res = request_irq(synth.keyboard_irq, irq_handler, IRQF_SHARED, synth.name, &synth);
	if (res == 0)
	{
		// создание очереди работ
		synth.wq = alloc_workqueue(«sound_player, WQ_UNBOUND,0);
		if (synth.wq)
			printk(„success“);
		else
		{
			free_irq(synth.keyboard_irq, &synth);
			…
			return _ENOMEM;
		}
	}
	else
	{
		…
```

```
                return res;
        }
static irqreturn_t irq_handler(int irq, void *dev_id)
{
        int loc_cur_scancode = inb(0x60);
        if (loc_cur_scancode & 0x80)
        {}
        else
        {
                …
                if (<..>==0)
                {
                        …
                        queue_work(synth.wq, &work); // work: struct work_struct
                }
        }
        return IRQ_HANDLER;
}
static void work_handler(struct work_struct *loc_work)
{
        …
        msleep(TIMEOUT);
        …
}
```

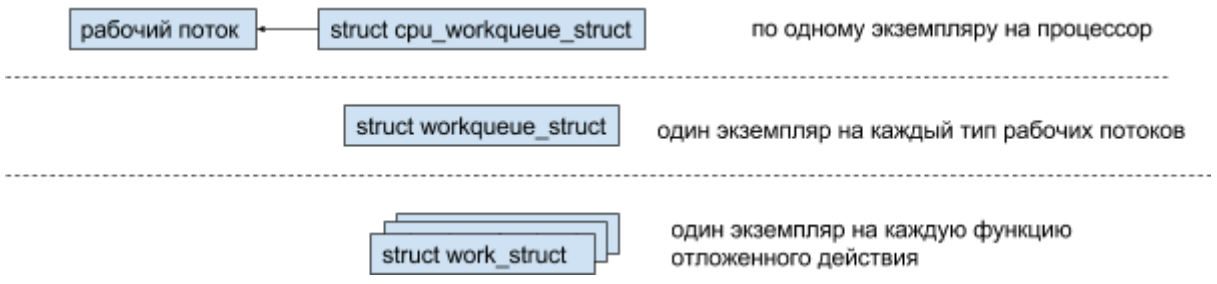
Разделяемые очереди

Во многих случаях драйверы устройств не заинтересованы в собственных очередях работ. Это обычно происходит, если задание ставится в очередь редко, время от времени. В этом случае более эффективно использовать разделение, т. е. Существующие в ядре очереди работ. Очевидно, что если использовать такие очереди, то необходимо об этом помнить. В частности, если мы используем разделяемую очередь работ, то мы не можем ее монополизировать в течение длительного периода времени.

Jiq – just in queue

```
staic struct work_struct jiq_work;
/* initialization */
INIT_WORK(&jiq_work, jiq_print_wq, &jiq_data);
int schedule_work(struct work_struct *work);
```

следует отметить, что функции используются как с разделяемой очередью, так и обычной.



```
struct workqueue_struct
{
        unsigned int flags;
        union
        {
                struct cpu_workqueue_struct_percpu *pcpu;
                struct cpu_workqueue_struct *single;
                unsigned long v;
        } cpu wq;
        struct list_head list;
        char name[]; //workqueue name
};
```

Очередь – набор тасков. Их может быть в этой очереди много. Про поля надо читать в man.

```
Sturct work_struct
{
        atomic_long_t data;
        struct list_head entry;
        work_func_t func; //обработчик нижней половины
#IFDEF CONFIG_LOCKED
        struct lockder_map lockder_map;
#endif
};
```

Флаги

На очереди работ определены следующие флаги:

- WQ_HIGHPRI – высокий приоритет. В результате задание помещается в начало очереди и запускается немедленно
- WQ_UNBOUND – непривязанная к конкретному циклу очереди. Очереди были разработаны таким образом, что они запускаются на том процессоре, на котором они были запланированы обработчиком прерывания. Это ради лучшего использования кеша. Флаг отключает такое поведение. Комбинация с WQ_HIGHPRI удаляет эту очередь работ workqueue из режима параллельного управления полностью. Сам флаг имеет смысл только для привязанных очереди и означает отказ от дополнительной организации параллельного управления. Речь о том, что у нас много процессоров (до четырех) и для более интенсивного обслуживания очередей работ система может распределять очереди работ по разным процессорам и тогда, естественно, будет параллельное выполнение.
- WQ_CPU_INTENSIVE –
- WQ_RESCUER
- WQ_FREEZABLE – может быть заморожена. Очевидно, что очередь, запускающая задание как часть … , этот флаг устанавливать не должна.
- WQ_NON_REENTRANT – гарантирует, что разные задания очереди будут выполняться параллельно.

Для того, чтобы поместить задание в очередь работ (task workqueue), следует заполнить структуру work_struct. Это может быть сделано с помощью следующих макросов:

1. статически с помощью макроса declare_work, которому передается имя и обработчик нижней половины:

DECLARE_WORK(name, void(*function)(void*));

2. динамически – используется два макроса:

INIT_WORK(struct work_struct *work, void (*function)(void *)); – если структура определяется в первый раз.

PREPARE_WORK(--) – если нужно изменить существующую структуру.

Имеются также две функции, отправляющие task в очередь.

Int queue_work(struct workqueue_struct *queue, struct work_struct *work);

int queue_delay_work(--,unsigned long delay);

0 – работа поставлена в очередь

другое значение – работа уже существует в очереди и не была заново поставлена.

Билет 14

1. Взаимодействие параллельных процессов: монопольное использование - реализация; типы реализации взаимоисключения. Мониторы - определение, примеры: простой монитор, монитор “кольцевой буфер” и монитор “читатели-писатели”.

Часть кода, в которой
происходит обращение к такой переменной – критическая секция. Для того, чтобы не происходило потери данных, необходимо обеспечить монопольный доступ к разделяемой переменной. Монопольный доступ обеспечивается методом взаимоисключения, т. е. Если какой-то процесс находится в критической секции по данной разделяемой переменной, то никакой другой процесс не может войти в эту же критическую секцию по данной переменной.
Существует несколько подходов к реализации взаимоисключений:
чисто программный способ

```
main:
flag = 0;
parbegin
      p1, p2;
parend;
```

p1	p2

while(1)
{
  while (flag);
  flag = 1;
  CR1;
  flag = 0;
  PR1;
  …
}
	while(1)
	{
	  while (flag);
	  flag = 1;
	  CR2;
	  flag = 0;
	  PR2;
	  …
	}

Активное ожидание на процессоре – выполнение while(flag); к примеру.
С двумя флагами: ничего не поменяется.

p1	p2

while(1)
{
  while (flag2);
  flag1 = 1;
  CR1;
  flag1 = 0;
	while(1)
	{
	  while (flag1);
	  flag2 = 1;
	  CR2;
	  flag2 = 0;

```
    PR1;
    …
}
```

```
    PR2;
    …
}
```

p1	p2

while(1)
{
flag1 = 1;
while (flag2);
CR1;
flag1 = 0;
PR1;
…
}
	while(1)
	{
	flag2 = 1;
	while (flag1);
	CR2;
	flag2 = 0;
	PR2;
	…
	}

Deadlock: оба процесса не могут продолжить свое выполнение.

Алгоритм Деккера

Он решил поставленную задачу только для двух процессов введением переменной «чья очередь».

Turn1, turn2: logical;

que: int

```
P1: while(1)
{
    turn1=1;
    while (turn2==1)
    {
        if (que==2)
        {
            turn1=0
            while (que==2);
            turn1=1;
        }
    }
    CR1;
    turn1=0;
    que=2;
    PR1;
}
```

```
P2: while(1)
{
    turn2=1;
    while (turn1==1)
    {
        if (que==1)
        {
            turn1=0
            while (que==1);
            turn2=1;
        }
    }
    CR2;
    turn2=0;
    que=1;
    PR2;
}
```

```
que = 1;
turn1 = 0;
turn2 = 0;
parbegin
   p1; p2;
parend;
```

Bakery Algorithm

Решает проблему критической секции для n числа объектов. Также известен алгоритм Лампорта, который усовершенствовал решение, предложенное Дейкстрой. Этот же человек предложил нам семафоры.

Покупатель на входе получает номер в очереди. Соответственно, он получает номер самой большой среди выданных. Будет обслуживаться клиент с наименьшим номером. Но в дверь могут войти сразу два покупателя одновременно.

```
// Инициализация
Typedef char boolean;
...
shared boolean choosing[n];
shared int num[n];
...
for (j=0; j<n; j++)
{
   num[j]=0;
}
//Конец инициализации
//----------------
// entryprotocol
//Choose number
choosing[i]=true;
num[i]=max(num[0],...,num[n-1])+1;
choosing[i]=false;
//for all other processes
for(j=0; j<n; j++)
{
   //wait if process is currently choosing
   while (choosing[j]) {/*nothing*/};
   while((num[j]!=0)&&(num[j],j)<num[i],i)
      if ((num[j]>0)&&((num[j]<num[i])||(num[j]==num[i])&&(j<i)))
      {
         while (num[j]>0);
      }
}
/*critical section*/
```

Решает проблемы n асинхронных ЭВМ, связанных друг с другом только через общую память. Каждая ЭВМ выполняет циклическую программу из 2 секций: критической и некритической.

Задача программы состоит в следующем:

1. В произвольный момент времени самое большое -- 1 ЭВМ находится в критической секции.
2. Каждая ЭВМ должна в конце концов получить возможность войти в КС, если не произойдет останова.
3. Каждая ЭВМ может совершить останов в некритической секции выполняемой программы.

4. Нельзя сделать никаких предположений относительно скорости функционирования ЭВМ.

В решении Лампорта n процессоров, каждый из них имеет свою память, может обращаться за чтением к любой другой памяти. Особенности следующие:

1. Соблюдаются 3 и 4, но только операция записи должна выполняться правильно; чтение -- что угодно (?)
2. На входе процесс проверяет другие процессы и ждет каждый процесс, который имеет меньший номер.
3. Первоначально массив номеров пустой
4. Процесс p_i ждет, пока он не будет иметь наименьший номер среди всех желающих войти в критическую секцию.
5. До for выбор номера считать критическим действием.
6. Если p_i не пытается войти в критический участок, то num[i] = = 0.

У Лампорта:

if ((num[i]!=0) && ((num[j],j)<(num[i],i))
{...}

(a,b)<(c,d) истина, если a<c или a=c, но b<d

Аппаратная реализация активного ожидания. Команда Test&Set.

Появившись еще в IBM370, данная команда является неделимой, и проверяет и устанавливает содержимое ячейки памяти (байта блокировки).

Если байт блокировки = 0 -- доступен
= 1 -- занят

Рассмотрим testandset, a,b, потом t.

test-and-set
flag,c1,c2: logical;

p1: while(1)
{
c1=1;
while(c1==1)
testandset(c1,flag);
cr1;
flag=0;
rr1;
}
	p2: while(1)
	{
	c2=1;
	while(c2==1)
	testandset(c2,flag);
	cr2;
	flag=0;
	rr2;
	}

main
{
flag=0;
parbegin
p1;p2;
parend;
}

flag = 1, когда любой из процессов находится в критическом участке.

t&s -- машинная команда. Есть активное ожидание, но нет бесконечного откладывания.

Её использование для проверки в цикле влечет циклическую блокировку (spin-lock, simple lock, simple mutex). Активно используется в Linux. Чаще всего эта команда возвращает предыдущее значение.

```
void spin_lock(spin_lock_t *c)
{
   while(test_and_set(*c) != 0)
   // ресурс занят
}
void spin_unlock(spin_lock_t *c)
{
   *c=0
}
```

Обедающие философы

Задача: за круглым столом 5 тарелок и 5 философов.

Палочки – это приборы. Тут и вся идея.

1. Философ берет обе вилки. Если ему удается, то он ест и кладет обе вилки.
2. Философ берет левую вилку, и если возможно – правую, удерживая левую, даже если правую взять невозможно
3. Философ берет левую вилку. Если в течение некоторого времени он не может взять правую, то он кладет левую, потом, по истечении какого-то времени он пытается снова повторить действие 3.

Первая ситуация – бесконечное ожидание, starvation. В этом случае может умереть один философ.

Вторая ситуация – deadlock, тупик. Все помрут.

Третья ситуация – leavelock – динамический тупик. В реальной жизни это можно наблюдать в виде thrashing.

```
var
   forks: array[1..5] of semaphore;
   i: integer;
begin
   i=5;
   repeat
      forks[i]:=1;
      i-=1;
   while i=0;
cobegin
1:begin
var
   left, right:=1..5;
   ...
end;
...
5:begin
var
```

```
    left,right:=1..5;
    left:=5; right:=1;
  repeat
    <размышляет>
    p(forks[left],forks[right]);
    <ест>
    v(forks[left],forks[right]);
  forever
  end; //5
  coend;
```

Мониторы

Монитор – это структура, состоящая из данных и подпрограмм, которые могут изменять или обращаться к этим данным. Монитор начинается ключевым словом monitor, они похожи на классы, хоть и появились они раньше. Монитор защищает свои данные, это заключается в том, что к данным можно обратиться только через подпрограммы монитора, т.е. монитор сам является ресурсом. Он гарантирует, что в каждый момент времени ресурсы монитора могут использоваться только одним процессом – захвативший монитор процессор – это процесс, находящийся в мониторе. Все другие процессы, пытающиеся захватить монитор – они вне монитора и стоят к нему в очереди.

Использование монитора приводит к снижению производительности – снижение может достигать 70ти %. Определены две операции – wait и signal.

Введен специальный тип переменных – condition, условие. Для каждой отдельной взятой причины, по которой процесс может быть переведен в состояние ожидания, назначается свое “условие”.

Рассмотрим: простой монитор, монитор читателя-писателя, кольцевой буфер

1. Простой монитор: обеспечивает выделение ресурсов произвольному числу процессов.

```
resource: monitor
var
  busy: logical;
  x: condition;
```

procedure acquire;
begin
  if busy then wait(x);
  busy := true;
end;
	procedure release;
	begin
	  busy:=false;
	  signal(x);
	end;

```
begin
  busy:=false;
end;
```

2. Кольцевой буфер: такой образ характерен для spooler-буферов.

```
resource: monitor;
var
```

```
bcircle: array[0,...,n-1] of type;
pos: 0..n; //тек.поз
j: 0..n-1; // заполняемая позиция
k: 0..n-1; // освобождаемая позиция
bufferfull, bufferempty:condition;
```

procedure producer (data:type);
begin
if pos=n then wait(bufferempty);
bcircle[j]:=data;
pos+=1;
j:=(j+1) mod n;
signal(bufferfull);
end;
	procedure consumer(var data: type);
	begin
	if pos = 0 then wait(bufferfull);
	data := bcircle[k];
	pos := pos-1;
	k := (k+1) mod n;
	signal (bufferempty);
	end;

```
begin
   pos:=0;
   j:=0;
   k:=0;
end.
```

3. Монитор читателя-писателя (Хоара) (Джеффри Рихтер “Windows для профессионалов” для лабораторных): там 4 функции: startread, stopread, startwrite, stopwrite. Два типа процессов: читатели (могут только читать) и писатели (могут изменять данные). Задача важна: в реальной жизни она обеспечивается в билетных системах, там, где есть конкретное указание “день-время-место”. Характерна для ядра ОС.

```
resource:monitor;
var
   nr: integer; //читатели
   wrt: logical; //писатель
   c_read, c_write : condition;
```

procedure startread;
begin
if wrt or turn(c_write) then
wait(c_read);
nr+=1;
signal(c_read);
end;
	procedure startwrite;
	begin
	if nr > 0 or wrt then wait(c_write);
	wrt := true;
	end;

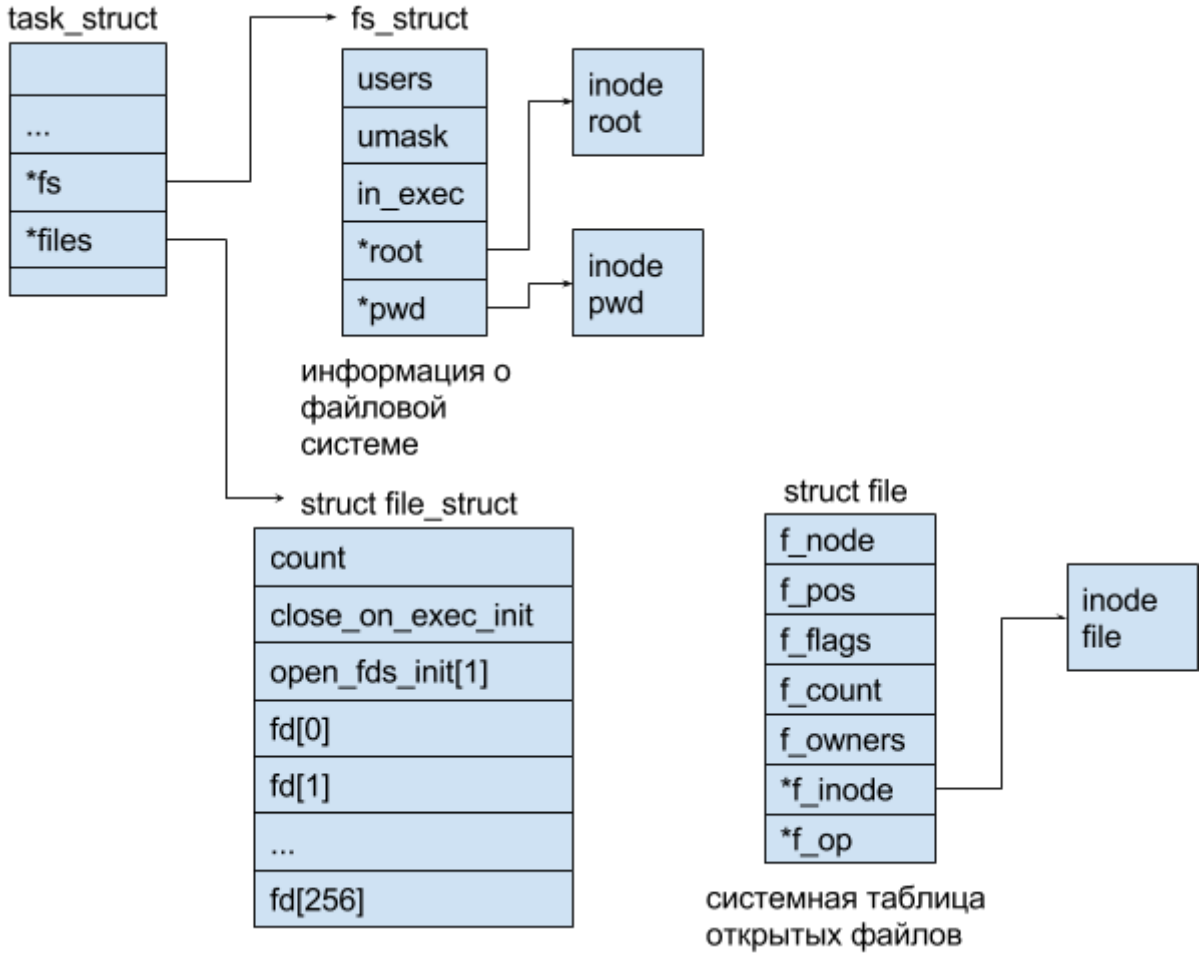
procedure stopread;
begin
nr-=1;
if nr = 0 then
signal(c_write);
end;
	procedure stopwrite;
	begin
	wrt:=false;
	if turn(c_read) then
	signal(c_read);
	else signal(c_write);
	end;

```
begin
   nr:=0;
   wrt:=false;
end.
```

2. Файловая система Linux - открытые файлы: структуры, связанные с процессом и struct file. Struct file_operations. Регистрация и дерегистрация файловых систем.

files_struct описывает открытые процессом файлы - эта структура доступна процессу.

Каждый открытый файл, открытый разными процессами или двумя разными вызванными open() создают два разных дескриптора.



Объект файл - это представление, которое хранится в RAM (struct file), создаваемый в результате сисвызова open() и удаляется в результате сисвызова close(). Все “файловые вызовы” - методы, связанные с files_struct_operations, т.к. несколько процессов могут одновременно открыть и использовать один и тот же файл, то для одного файла может существовать несколько объектов “файл”. Структура file определена в linux/fs.h. Эта структура используется исключительно ядром и никогда не используется прикладными программами (работающими в пространстве userspace). FILE определяется библиотекой glibc. Она в ядре нигде не используется. Как правило, указатели на структуру file называются filp. Структура fs_struct содержит информацию

о ФС, которая имеет отношение к процессу, о чем говорит то, что дескриптор процесса ссылается на эту структуру. Структура fs_struct определна в файле linux/fs_struct.h. Для версии 4.10 имеет следующий вид:

```
struct fs_struct
{
        int users;
        spinlock_t lock;
        seqcount_t seq;
        int umask;
        int in_exec;
        struct path root, pwd;
}
struct path
{
        struct vfsmount *mnt;
        struct dentry *dentry;
}
struct file_struct
{
        atomic_t count; // изменение счетчика должно происходить с помощью
                        // неделимых операций
        bool resize_in_progress;
        wait_queue_head_t resize_wait;
        struct fdtable _rcu *fdt;
        struct fdtable fdtab;
        // written part on a separate cache line in SMP
        spinlock_t file_lock cacheline aligned_in_smp;
        unsigned int next_fd; // следующий номер файлового дескриптора
        unsigned long close_on_exec_init[1]; // определяет число файлов, которое
                                             // должно быть закрыто
        exec()
        unsigned long open_fds_init[1]; // число открытых файлов
        unsigned long full_fds_bits_init[1];
        struct file rcu *fd_array[NR_OPEN_DEFAULT]; //массив дескрипторов файлов
}
```

spinlock используется для взаимных блокировок.

struct file:

```
struct file
{
        …
        struct path f_path;
        struct inode f_inode;
        const struct file_operations *f_op; // сисвызовы для работы с файлами
        spinlock_t f_lock;
        atomic_long_t f_count;
        …
```

```
        loft_t f_pos;
        ...
}
```

Любое устройство будет иметь дескриптор struct file в таблице открытых файлов (системной)

Методы read, write выполняют похожие действия. Одна копирует данные из, другая копирует данные в приложение, поэтому их прототипы схожи.

```
ssize_t read(struct file *filp, char *buff, size_t count, loft_t *offp);
ssize_t write(struct file *filp, char *buff, size_t count, loft_t *offp);
struct file_opetations
{
struct module *owner;
loft_t (*elseak)(struct file*, loff_t, int);
ssize_t (*read)(struct file*, const char _user*, size_t, loff_t*);
ssize_t (*write)(struct file *, const char _user*, size_t, loff_t *);
int (*open) (struct inode *, struct file*);
}
```

Структура file_operations перечисляет системные вызовы, которые оперируют с файловыми дескрипторами. Аналогично inode_operations.

Регистрация и дерегистрация ФС.

file_system_type - стурктура, описывающая ФС. На ФС она может быть только одна.

```
Struct file_system encfs_fs_type =
{
        .owner = THIS_MODULE;
        .name = «encfs»;
        .mount = encfs_mount;
        .kill_sb = encfs_kill_superblock;
        .fs_flags = FS_REQUIRES_DEV;
};
struct file_system_type my_fs_type =
{
        .owner = THIS_MODULE,
        .name = «my»,
        .mount = my_mount,
        .kill_sb = my_kill_superblock,
        .fs_flags = FS_REQUIRES_DEV,
};
```

Это один из способов заполнения структур, описанных в стандарте C99. Это связано с компилятором. GCC 2.95 поддерживает синтаксис C99.

```
Struct file_system_type {
        const char *name;           // Имя, доступное пользователю (/proc/filesystem)
                                    // Должно быть уникальным.
        int fs_flags;               // Используется для ФС, монтир. На блочное устр.
#define FS_REQUIRES_DEV 1    // единственный сохранившийся флаг
#define FS_BINARY_MOUNTDATA 2
```

```
#define FS_HAS_SUBTYPE 4
#define FS_USERNS_MOUNT 8  // Can be mounted by users root
#define FS_RENAME_DOES_D_MOVE 32768
        struct dentry *(*mount)(struct file_system_type *,
                                int, const char *, void*); // монтир.
        Void (*kill_sb) (struct super_block*); //размонтир.; необязательно
        struct module *owner;       // счетчик, необходимый для удержания модуля,
                                    // монтирующего ФС
                                    // Указывает на адресное пространство модуля (?)
        struct file_system_type *next;
        struct hlist_head fs_supers;
        struct lock_class_key s_lock_key;
        struct lock_class_key s_umount_key;
        struct lock_class_key s_vfs_rename_key;
        struct lock_class_key
        s_writers_key[SB_FREEZE_LEVELS];
        struct lock_class_key i_lock_key;
        struct lock_class_key i_mutex_key;
        struct lock_class_key i_mutex_dir_key;
};
```

Поскольку интерфейс VFS экспортируется, все ФС Linux реализуются как модули. Код, реализующий ФС, может быть динамически загружаемым или статически привязанным к ядру. После запоминания структуры file_system_type, надо ее зарегистрировать в таблице VFS, исп. Функцию регистрации register_filesystem.

```
Static int _init my_init(void)
{
        int ret;
        my_inode_cachep = kmem_cache_create («my_inode_cachep»,
                                                sizeof(struct my_inode), 0,
                                                (SLAB_RECLAIM_ACCOUNT |
                                                SLAB_MEM_SPREAD), NULL);
        if (!my_inode_cachep)
        {
                return _ENOMEM;
        }
        ret = register_filesystem(&my_fs_type);
        if (likely(ret==0))
                printk(KERN_INFO «Success \n»);
        else
                printk(KERN_ERR «Failed.Error-%d\n»,ret);
        return ret;
}
```

Рассмотрим printk(). Она выполняет форматированный вывод сообщений, является очень устойчивой, но ее нельзя использовать в начальные моменты загрузки ядра. Существует функция early_printk(), которая может выводить сообщения на ранних стадиях загрузки ядра. Функция printk() может указывать loglevel — уровень вывода

сообщений ядра, такие, как KERN_WARNING, KERN_INFO, KERN_ERR, KERN_DEBUG. Функции определены в linux/kernel.h.

```
struct void _exit my_cleanup(void)
{
        int ret;
        ret = unregister_filesystem(&my_fs_type);
        kmem_cache_destroy(my_inode_cachep);
        if (likely(ret==0))
                printk(KERN_INFO «Success \n»);
        else
                printk(KERN_ERR «Failed.Error-%d\n»,ret);
        return ret;
}
```

Любой модуль ядра заканчивается двумя макросами:

```
module_init(my_init);
module_exit(my_cleanup);
```

Билет 15

1. Тупик. Определение тупиковой ситуации для повторно используемых ресурсов, 4 условия возникновения тупика; обход тупиков - алгоритм банкира.

Тупики – это ситуация, возникающая в результате монопольного использования разделяемых ресурсов, когда процесс, владея ресурсом, запрашивает другой ресурс, занятый непосредственно или через цепочку запросов другим процессом, который со своей стороны запрашивает и ожидает освобождения ресурса, занятого первым процессом.

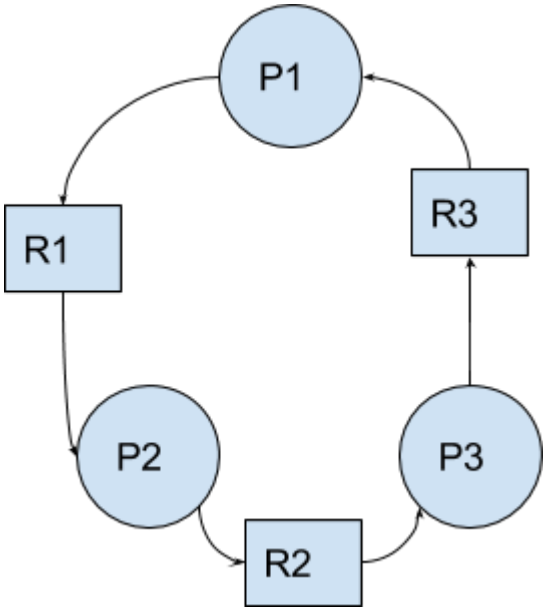
Повторно используемые ресурсы характеризуются следующими способами:

1. Число ресурсов единиц ограничено
2. Изменение числа единиц ресурса является исключением и следствием особых ситуаций в системе, например, выхода из строя внешнего устройства

Условие возникновения тупиков

Условия были сформулированы Ханвендером и являются ключом к борьбе с тупиками. Для возниковения тупиковой ситуации необходимо и достаточно 4 условия:

1. Взаимоисключение, mutual exclusion – когда процессы монопольно используют предоставляемые им ресурсы.
2. Ожидание, hold and wait – процессы удерживают занятые ресурсы, ожидая получения дополнительных ресурсов, которые им необходимы для продолжения выполнения.
3. Непереразделяемость – ресурсы нельзя отобрать у процесса до его завершения или освобождения ресурса самим процессом.
4. Круговое ожидание – возникновение в замкнутой цепи запросов процессов, в которой каждый процесс занимает ресурс, необходимый следующему в цепочке процессу для продолжения выполнения. (см. картинку)



Существуют три основных подхода к проблеме борьбы с тупиками:

1. Исключение самой возможности возникновения тупиков
2. Недопущение или предотвращение или обход тупиков
3. Обнаружение и восстановление работоспособности системы

Второй метод – обход – предполагает их существование, но предлагает действовать в обход, чтобы их не происходило. Наиболее известный алгоритм – алгоритм Банкира. Банкир обладает некоторым капиталом, выдавая ссуды. Заемщики, получив ссуду, часто обращаются за дополнительными средствами, чтобы вернуть полученную ссуду. Очевидно – банкир должен быть осмотрительным, чтобы не разориться и перераспределять имеющиеся у него средства таким образом, чтобы заемщики деньги возвращали. В качестве банкира выступает менеджер ресурсов. Заемщики – процессы, делающие заявки на ресурсы. (...) Процесс не может затребовать ресурсов больше, чем он указал в своей заявке. Это позволяет менеджеру ресурсов проводить предварительный анализ ситуации, возникающей в системе, с целью недопущения тупика.

Алгоритм банкира имеет ряд ограничений: для выполнения алгоритма необходимо выполнение условий:

1. Процесс не может требовать ресурсов больше, чем их есть у системы.
2. Процесс не может получить ресурсов больше, чем указано в его заявке.
3. Число процессов в системе фиксировано.
4. Количество распределенных ресурсов каждого класса не может превышать количество единиц ресурса каждого класса в системе.

Менеджер ресурсов анализирует ситуацию в системе, при этом она может измениться в результате запроса процесса на ресурс. Каждый запрос проверяется по отношению к количеству ресурсов в системе. Менеджер ресурсов ищет последовательность процессов, которая может завершиться. Если она есть, то система находится в безопасном состоянии.

	Кол-во выдел ед. ресурсов	Количество свободных единиц	Заявка
P1 P2 P3	1 3 5		4 5 9
		2	

Второй процесс не может запросить больше двух единиц ресурса, так как его максимальная потребность – 5, и он уже получил 3. P2 сможет завершиться и освободить занимаемые единицы ресурса. Освобожденные единицы ресурса в сумме с свободными единицами ресурса смогут обеспечить максимальную потребность у P1, чтобы он тоже освободился, также впоследствии с третьим. (..) Однако, если текущее состояние является надежным, то это не обязательно распространяется на последующие. Например, если система в предыдущей таблице перейдет в состояние таблицы 2:

	Кол-во выдел ед. ресурсов	Количество свободных единиц	Заявка

P1 P2 P3	2 3 5		4 5 9
		1	

Состояние является безопасным, если существует последовательность процессов такая, что:

- Первый процесс последовательности завершится и вернет системе занимаемые им единицы ресурса.
- Второй процесс последовательности завершится, если первый завершится и вернет системе все занимаемые им единицы ресурса, что в сумме со свободными единицами позволит удовлетворить его максимальную потребность в ресурсе
- И так далее... iй процесс последовательности завершится, если все предыдущие завершаться, и ....

Таким образом, всякий раз менеджер ресурсов должен найти успешно завершающуюся последовательность процессов, и только в этом случае запрос процесса может быть удовлетворен. Система удовлетворяет только те запросы, при которых ее состояние остается надежным. Запрос процесса, приводящий к ненадежному состоянию отложен до благоприятного момента. Так система поддерживается в надежном состоянии, и рано или поздно (за конечное время) все запросы будут удовлетворены и процессы смогут завершить свою работу. Для признания состояния системы надежным надо будет исследовать n! Последовательностей, т.е. крайне затратный алгоритм. В силу этого и ограничений алгоритма он имеет только теоретическое значение, так как показывает путь решения проблемы обхода тупика. Есть версии, которые сокращают число проверок до n^2 и больше.

Наиболее часто употребляемым алгоритмом является алгоритм Хабермана с порядком O(n^2). Сокращение количество проверок достигается за счет отказа от необходимости выделять запрашиваемые ресурсы согласно порядку следования безопасной последовательности распределения. Формулировка:

- Анализируется совокупность процессов, пока она не пуста.
- В цикле выполняются: найти процесс А в последовательности Эс, который может завершиться. Если такой есть, то вывести его оттуда, отобрать у него все ресурсы в пул ресурсов.

  S[0,..,r-1] где r – число единиц ресурса
  S[i] = r-I для всех i:0<i<r
  Если процесс, заявивший с единиц ресурса и
  удерживает h единиц ресурса
  Запросить единицу ресурса
  S[i] – для всех i: 0<=i<c-h

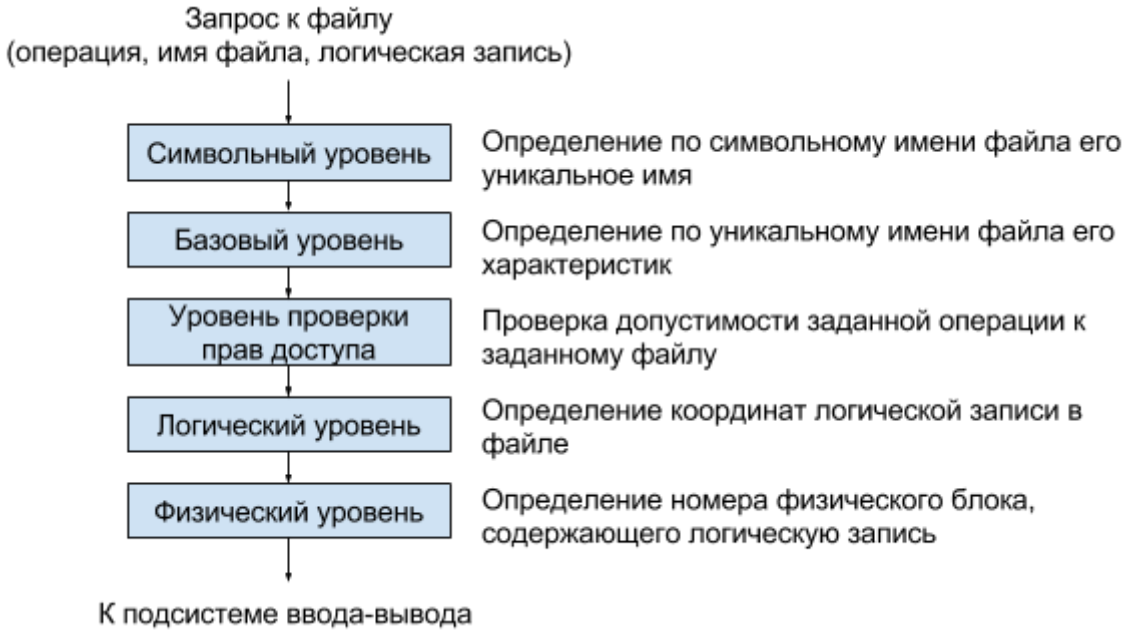
Если какой-то из s[i] становится отрицательным, то ситуация опасна относительно тупика.

Это все показывает, что при анализе системы можно не допустить тупика.

2. Файловые системы. Иерархическая структура файловой системы. Задачи уровней файловой системы. Доступ к файлам - работа с директориями: struct dentry, структура inode каталога. Пример, демонстрирующий доступ к файлу.

Файл – некая поименованная совокупность данных. В UNIX файл – это контейнер для хранения данных. С точки зрения пользователя интерфейс файловой системы объединяет системные вызовы и утилиты, при помощи которых пользователь или приложения могут выполнять различные действия над файлами.

Принято выделять несколько уровней файловой подсистемы:



Файловая система – некоторая организация данных и метаданных на устройствах хранения информации.

Особенностью ОС Unix является поддержка большого числа ФС (до 50 в Linux). Есть родные ФС (ext*, ufs, nfs) и поддерживаемые (ms-dos, ntfs). Все ФС составляют дерево каталогов и файлов. Для Unix характерна иерархическая организация дерева. Эти каталоги в качестве поддиректории могут содержать разные ФС

Дерево каталогов и файлов может добавляться новыми ветвями по мере монтирования новых ФС. Каждая ФС монтируется к какому-то каталогу, и этот каталог является точкой монтирования. Если каталог был не пустой и мы к нему подмонтировались, то его содержимое никуда не исчезает. В Linux всегда существует корневая ФС („/“). В FAT – корневой каталог. В поддиректории home может быть смонтирована ФС другого типа, т. е. Любая ФС, которая поддерживается ОС, т. е. Те, для которых определена структура File System Type.

I-ноды.

Информация о файле часто называется метаданными. Иными словами, inode – метаданные. Система при доступе ищет точный inode в таблице inode. (...) В результате пользователь получает доступ к файлу по его inode, но пользователь указывает имя файла. Например:

#include <sys/types.h>

#include <sys/stat.h>
#include <fcntl.h>
int open(const char *pathname, int flags);
int open(const char *pathname, int flags, mode_t mode);
// ими возвращается unsigned short int – и это значение используется в функциях типа lseek(..)
// аргумент pathname используется для поиска вхождения dentry для данного файла в кеше каталогов (dentry cache)

Для доступа к inode не требуется имя файла, но для того, чтобы по имени получить доступ, надо сопоставить имя с его inode. Посмотрим структуру айнод-каталога:

Inode number: 3470036	
. (DOT)	3470036
.. (DOT)	3470017
Folder1	3470031
File1	3470043
File2	3470023
Filder2	3470024
File3	3470065
...	

Структура inode-каталога состоит просто из имен к inode, соотв. этим файлам и поддиректориям.

Mode {umode_t imode}
Owner info {uid_t i_uid; gid_t i_gid}
Size
Time stamps {time_t i_atime, i_mtime, i_ctime}
Direct blocks
Indirect blocks
Double direct
Triple direct

Основные поля структуры inode (версия 4.10)

```
struct inode
{
        umode_t imode;
```

```
        unsigned short i_opflags;
        kuid_t i_uid;
        kuid_t i_gid;
        …
        const struct inode_operations *i_op;
        struct super_block *i_sb;
        struct address_space *i_mapping;
        …
        dev_t i_rdev;
        loff_t i_size;
        struct timespec i_atime;
        struct timespec i_mtime;
        struct timespec i_ctime;
        …
        atomic_t i_count;
        …
        const struct file_operations *i_fop;
        struct address_space i_data;
        ...
}
Struct dentry
{
        …
        struct dentry *d_parent; //родительская директория
        struct qstr d_name; //имя dentry
        struct inode *d_inode; //соотв. Inode
        const struct dentry_operations *d_op;
        struct super_block *d_sb; //Связанный суперблок
        union
        {
                struct list_head d_lru /* LRU список */
        wait_queue_head_t *d_wait;
        }
        struct list_head d_child; //подкаталоги в списке родителя
        struct list_head d_subdirs; //подкаталоги
        ...
}
```

Билет 16

1. Виртуальная память. Сегментно-страничное управление памятью, схема. Достоинства и недостатки. Сравнение со страничным. Доп. вопросы: почему "по запросам"? какой из видов используется сейчас? как решили проблемы с коллективным управлением(флаг copy on write)?

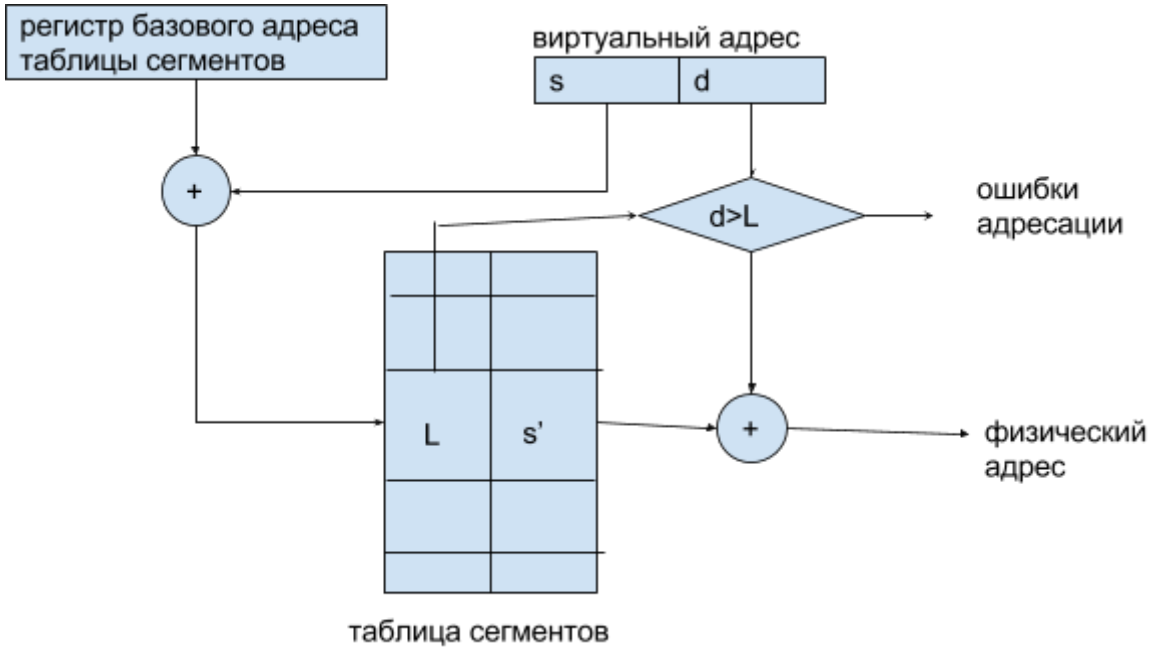
Виртуальная память - память, размер которой значительно превосходит размер физического адресного пространства.

Управление памятью сегментами по запросам.

Является исправимым исключением. Обрабатывая его, система переходит в режим ядра, обращается к менеджеру памяти. Он инициализирует копирование страницы с диска в страницу физической памяти. Аппаратное прерывание от диска информирует о том, что загрузка нужной страницы завершена. Повышение приоритета процесса заблокировано кроме как в процессе прерывания от диска. Получив такое прерывание, приоритет процесса повышается до приоритета прерывания от диска.

В отличие от страницы, которая является единицей физического деления памяти, т.е. размер установлен в системе, сегмент является единицей логического деления памяти. Размер сегмента определяется размером программного кода.

В процессоре должен быть регистр базового адреса. Таблиц сегментов будет столько, сколько и процессов. В регистре сегментов будет обязательно поле, отражающее размер сегмента, при этом под размер сегмента в любой системе отводится определенное количество разрядов. В Intel под лимит отводится 20 бит.



Существует три типа организации таблиц сегментов:

A. Единая таблица

Каждый сегмент будет иметь глобальное имя.

Она содержит все дескрипторы, единственное имя сегмента является индексом его дескрипторов в таблице. Дескриптор сегмента должен содержать список прав доступа всех пользователей данного сегмента. Это довольно большой объем информации, т.к. процессов в памяти много.

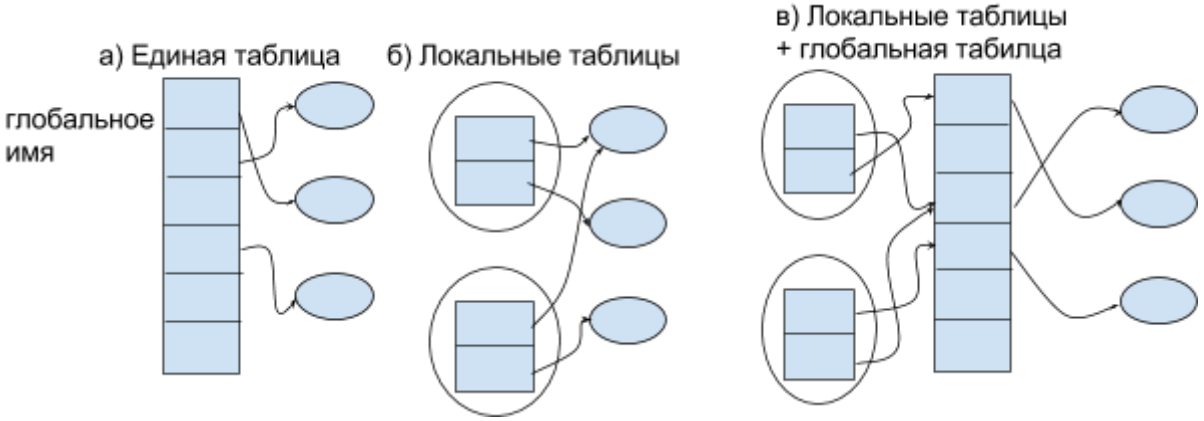
Такой тип организации неудобен при совместном использовании сегментов, т.к. все пользователи должны использовать одно единственное глобальное имя сегмента.

B. Локальные таблицы

Каждое адресное пространство процесса описывается собственной локальной таблицей сегментов. Доступ к локальному дескриптору осуществляется по имени. Отсюда, если сегмент доступен для нескольких процессов, или несколько процессов нуждаются в использовании одного и того же сегмента, то такой сегмент будет иметь несколько дескрипторов, и, соответственно, несколько локальных имен.

C. Локальные таблицы + глобальная таблица

Каждый сегмент имеет глобальный дескриптор, который содержит все характеристики физического размещения, а именно: размер, адрес и другие характеристики.



(на б и в видно коллективное использование). В Intel используется в.

Локальный дескриптор содержит информацию, описывающую сегмент виртуального адресного пространства, т.е. локальной среды, и имеет указатель на глобальный дескриптор. Глобальные дескрипторы соответствуют права доступа к глобальному сегменту.

2. Демоны в Unix. Правила программирования демонов. Примеры. Доп. вопросы: просила показать самого главного первого демона и рассказать о демонах для работы с soft irq.

Демоны - это долгоживущие процессы. Зачастую они запускаются во время загрузки системы и завершают работу вместе с ней. Т.к. они не имеют управляющего терминала, говорят, что они работают в фоновом режиме. В системе UNIX демоны решают множество повседневных задач.

Давайте рассмотрим некоторые наиболее распространенные системные демоны и их отношения с группами процессов, управляющими терминалами и сессиями.

ps -axj:

PPID	PID	PGID	SID	TTY	TPGID	UID	COMMAND

0	1	0	0	?	-1	0	init
1	2	1	1	?	-1	0	[keventd]
1	3	1	1	?	-1	0	[kapmd]
0	5	1	1	?	-1	0	[kswapd]
0	6	1	1	?	-1	0	[bdflush]
0	7	1	1	?	-1	0	[kupdated]
1	1009	1009	1009	?	-1	32	portmap
1	1048	1048	1048	?	-1	0	syslogd -m 0
1	1335	1335	11335	?	-1	0	xinetd -pidfile /var/run/xinetd.pid
1	1403	1	1	?	-1	0	[nfsd]
1	1405	1	1	?	-1	0	[lockd]
1405	1406	1	1	?	-1	0	[rpciod]
1	1853	1853	1853	?	-1	0	crond
1	2182	2182	2182	?	-1	0	/usr/sbin/cupsd

Процесс с идентификатором 1 - это обычно процесс init. Это системный демон, который, кроме всего прочего, отвечает за запуск различных системных служб на различных уровнях загрузки. Как правило, эти службы также реализованы в виде демонов.

Правила программирования демонов

При программировании демонов во избежание нежелательных взаимодействий следует придерживаться определенных правил. Сначала мы перечислим эти правила, а затем продемонстрируем функцию daemonize, которая их реализует.

1. Прежде всего нужно вызвать функцию umask, чтобы сбросить маску режима создания файлов в значение 0. Маска режима создания файлов, наследуемая от запускающего процесса, может маскировать некоторые биты прав доступа. Если предполагается, что процесс-демон будет создавать файлы, может потребоваться установка определенных битов прав доступа. Например, если демон создает файлы с правом на чтение и на запись для группы, маска режима создания файла, которая выключает любой из этих битов, воспрепятствовала бы этому.
2. Вызвать функцию fork и завершить родительский процесс. Для чего это делается? Во-первых, если демон был запущен как обычная команда оболочки, то завершив родительский процесс, мы заставим командную оболочку думать, что команда была выполнена. Во-вторых, дочерний процесс наследует идентификатор группы процессов от родителя, но получает свой идентификатор процесса; таким образом мы гарантируем, что дочерний

процесс не будет являться лидером группы, а это необходимое условие для вызова функции setsid, который будет произведен далее.

3. Создать новую сессию, обратившись к функции setsid. При этом процесс становится (а) лидером новой сессии, (б) лидером новой группы процессов и (в) лишается управляющего терминала.
4. Сделать корневой каталог текущим рабочим каталогом. Текущий рабочий каталог, унаследованный от родительского процесса, может находиться на смонтированной файловой системе. Поскольку демон, как правило, существует все время, пока система не будет перезагружена, то в подобной ситуации, когда рабочий каталог демона находится в смонтированной файловой системе, ее невозможно будет отмонтировать. Как вариант,' некоторые демоны могут устанавливать собственный текущий рабочий каталог, в котором они производят все необходимые действия. Например, демоны печати в качестве текущего рабочего каталога часто выбирают буферный каталог, куда помещаются задания для печати.
5. Закрыть все ненужные файловые дескрипторы. Это предотвращает удержание в открытом состоянии некоторых дескрипторов, унаследованных от родительского процесса (командной оболочки или другого процесса). С помощью нашей функции open_max или с помощью функции getrlimit можно определить максимально возможный номер дескриптора и закрыть все дескрипторы вплоть до этого номера.
6. Некоторые демоны открывают файловые дескрипторы с номерами 0,1 и 2 на устройстве /dev/null - таким образом, любые библиотечные функции, которые пытаются читать со стандартного устройства ввода или писать на стандартное устройство вывода или сообщений об ошибках, не будут оказывать никакого влияния. Поскольку демон не связан ни с одним терминальным устройством, он не сможет взаимодействовать с пользователем в интерактивном режиме. Даже если демон изначально был запущен в рамках интерактивной сессии, он все равно переходит в фоновый режим, и начальная сессия может завершиться без воздействия на процесс-демон. С этого же терминала в систему могут входить другие пользователи, и демон не должен выводить какую-либо информацию на терминал, да и пользователи не ждут того, что их ввод с терминала будет прочитан демоном.

Пример

```
#include "apue.h"
#include <syslog.h>
#include <fcntl.h>
#include <sys/resource.h>
void daemonize(const char *cmd)
{
    int i, fd0, fd1, fd2;
    pid_t pid;
    struct rlimit rl;
    struct sigaction sa;

    /* Clear file creation mask. */
```

```
        umask(0);
        /* Get maximum number of file descriptors. */
        if (getrlimit(RLIMIT_NOFILE, &rl) < 0)
                err_quit("%s: can't get file limit", cmd);
        /*  Become a session leader to lose controlling TTY. */
        if ((pid = fork()) < 0)
                err_quit("%s: can't fork", cmd);
        else if (pid != 0) /* parent */
                exit(0);
        setsid();
        /* Ensure future opens won't allocate controlling TTYs. */
        sa.sa_handler = SIG_IGN;
        sigemptyset(&sa.sa_mask);
        sa.sa_flags = 0;
        if (sigaction(SIGHUP, &sa, NULL) < 0)
                err_quit("%s: can't ignore SIGHUP", cmd);
        if ((pid = fork()) < 0)
                err_quit("%s: can't fork", cmd);
        else if (pid != 0) /* parent */
                exit(0);
        /* Change the current working directory to the root so we won't prevent
         * file systems from being unmounted.
         */
        if (chdir("/") < 0)
                err_quit("%s: can't change directory to /", cmd);
        /* Close all open file descriptors. */
        if (rl.rlim_max == RLIM_INFINITY)
                rl.rlim_max = 1024;
        for (i = 0; i < rl.rlim_max; i++)
                close(i);
        /* Attach file descriptors 0, 1, and 2 to /dev/null.  */
        fd0 = open("/dev/null", O_RDWR);
        fd1 = dup(0);
        fd2 = dup(0);
        /* Initialize the log file. */
        openlog(cmd, LOG_CONS, LOG_DAEMON);
        if (fd0 != 0 || fd1 != 1 || fd2 != 2) {
                syslog(LOG_ERR, "unexpected file descriptors %d %d %d",
                  fd0, fd1, fd2);
                exit(1);
        }
}
```

Демон softirq

Это поток ядра каждого процессора. Когда компьютер нагружен мягкими прерываниями (причем они обслуживаются при возвращении из аппаратного прерывания, но возможно, что мягкие прерывания переключаются значительно

быстрее, чем они могут быть обслужены) (это происходит в случае, когда прерывания приходят очень быстро, одно за другим, и ОС не успевает закончить обслуживание одного до прихода другого прерывания: например, сетевая карта с высокой скоростью получает пакеты в течение короткого промежутка времени), ОС не может справиться с таким потоком прерываний – они должны быть обработаны, должна быть обработана нижняя часть прерываний – тогда создается очередь для их последовательной обработки с помощью специального процесса ядра – ksoftirqd. Если демон этот занимает слишком много процессорного времени, то это указывает на то, что компьютер указывает на то, что компьютер находится под большой нагрузкой прерываний.

Билет 18

1. Система прерываний: аппаратные прерывания. Адресация обработчиков прерываний в защищенном режиме. Диаграмма выполнения команды и проверка сигнала прерывания.

Основная функция аппаратных прерываний –информирование процессора о завершении операции ввода-вывода, что позволяет процессору на время отключиться от выполнения процесса (распараллеливание – основа архитектуры).

Аппаратные прерывания – асинхронные события, которые вытесняются независимо от того, какая работа выполняется процессором: это значит, что система не знает, в каком месте потока будет прерывание.

В результате прерывания в системе x86 через таблицу дескрипторов прерываний осуществляется вызов соответствующего обработчика прерываний Interrupt Service Routing(ISR).

Обработчик прерывания выполняется в режиме ядра в линейном контексте, т.к. прерванный процесс не имеет отношения к возникшему прерыванию, обработчик не должен обращаться к контексту процесса, поэтому он не обладает правом блокировки.

Хотя, прервыание все равно оказывает влияние на процесс – время на обработку прерывания является частью выделенного процессу кванта.

Например, обработчик прерывания системного таймера использует тики текущего процесса т поэтому нуждается в доступе к его структуре proc. Следует отметить, что контекст процесса неполностью защищен от аппаратных прерываний, неправильно написанный обработчик прерывания может нанести вред данным.

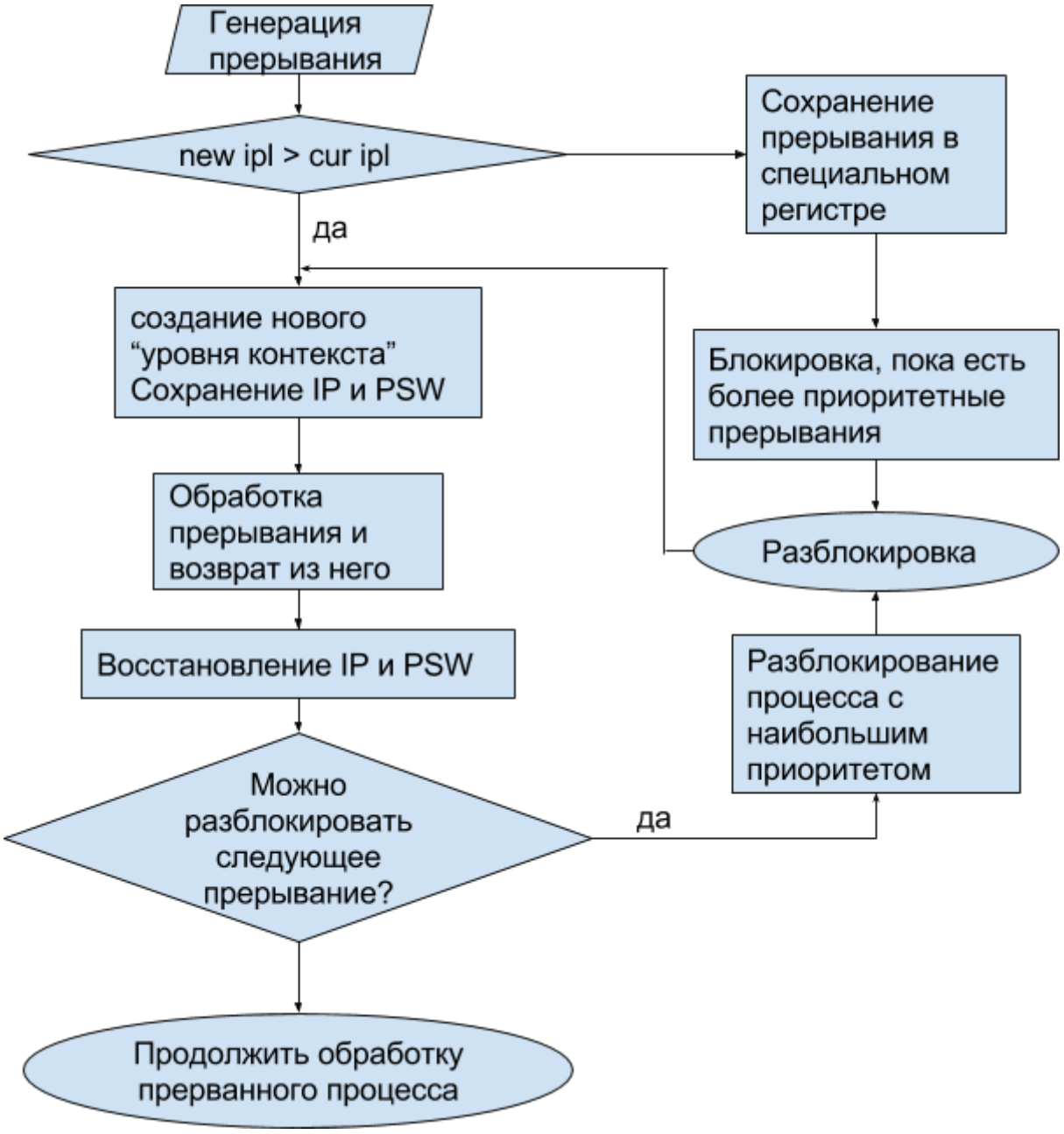
Прерывания могут возникнуть вследствие множества независимых событий в системе, поэтому реальна ситуация, когда во время обработки прерыания возникнет другое. При этом прерывание таймера должно выполнится сразу, например, раньше прерывания шин, потому вводится поддержка различных приоритетов (UNIX – Interrupt Priority Level)

Уровни IPL сравниваются и связываются на аппартаном уровне. Ядро может повысить IPL с целью блокировки на время выполнения критических инструкций.

На некоторых платформах поддерживается глобальный стек прерываний.

На платформе UNIX не поддерживается глобальный стек прерываний, используется аппаратный стек текущего процесса.

Должен быть обеспечен механизм изоляции от обработчика, для этого ядро помещает в стек уровень контекста перед вызовом обработчика. Уровень контекста содержит информацию для восстановления кода, который был прерван.



// Я бы не стал эту схему рисовать, потому что не помню такого в лекциях и Рязанова вроде как трехает за схемы не по госту

Адресация прерываний в защищенном режиме

На ножку IRQ0 поступает прерывание от системного таймера, на IRQ1 подключается клавиатура (номер IRQ сохраняется в современных системах)Если сигнал не замаскирован, ведущий контроллер прерываний формирует сигнал INT, который поступает на выделенную ножку процессора. Процессор формирует ответный сигнал INTA и контроллер формирует вектор прерывания, который используется для адресации обработчика прерывания. Обработчик прерывания адресуется через IDT в защищенном режиме. Вектор прерывания используется, как смещение.Прерывание формируется контроллером прерывания по завершении операций ввода-вывода.

2. Специальные файлы устройств, каталог /dev, старший и младший номера устройств. Структура usb_driver: функции probe() и disconnect(). HID драйверы. Обработчики аппаратных прерываний: регистрация. Верхняя и нижняя половины обработчиков прерываний.

ФС /dev

Эта ФС содержит информацию об устройствах (в том числе и виртуальных).

Если перейти в каталог dev и набрать ls -l, то мы увидим следующую информацию:

crw-rw-rw-	1	root root	1, 3	июн 18 09:28	null
crw-------	1	root root	10, 1	тут	psavx
	1	root root	4, 1	инфа	tty1
	1	root tty	4, 64	о	ttys0
	1	root uuep	4, 65	дате	ttys1
	1	vcsa tty	7, 1	но этот	vcs1
	1	vcsa tty	7, 129	русский писать	vlsa1
	1	root root	1, 5	не надо	zero

major/minor

Два числа – major и minor. Несколько устройств могут разделять один и тот же старший номер. Это достижение новых версий ядра.

Различаются два типа устройств – символьные и блочные. Некоторые старшие номера зарезервированы за драйвером устройства. Другие старшие номера присваиваются динамически: например: 94 — direct access storage device, 4 – посл. Ин-сы, 13 – мыши, 14 – аудиоустройства. Младшие номера используются системой для различия конкретных физических/логических устройств. Посмотреть устройства в системе можно в файле /proc/devices:

Символьные устройства:

1 mem
2 pty
3 ttyp
4 ttys
6 lp
7 vcs
10 misc
13 input
14 sound
21 sg
180 usb

Блочные устройства:

2 fd

8 sd
11 sr
65 sd

Перечень устройств можно найти в (usr/src/linux/)documentation/devices.txt.

/dev/hd?

Hd.....ide device

/dev/hda(3)

first drive or primary master

/dev/hdb(3)

second or primary slave

/dev/hdc(22)

third device or secondary master

/dev/hdd(22)

....

unsigned int major(dev_t dev);

unsigned int minor(dev_t dev);

Эти макросы не входят в POSIX1. Эти макросы дают нам возможность получить старшую и младшую части идентификатора устройства.

Аппаратные прерывание – это асинхронный естественный параллелизм, событие, которое происходит вне зависимости от любой деятельности процессора. Uninterruptible sleep у процессов возникает при прерываниях от устройств (при запросе i/o).

Является абсолютно архитектурно зависимым действием обработка прерываний. Например, в MS-DOS контроллером PIC адрес обработчика прерывания, т. н. вектор прерываний размером 4 байта, находился в таблице векторов прерываний, которая начиналась с нулевого адреса. При прерывании контроллером формировался вектор в виде суммы … в этой схеме был ведущий и ведомый контроллер, и у ведомого контроллера был номер 8. Отсюда прерывание от таймера именно по этому называется int 8h. Вектор прерывания используется для получения адреса обработчика прерывания из … В 32бит системах контроллер прерывания APIC имеет значительно большее количество линий прерываний. Для доступа к обработчику прерывания используется IDT. Вектор прерывания, формирующийся контроллером, используется как смещение к дескриптору. В любом случае, в любой архитектуре, аппаратные прерывания прерывают текущую деятельность на одном из процессоров и вызывают выполнение соответствующей процедуры обработки прерывания (Interrupt Service Routine).

В Linux процедура обработки для заданного конкретного прерывания может быть зарегистрирована с помощью функции request_irq.

<linux/interrupt.h>

typedef irqreturn_t (*irq_handler_t)(int void *);

int request_irq(unsigned irq, irq_handler_t handler, unsigned long flags,
const char *name, void *dev);

extern void free_irq(unsigned int irq, *);

Каждое устройство имеет один драйвер, и использующие прерывания драйвера регистрируют один обработчик прерывания. Драйверы могут зарегистрировать

обработчик и разрешать определенную линию прерывания посредством функции request_irq.

Параметры:

1. irq – определяет номер irq устройства. Для некоторых устройств, такие как Legacy PC device (систаймер = 0, клавиатура, мышь PS/2) устанавливаются аппаратно. Для большинства иных устройств определяются программно и динамически.
2. Handler – указатель на функцию обработчика прерывания. Функция вызывается, когда процессор получает сигнал прерывания. Прототип обработчика имеет вид: static irqreturn_t intr_handler(int irq, void *dev, struct pt_reqs *reqs);
3. flags – битовая маска следующих флагов:
   IRQ_SHARED
   IRQ_PROBE_SHARED
   _IRQF_TIMER
   IRQF_PERCPU
   IRQF_NOBALANCING
   IRQF_IRQPOLL
   IRQF_ONESHOT
   IRQF_NO_SUSPEND
   IRQF_FORCE_RESUME
   IRQF_NO_THREAD
   IRQF_EARLY_RESUME
   IRQF_COND_SUSPEND
   IRQF_TIMER _IRQF_TIMER
   IRQF_NOSUSPEND
   IRQF_NOTHREAD

Как только видим в литературе SA_ – это устаревшая информация.

Прерывания в системе делятся на быстрые и медленные. Установка флага SA_INTERRUPT указывала, что данный обработчик является обработчком быстрого прерывания. В системах таким быстрым прерыванием, способным быстро завершиться, является прерывание только от системного таймера, поэтому флаг SA_INTERRUPT заменен на IRQ_TIMER.

Медленные прерывания (обработчики прерываний) делятся на две части: верхнюю и нижнюю (top и bottom). Верхняя выполняется как быстрое прерывание, а нижняя ставится верхней в очередь на выполнение и выполняется уже при разрешенных прерываниях через какое-то время.

Линия прерывания может разделяться несколькими обработчиками прерываний. Это избавляет разработчика от необходимости написания драйвера, т. е. Функциональность внешнего устройства может определяться своим обработчиком прерывания. Например, повесить на кнопки клавиатуры звуки.

SA_SAMPLE_RANDOM – задействуется пул энтропии ядра. Он производит реально случайные числа. Устаревший флаг.

4. const char *name – ASCII-код, может представлять устройство, связанное с прерыванием, например, KEYBOARD. Это имя используется в /proc/irq, /proc/interrupts.

5. void *dev: используется прежде всего для разделения линий прерывания. Обеспечивает уникальность формируемых файлов и это позволяет выполнить удаление только нужного обработчика прерывания с линией прерывания. …

Вернемся к прототипу обработчика. interrupt_handler имеет 3 параметра: irq, dev, reqs. Irq – номер прерывания, dev – тот же параметр, что и в предыдущей функции для различия устройств, исп. Один и тот же драйвер, reqs содержит указатель на структуру, содержащей регистры процессора, которая хранит контекст процессора перед вызовом прерывания.

irqreturn_t определено в <linux/irqreturn.h>. Функция может вернуть три значения:

#define irq_none (0) – когда обнаружено, что прерывание не иницинировано. В результате прерывание должно быть передано другим обработчикам, зарегистрированным на данной линии irq.

#define irq_handled(1) – значение возвращается, когда обработчик был корректно вызван и его устройство действительно сформировало прерывание и оно успешно обработано.

#define irq_retval(x) – если x не равен 0, то макрос возвращает irq_handled. В противном случае макрос возвращает irq_none.

```
Static irqreturn_t intr_handler (int irq,void *dev)
{
        if (!/*проверка запроса прерывания*/)
                return IRQ_NONE;
        /*код обслуживания устройства*/
        return IRQ_HANDLED;
}
```

Получение данных от внешнего устройства будет как в случае read(), так и в случае write(). Грубо говоря, обработчик должен выполнить минимальный объем действий, связанный с передачей данных запросившему эти действия процессу.

Обработчики аппаратных прерываний .. на локальном процессоре и текущая линия прерывания глобально запрещена на остальных (?)… речь идет о SMT-архитектурах.

В любой системе аппаратные прерывания большую часть выполняются в два этапа:

1. tophalf
2. bottomhalf (в Windows: DPC).

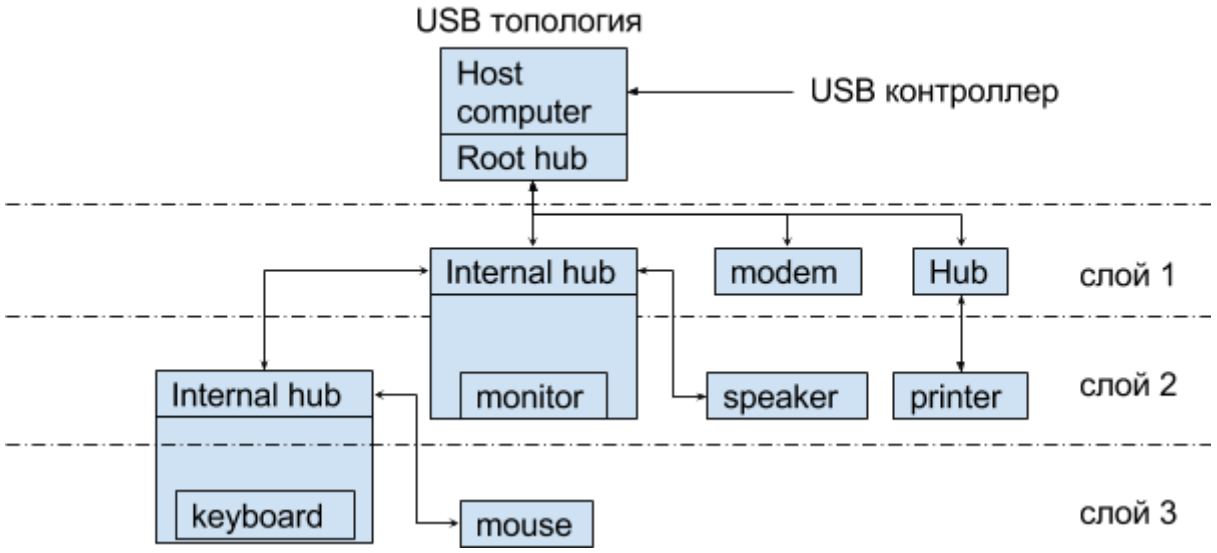
Softirq – мягкие, гибкие прерывания.

Tasklet

Очереди работ (workqueue)

USB-драйверы

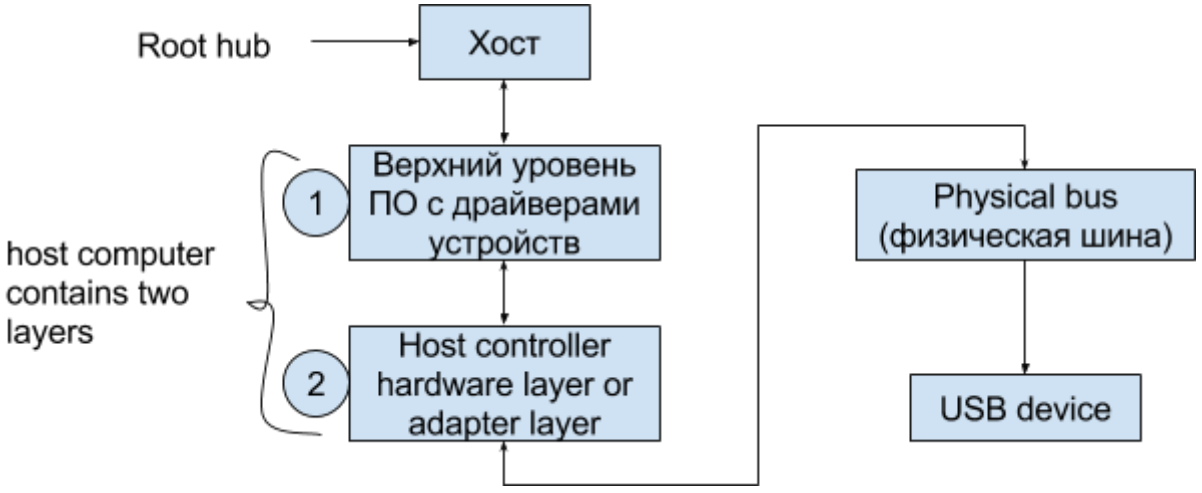
USB – Universal Serial Bus – универсальная последовательная шина – главным там является хост. Вся эта структура – иерархическая.



Host computer – корневой узел – содержит неявные (implicit) узлы – хабы.

Хаб распространяет сигналы (электрические) к одному или более портам посредством увеличения общего количества функций на шине. Хаб имеет одно соединение – upstream port – соединение к верхнему уровню дерева и некоторое количество портов для подключения внешних устройств или других хабов. Хабы – активные электронные устройства. На рисунке показана связь мыши со встроенным в клавиатуру хабом, т. е. некоторое устройство имеет встроенные хабы.

Архитектура USB-драйвера.



Host-computer – программно-управляемая схема, интегрированная в схему южного моста компьютера. Его назначение – контроль над данными, передаваемые USB-устройствам(и). Имеет два уровня. Ниже них идет физическая шина (обеспечивает соединения контроллера и периферии) и на самом низком уровне – USB device, использующее спецификацию USB Electrical и спецификацию форматов данных для связи с host computer. USB-кабель состоит из четырех медных проводников – 2 питания — 2 витая пара – и заземленная оплетка.

На USB Specification определено 4 вида передачи данных:

1) Control
2) Interrupt
3) Bulk – дает гарантии целостности данных, но не скорости и задержки

отправки данных.

4) Isochronous

На логическом уровне USB поддерживает транзакции приема и передачи данных. Каждый пакет каждой транзакции содержит endpoint на устройстве. При подключении устройства драйверы ядра ОС читают с устройства список конечных точек и создают управляющие структуры для общения с каждой точкой взаимодействия. Совокупность конечной точки и структуры данных называется каналом (pipe).

```
struct usb_driver //одна из основных структур с точками входа
{
        const char* name;
        int (*probe)(struct usb_interface *intf, const struct usb_device_id *id);
        void (*disconnect)(struct usb_interface *intf);
        ...
}
```

Драйвер USB пишутся для интерфейсов устройств, а не для устройства в целом.

Обратные вызовы probe и disconnect выполняются драйвером для каждого зарегистрированного устройства – первый параметр – соотв. ему интерфейс.

Для получения дескриптора устройства можно воспользоваться макросом:

```
struct usb_device device=interface_to_usbdev(interface);
static struct usb_driver my_usb_dr=
{
        .name = «usb_my»;
        .probe = my_usb_probe;
        .disconnect = my_usb_disconnect;
        .id_table = my_usb_table;                    // указывает таблицу устройств,
                                                     // работающих с этим драйвером
}
```

Таблица id_table описывается до этой структуры:

```
static struct usb_device_id my_usb_table[]=
{
        {USB_DEVICE(0x046d; 0x080)},
        {} // обязательный символ-терминатор
};
```

Такие драйверы реализуются в виде загружаемых модулей и есть определенные требования к написанию соответствующих функций

```
... my_usb_probe(...)
{
        struct my_usb_info *usb_info;
        struct usb_device *dev = interface_to_usbdev(intf)
        ...
}
... my_cleanup_module(...)
{
        usb_deregister(&my_usb_driver);
}
```

Кроме USB-драйверов есть класс **USB-HID** драйверов. Для них существуют специальные правила. Обстоятельство: согласно спецификации HID общение между компьютером и HID-устройством производится при помощи отчетов (report) – структура, которая хранит в себе данные, которые передаются устройством, описывается это все дескрипторами отчетов. Если мы знаем этот дескриптор, то, заменив отдельные поля отчета, мы можем поменять функциональность устройства.

Существуют много способов изменения функциональности внешних устройств: можно написать собственный обработчик прерывания в составе драйвера, просто обработчик прерывания, заставив линию прерывания работать в разделяемом режиме, можем поменять значения полей отчета – особенно радует обилие библиотек, работающих с внешними устройствами. В частности, есть библиотека usb.h, keyboard.h, cdef.h, platformdevice.h (находятся в linux/)

```
static struct usb_device_id usb_mouse_id_table[]=
{
        {USB_INTERFACE_INFO(USB_INTERFACE_CLASS_HID,
                            USB_INTERFACE_SUBCLASS_BOOT,
                            USB_INTERFACE_PROTOCOL_MOUSE)};
        {}
};
```

Особенности установки драйвера в системе

Когда драйвер написан и отлажен, его надо установить, но следует вначале выгрузить модуль USB_HID, который автоматически регистрирует все стандартные драйверы в системе – например, стандартный драйвер мыши. После его отключения можно установить собственный драйвер и загрузить обратно USB_HID для поддержки функциональности других HID-устройств.

Билет 19

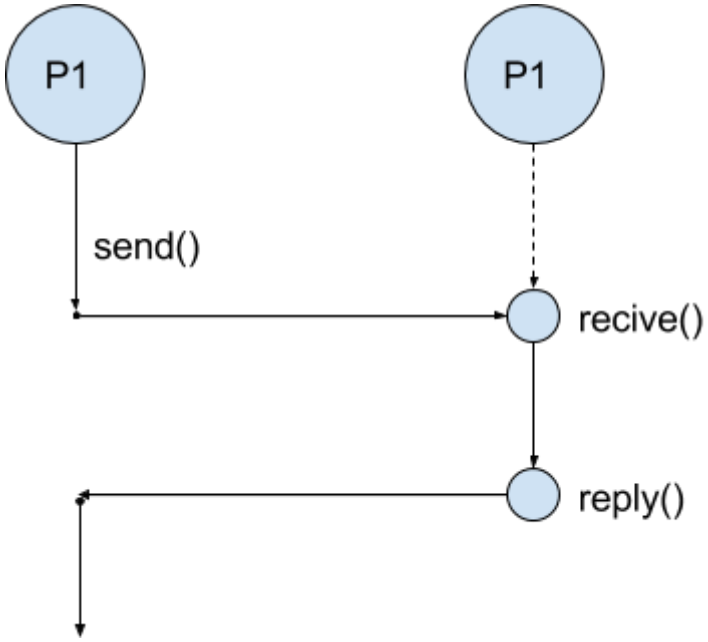
1. Передача данных между процессами:очереди сообщений, разделение памяти, программный канал

Передача сообщений

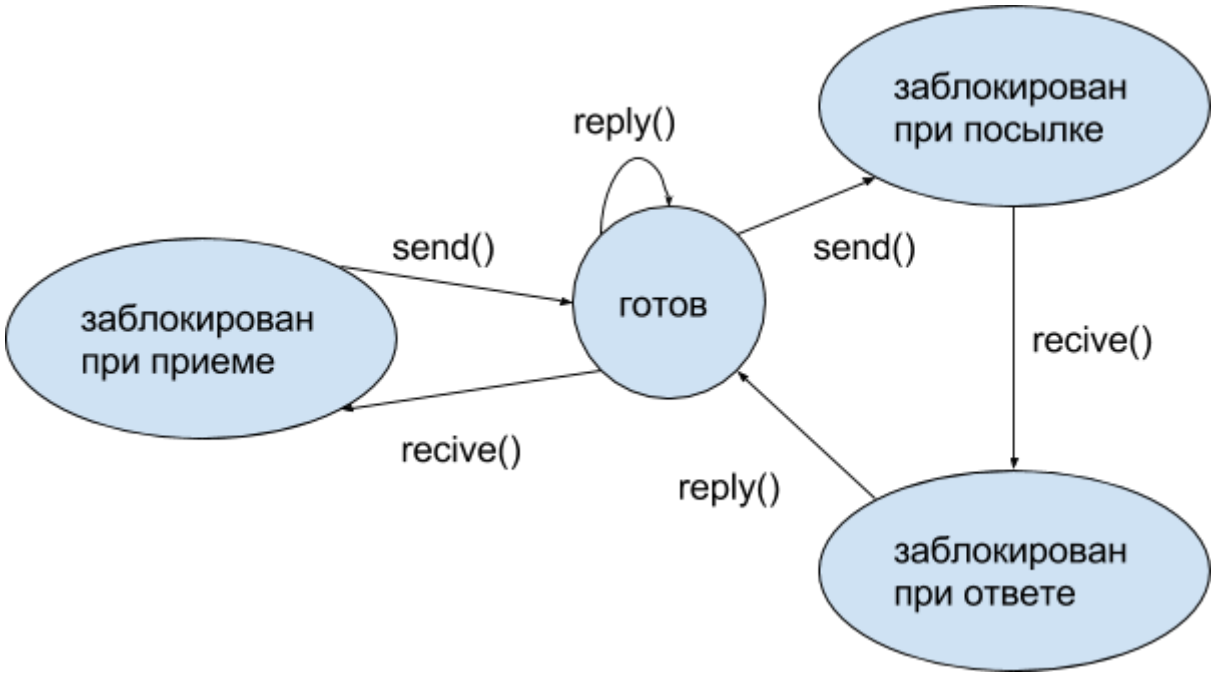
Есть два понятия: взаимоисключение и синхронизация, которые не стоит смешивать.

Для передачи используются команды send() и receive(), а также reply() для обработки и отсылки результата работы отправителю, но вообще используются только два первых примитива.

Диаграмма последовательностей событий:



Синхронизация процессов: передача сообщений может быть использована для синхронизации процессов, как только процесс вызвал send(), он блокируется до получения ответа, при этом при передачи процесса возможны три состояния блокировки процесса:



Синхронизация – такая связь между процессами, в которой один процесс достигает своей определенной точки, разблокируя другой процесс.

Программные каналы

В программе именованные pipes можно создавать:

Mknod(name, IFIFO|ACCESS, 0);

Типы каналов:

1. Именованный: важно указать тип FIFO. Можно создавать в программе
2. Неименованный: создается системным вызовом pipe. Не имеет id, но есть дескриптор.

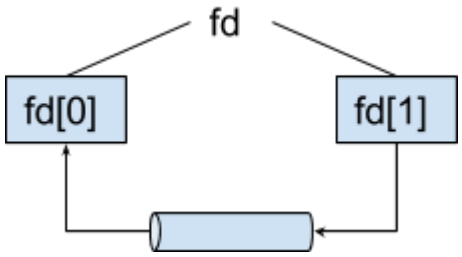
```
int fd[2]; // fd – file descriptor
pipe(fd);
```

inode (index node) – индексный узел.

Без имени. Имеет дескриптор (файловый).

Процесс-потомок наследует дескрипторы программных каналов, открытых предком.

```
if (child_pid == 0)
{
    ...
    close(fd[1]);
    read(fd[0], line, 100);
    ...
} else {
    close(fd[0]);
    write(fd[1], test, strlen(test));
    ...
}
```



Разделяемая память

В других средствах межпроцессового взаимодействия (IPC) обмен информацией между процессами проходит через ядро, что приводит к переключению контекста между процессом и ядром, т.е. к потерям производительности.

Техника разделяемой памяти позволяет осуществлять обмен информацией через общий для процессов сегмент памяти без использования системных вызовов ядра. Сегмент разделяемой памяти подключается в свободную часть виртуального адресного пространства процесса. Таким образом, два разных процесса могут иметь разные адреса одной и той же ячейки подключенной разделяемой памяти.

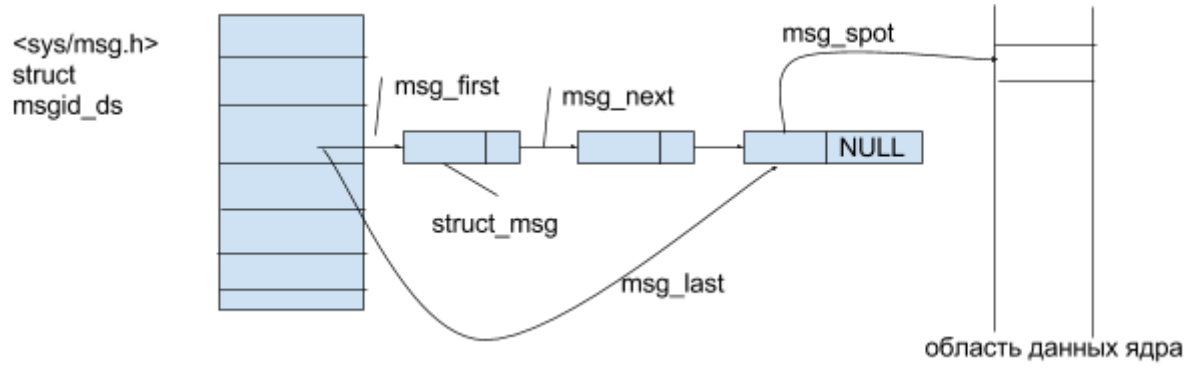
После создания разделяемого сегмента памяти любой из пользовательских процессов может подсоединить его к своему собственному виртуальному пространству и работать с ним, как с обычным сегментом памяти. Недостатком такого обмена информацией является отсутствие каких бы то ни было средств синхронизации, однако для преодоления этого недостатка можно использовать технику семафоров.

shmget — создание сегмента разделяемой памяти с привязкой к целочисленному идентификатору, либо анонимного сегмента разделяемой памяти (при указании вместо идентификатора значения IPC_PRIVATE);
shmctl — установка параметров сегмента памяти;
shmat — подключение сегмента к адресному пространству процесса;
shmdt — отключение сегмента от адресного пространства процесса.

```
#include <string.h>
#include <sys/stat.h>
#include <sys/ipc.h>
#include <sys/shm.h>
int main() {
    int perms = S_IRWXU | S_IRWXG | S_IRWXO;
    int fd = shmget(100, 1024, IPC_CREATE | perms);
    if (fd == -1) { perror(“shmget”); exit(1); }
    char *addr = (char *)shmat(fd, 0, 0);
    if (addr == (char *)(-1)) { perror(“shmat”); exit(1); }
    strcpy(addr, “Hello”);
    if (shmdt(addr) == -1) { perror(“shmdt”); }
    return 0;
}
```

Системные ограничения:
SHMMNI – число разделяемых областей
SHMMIN – минимальный размер РП в байтах
SHMMAX – максимальный размер РП в байтах

Очереди сообщений



При помещении сообщения в очередь создается новая запись и помещается в конец списка записей указанной очереди. В каждой записи: тип сообщения, число байт данных, указатель на сообщение. Когда процесс выбирает сообщение из очереди, ядро копирует его в адресное пространство процесса, после чего запись удаляется.

msgget(), msgctl(), msgsud(), msgrev()

Пример

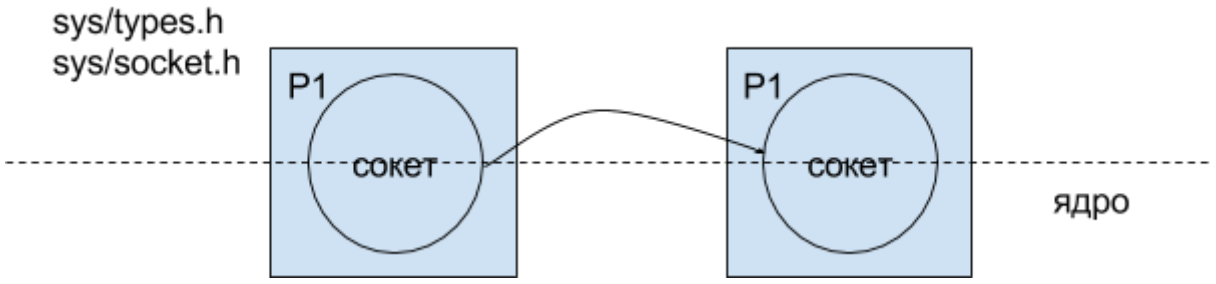
новая очередь сообщений stomp и устанавливает следующие права: rw-r---w-. Если msgget() успешен, то сообщение hello, иначе -- 15 (???); этот вызов неблокирующий.

```
#include <string.h>
#include <syslipc.h>
#include <sys/msg.h>
#define MSGMAX 1024
struct mbuf
{
   long mtype;
   char mtext[MSGMAX];
} mobj={15,"hello"};
int main()
{
   int fd = msgget(100,IPC_CREATE|IPC_EXCL}
   if (fd ==-1 || msgsnd(fd, &mobj, strlen(mobj.text)+1, IPC_NOWAIT))
      perror("message");
   return 0;
}
```

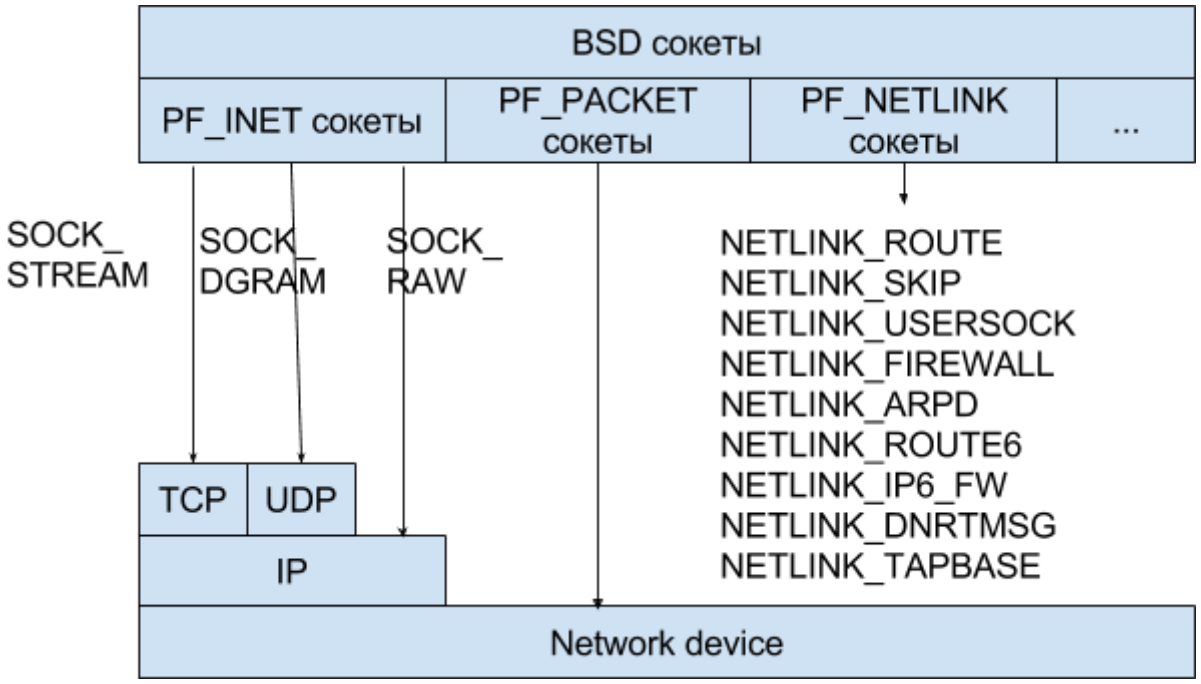
2. Мультиплексирование, сокеты, соединение клиент-сервер

Сокет – абстракция конечной точки взаимодействия. В результате приложение может использовать единобразный интерфейс сокетов для отправки и получения данных как по сети, так и на локальной машине. Нужные протоколы определяются на основе трех параметров:

1) family
2) type
3) protocol



int socket(int family, int type, int protocol);



PF_NETLINK созданы для общения между kernel space и user space (?)

Система предоставляет для работы с сокетами один-единственный сисвызов, который можно посмотреть в <net/socket.c> – посмотрим коротко структуру этого сисвызова.

```
Asmlinkage long sys_socketcall(int call, unsigned long *args)
{
        int err;
        if copy_from_user(a, args, nargs[call])
        return -EFAULT;
        switch(call)
        {
                case SYS_SOCKET: err = sys_socket(…); break;
                case SYS_BIND: err=sys_bind(…); break;
                case SYS_CONNECT: err = sys_connect(…); break;
        }
        return err;
}
```

Функция реализована как переключатель системных вызовов: параметр call определяет номер нужной функции. Все функции входят в сетевой протокол, определенный на сокетах.

```
#define SYS_SOCKET 1                    /* sys_socket(2) */
#define SYS_BIND 2                      /* sys_bind(2) */
#define SYS_CONNECT 3                   /* sys_connect(2) */
#define SYS_LISTEN 4                    /* sys_listen(2) */
#define SYS_ACCEPT 5                    /* sys_accept(2) */
#define SYS_GETSOCKNAME 6  /* sys_getsockname(2) */
#define SYS_GETPEERNAME 7  /* sys_getpeer name(2) */
#define SYS_SOCKETPAIR 8                /* sys_socketpair(2) */
#define SYS_SEND 9                      /* sys_send(2) */
#define SYS_RECV 10                     /* sys_recv(2) */
#define SYS_SENDTO 11                   /* sys_sendto(2) */
#define SYS_RECVFROM 12                 /* sys_recvfrom(2) */
#define SYS_SHUTDOWN 13                 /* sys_shutdown(2) */
#define SYS_SETSOCKOPT 14               /* sys_setsockopt(2) */
#define SYS_GETSOCKOPT 15               /* sys_getsockopt(2) */
#define SYS_SENDMSG 16                  /* sys_sendmsg(2) */
#define SYS_RECVMSG 17                  /* sys_recvmsg(2) */
#define SYS_ACCEPT4 18                  /* sys_accept4(2) */
struct socket {
        socket_stat e state;
        unsigned long flags;
        const struct proto_ops * ops;
        struct fasync_* list; // список асинхронного запуска
        struct file * file;
        struct sock * sk;
        wait_queue_head_t wait;
        short type;
};
```

- state – определяет 5 состояний сокета: typedef enum {
    - SS_FREE = 0, /* not allocated */
    - SS_UNCONNECTED , /* unconnected to any socket */
    - SS_CONNECTING , /* in process of connecting */
    - SS_CONNECTED , /* connected to socket */
    - SS_DISCONNECTING /* in process of disconnecting */
    - } socket_state;
- ops – ссылается на действие подключенного протокола (UDP/TCP)
- file – сылается на inode
- type – для хранения второго параметра функции socket(family,type,protocol). Допустимые значения:
    - SOCK_STREAM надёжная потокоориентированная служба (TCP) (сервис) или потоковый сокет
    - SOCK_DGRAM служба датаграмм (UDP) или датаграммный сокет
    - SOCK_SEQPACKET надёжная служба последовательных пакетов
    - SOCK_RAW Сырой сокет — сырой протокол поверх сетевого уровня.

Адреса сокетов

```
struct sockaddr {
```

```
        unsigned short sa_family ; // address family, AF_xxx
        char sa_data [14]; // 14 bytes of protocol address
};
```

При создании коммуникационных отношений нужно указать адрес конечной точки партнера взаимодействия.

```
// IPv4 AF_INET sockets:
struct sockaddr_in {
        short sin_family ; // e.g. AF_INET, AF_INET6
        unsigned short sin_port ; // номер порта e.g. htons(3490)
        struct in_addr sin_addr ; // определяет адрес интернета
        char sin_zero [8]; // zero this if you want to
};
struct in _addr {
        unsigned long s_addr ; // load with inet_pton()
};
```

- Адреса и номера портов должны быть указаны в сетевом порядке данных.
- Сокеты необходимо связать с адресом в выбранном
- Идентификация состоит из двух частей:
    - сетевой узел с номером сетевого адреса
    - конкретного процесса с номером службы

При организации взаимодействия на одной машине не нужно задумываться о порядке следования байтов – характеристика аппаратной платформы и определяет порядок байтов в данных типах данных.

Два порядка: big-endian (обратный, network) и little-endian(прямой, host, остроконечный)

Для Intel справедлив прямой порядок данных, для SPARC и PowerPC – обратный.

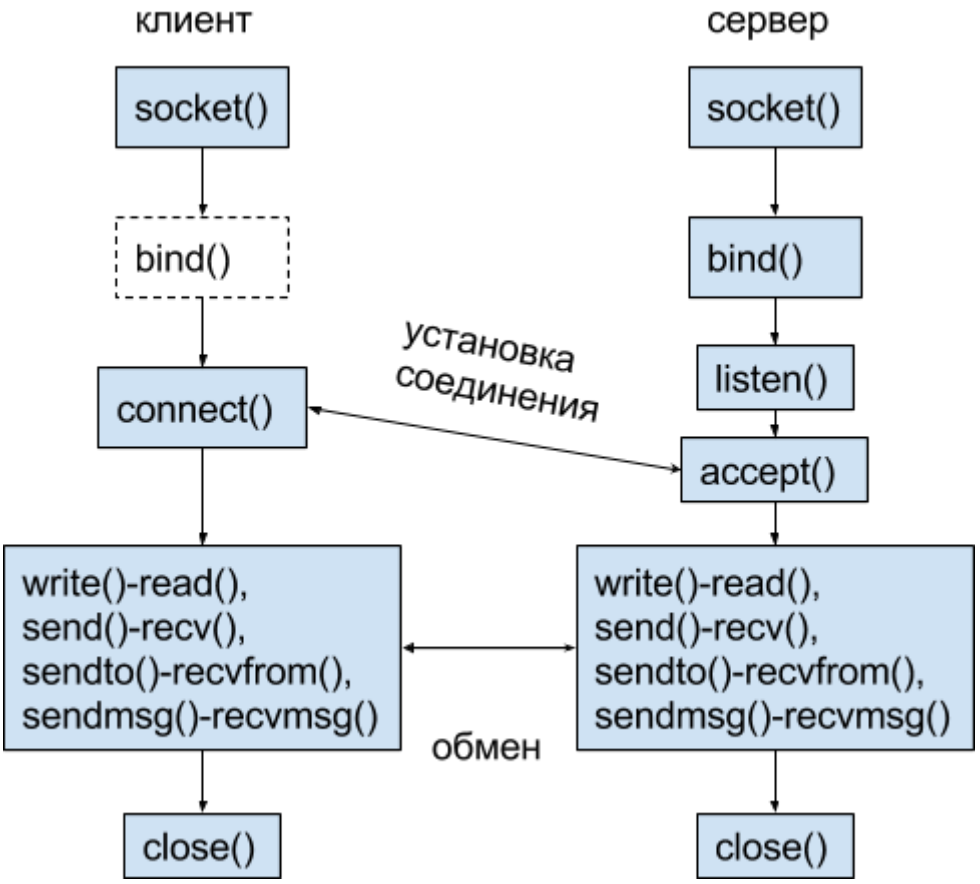
Преобразование из одного порядка в другой происходит с помощью 2 функций: htons (Host To Network Short) и htonl (...Long). Они определены в файле <arpa/inet.h>. Обратное преобразование делается соответствующими функциями ntohs и ntohl. На некоторых машинах все совпадает, но лучше использовать эти функции везде в целях переносимости. Все лишнее процессор уберет сам.

Операции сокетов.

Сокеты представляются как дескрипторы файлов. Они могут использоваться для операций чтения/записи, но установка коммуникационных отношений отличается от открытия файлов, поэтому нужны спец. Системные вызовы.

Взаимодействие через сокеты – это взаимодействие через клиент-сервер. Клиент активно инициирует установление контакта, а сервер сначала пассивно ждет активного запроса на установление контакта.

Стек вызовов:



- bind() используется для назначения локального адреса. Клиенты могут не вызывать bind(), так как их адрес назначается автоматически.
- Listen() выполняется сервером для информирования ОС о том, что в сокете надо принять соединение. Это имеет смысл для протоколов типа TCP, ориентированных на соединение.
- Connect() устанавливает соединение по переданному адресу. Для протоколов без установления соединения (UDP) может использоваться для указания адреса назначения для всех передаваемых пакетов.
- Accept() используется сервером для принятия соединения при условии, что он получит запрос на соединение, иначе вызов заблокирован, пока не получен запрос.
- Когда принимается соединение, сокет копируется, т. е. Первоначальный сокет в listen(), а новый сокет в состоянии connected.
- Вызовом accept() возвращается новый дескриптор файла для следующего сокета. Такое дублирование в ходе принятия дает серверу возможность принимать новые соединения, не закрывая предыдущие.

AF_UNIX – сокеты для межпроцессного взаимодействия на локальном компьютере
AF_INET – основанные на протоколе интернета версии 4
AF_INET6 – версия 6
AF_IPX – протокол IPX
AF_UNSPEC – неопределенный домен\

SOCK_STREAM – тип коммуникационного отношения – определяет на потоке надежное упорядоченное полнодуплексное логическое соединение между двумя сокетами

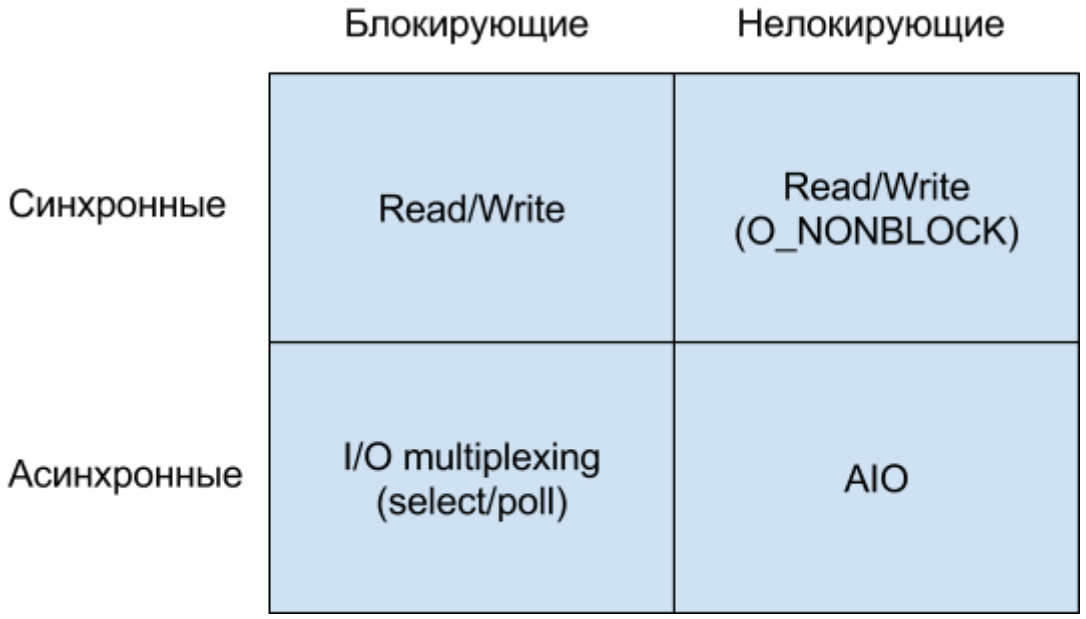
SOCK_DGRAM – определяют ненадежную службу без установления логического соединения, причем пакеты могут передаваться без сохранения порядка, так называемая широковещательная передача данных

SOCK_RAW – низкоуровневый интерфейс DGRAM по протоколу IP

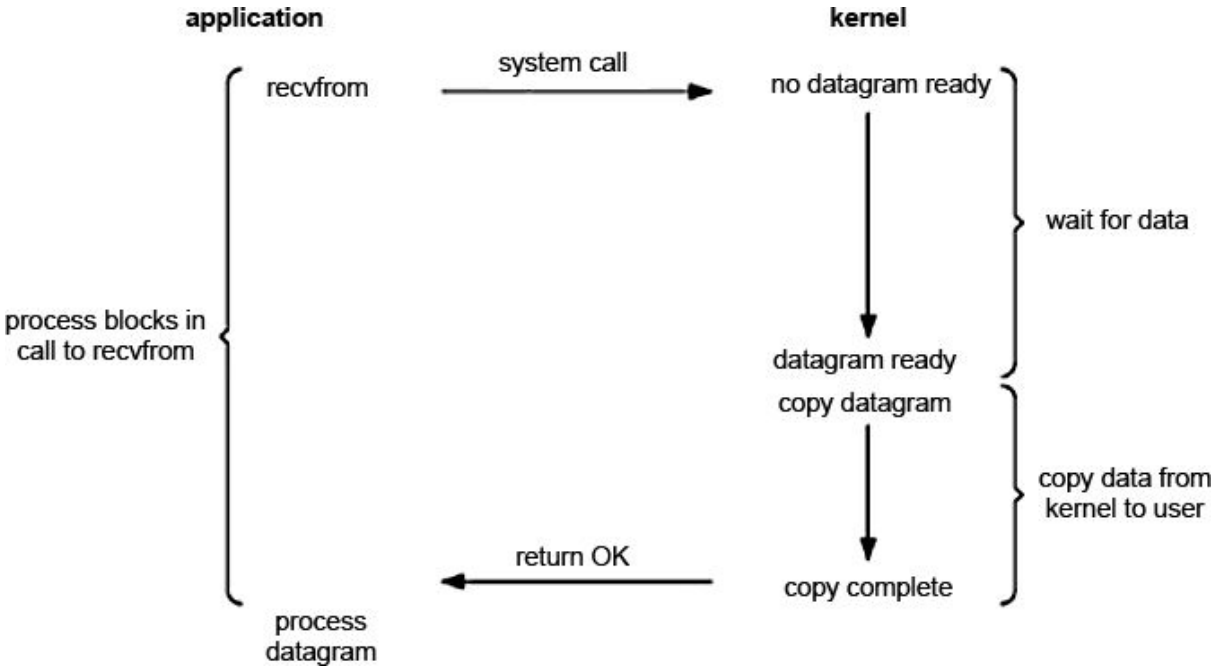
Третий аргумент – протокол – обычно этот аргумент равен 0 – выбор протокола будет выполнен по умолчанию (TCP/IP для …, SOCK_DGRAM – UDP)

5+1 моделей ввода/вывода

Все эти модели в самом общем виде могут быть представлены следующей диаграммой:



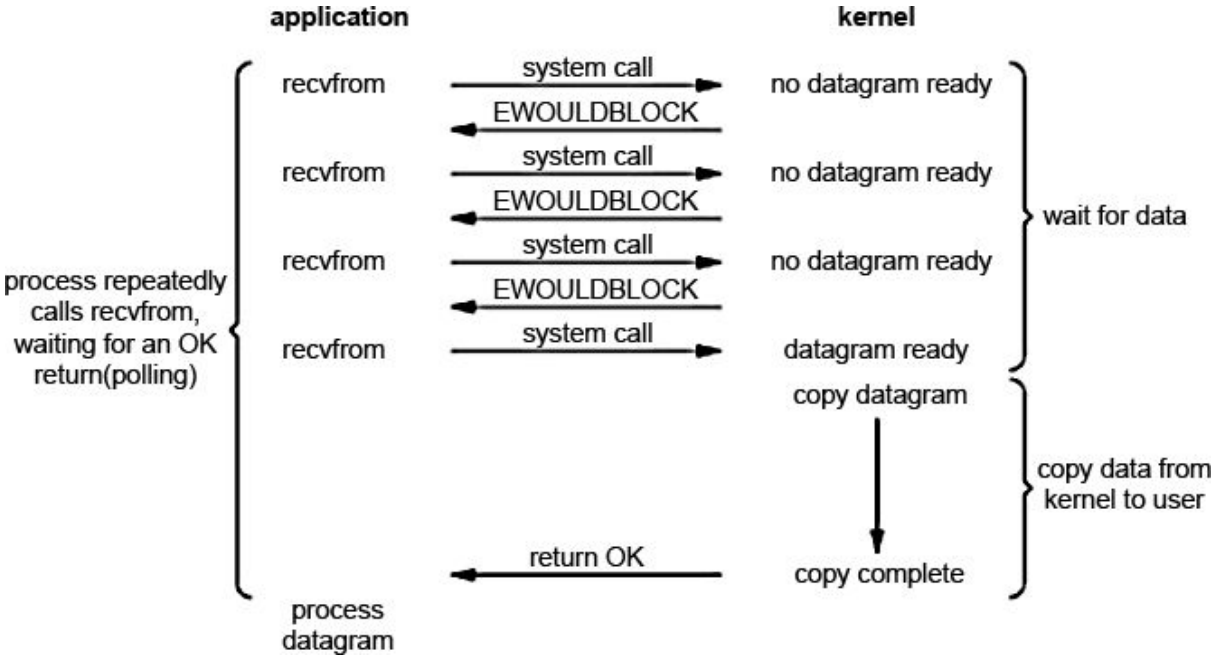
1) Блокирующий ввод вывод, по сути являющийся синхронным.



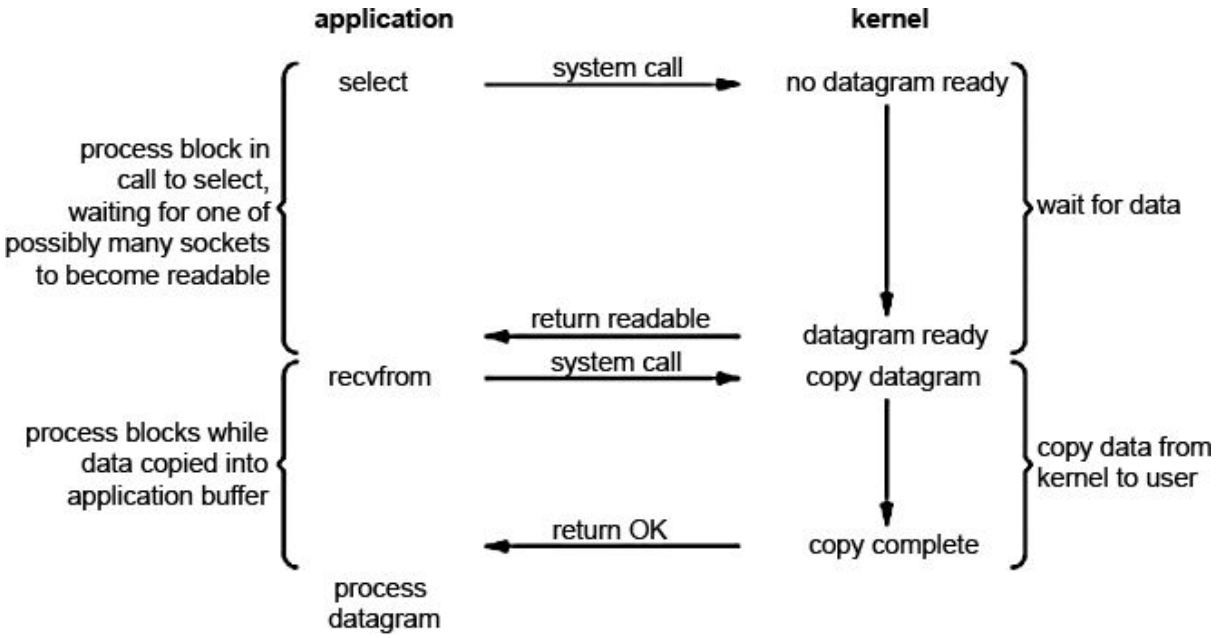
Понятно, что супервизор вызовет соответствующий драйвер устройства, он инициализирует процесс ввода/вывода, и при завершении IO это будет отмечено сигналом прерывания, …, будут вызваны callback-функции драйвера, и данные будут переданы приложению.

2) non-blocking IO (polling)

На этой основе можно создать цикл, который постоянно вызывает recvfrom (обращается за данными) для сокетов, открытых в неблокирующем режиме. Этот режим называется опросом (поллинг/polling) т.к. приложение все время опрашивает ядро системы на предмет наличия данных.

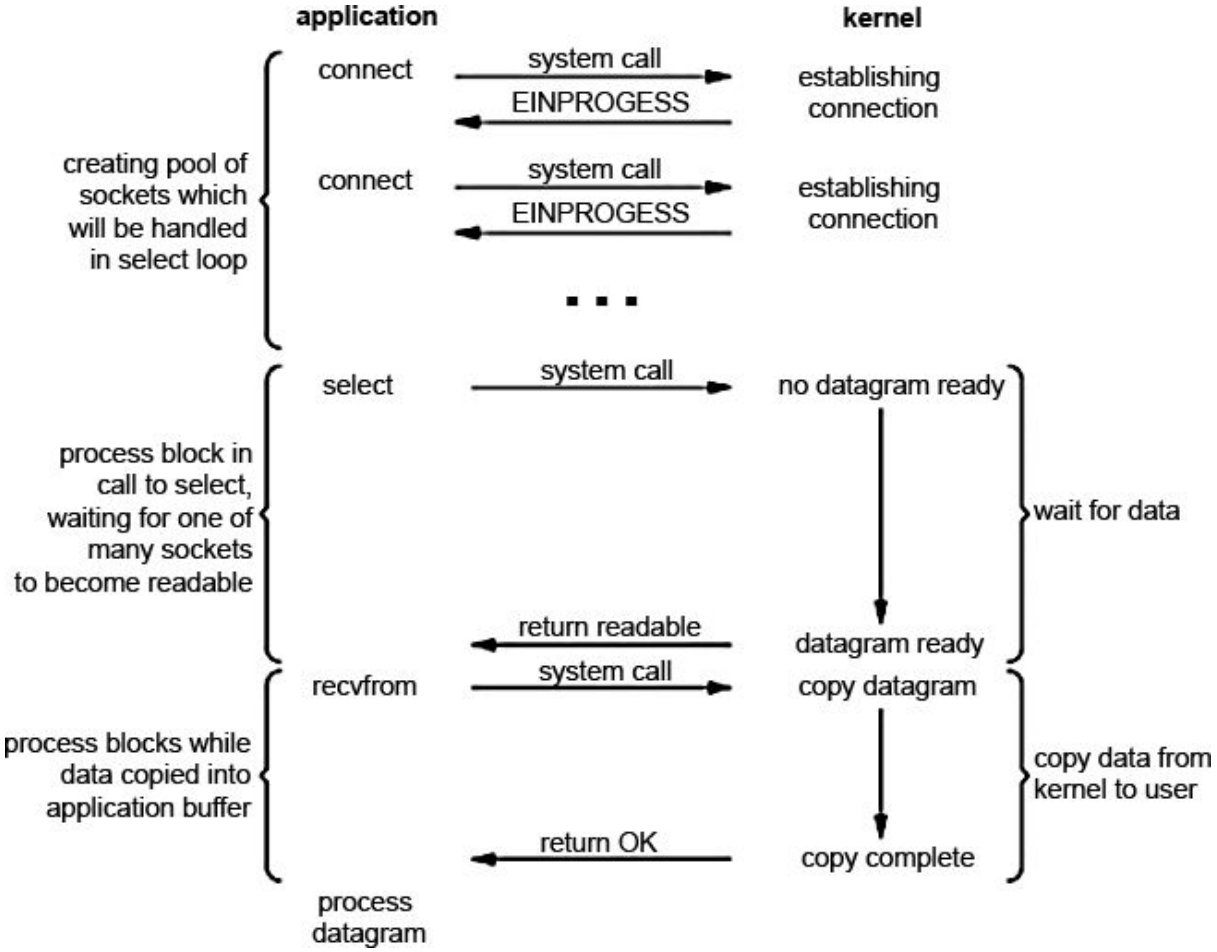


3) мультиплексирование (уплотнение) ввода/вывода



Приложение блокируется при вызове select'а ожидая когда сокет станет доступным для чтения. Затем ядро возвращает нам статус readable и можно получать данные помощью recvfrom. На первый взгляд — сплошное разочарование. Та же блокировка, ожидание, да и еще 2 системных вызова (select и recvfrom) — высокие накладные расходы. Но в отличии от блокирующего метода, select (и любой другой мультиплексор) позволяет ожидать данные не от одного, а от нескольких файловых дескрипторов. Надо сказать, что это наиболее разумный метод для обслуживания множества клиентов, особенно если ресурсы достаточно ограничены. Почему это так? Потому что мультиплексор снижает время простоя (сна).

Пояснение:



Пояснения слева буквально дословно можно перевести.

Преимущества такого способа перед блокирующим в том, что мы можем обрабатывать не один, а несколько дескрипторов. После получения статуса readable мы видим блокировку, но мультиплексирование снижает время блокировки, так как время ожидания готовности каждого сокета будет больше, чем время ожидания готовности первого сокета из пула сокетов, т. е. Вероятность готовности сокета в пуле выше, чем вероятность готовности одного сокета.

Если исключить мультиплексирование, то IO будет осуществляться с помощью обычной блокировки, при этом случайным образом выбирается соединение, из которого будет выполняться чтение. Поскольку первый сокет обрабатывается, могут подоспеть другие соединения, в результате время простоя сокращается.

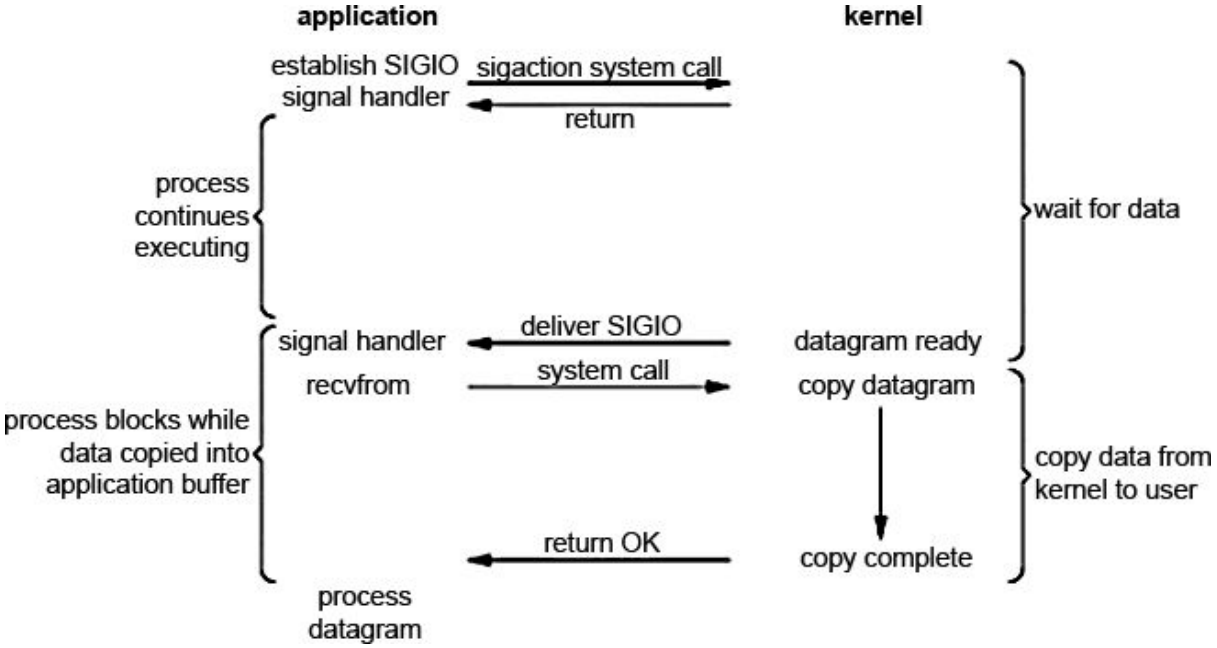
Способ, похожий на мультиплексирование, заключается в том, что несколько процессов или потоков пытаются выполнить тоже самое. Каждый из них при этом выполняет блокирующий IO.

Недостатки такого вида:

1) «дорогие» потоки с точки зрения команд

2) если взять Python, то в нем существует GIL – global interpreter lock, накладывающая некоторые ограничения на потоки, а именно – нельзя использовать несколько процессоров одновременно

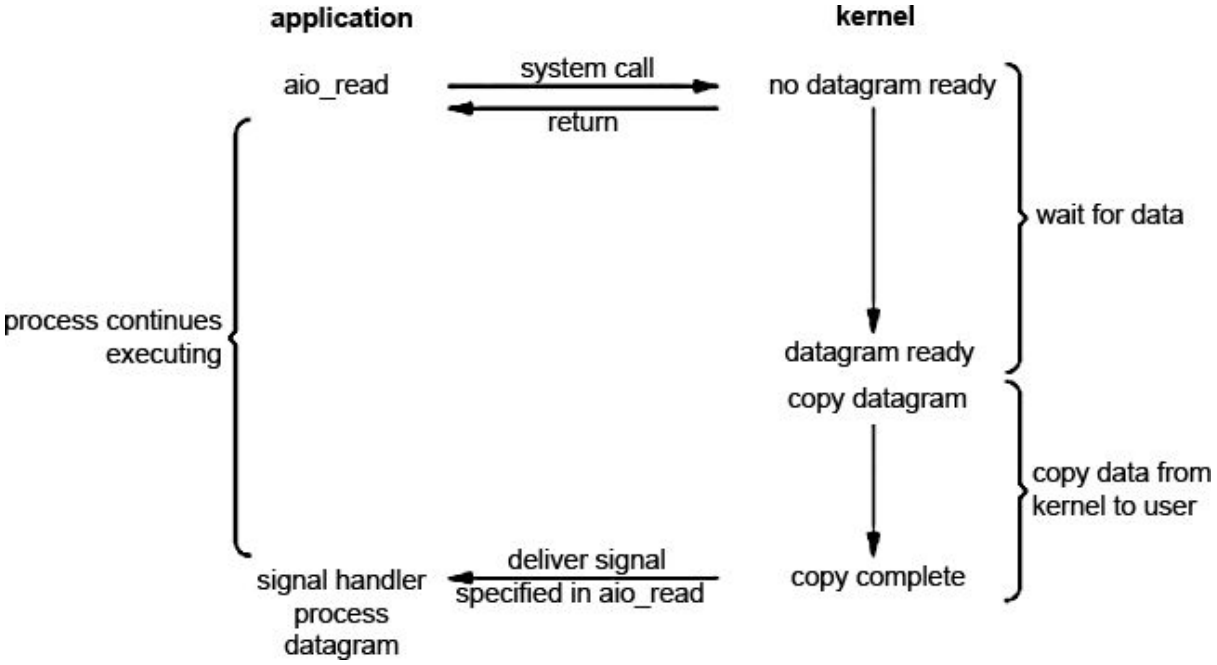
4) SIGIO (IO, управляемый сигналами)



Вначале необходимо установить параметры сокета для работы с сигналами и назначить обработчик сигналов (signal handler) с помощью системного вызова sigaction. Результат возвращается мгновенно и приложение, следовательно, не блокируется. Фактически, всю работу на себя берет ядро, т.к. оно отслеживает когда данные будут готовы и посылает нам сигнал SIGIO, который вызывает установленный на него обработчик (функция обратного вызова, callback). Соответственно сам вызов recvfrom можно сделать либо в обработчике сигнала, либо в основном потоке программы. Насколько я могу судить, Здесь есть одна проблема — сигнал для каждого процесса такого типа может быть только один. Т.е. за раз мы можем работать только с одним fd.

5) Async IO

Определен спецификацией POSIX, которая согласовала различия в функциях реального времени, появившиеся в разных стандартах, и объединила их. Основная особенность функций – сообщение ядру о начале операции и уведомление приложения о том, что вся операция завершена incl. Копирование данных



Очевидно, что все это относится только к системам массового обслуживания.

В AIO внимание сосредоточено на 2 моментах

1) на возможности определить, что IO можно выполнить быстро

2) на завершении IO-операции в любом случае (в случае невозможности выполнения IO сразу – возврат ошибки, и в случае возможности выполнения IO)

Основное отличие от 4й модели – в том, что в модели, управляемой сигналом: в ней ядро сообщает, когда операция IO может быть инициирована, а в AIO ядро сообщает о завершении операции IO. Системный вызов AIO возвращается сразу и приложение не блокируется, пока ждет завершения операции IO. Таким образом мы можем выступить критиками самой первой диаграммы.

Билет 20

1. Процессы: процесс как единица декомпозиции системы. Контекст процесса: аппаратный и полные контексты. Переключение контекста. Классификация алгоритмов планирования; алгоритм адаптивного планирования; ситуация - бесконечное откладывание - причины возникновения. Система приоритетов в ОС Windows и ОС Linux. Пересчет приоритетов.

Процесс - программа в стадии выполнения. Является единицей декомпозиции системы. Является потребителем системных ресурсов.

Контекст процесса. Переключение контекста.

Различается **аппаратный** и **полный** контекст.

Аппаратный – это содержимое регистров процессора (РОНы, счетчики команд, адресные регистры). Переключение аппаратных контекстов поддерживается аппаратно: есть комманда PUSHA. Ни одна система не позволяет оперировать процессу с устройствами напрямую, так как это сводит на нет любую защиту. На самом деле, read/write – системные вызовы. Они начинают выполняться в режиме пользователя (код программы), потом происходит переключение в режим ядра (реэнтирабельный код ОС) ==> произойдет переключение аппаратного контекста.

Полный — аппаратный + информация о выделенных процессу ресурсах: например, выделенная ему память. Память процесса – память, которая выделена процессу – описывается соответствующими таблицами. Очевидно, что когда происходит переключение процессов: он исчерпал свой квант или происходит системный вызов, то кроме аппаратного контекста необходимо иметь инфу о тех ресурсах, которые были выделены процессу, т. е. происходит переключение полного контекста. Каждый процесс имеет свою таблицу памяти. Любой процесс имеет свои таблицы страниц. Переключение полного контекста – затратная операция, так как аппаратно она не поддерживается. Теряется т.н актуальность кешей. В связи с затратностью по времени возник интерес к потокам.

Выделение процессора, т.е. процессорного времени, процессу выполняет диспетчер. Постановка процессов в очередь в соответствии с выбранной дисциплиной планирования выполняет планировщик.

Планирование процессов - это управление распределением ресурсов между конкурирующими процессами путем передачи им управления согласно некоторой стратегии планирования. По-английски планировщик - scheduler.

Алгоритмы планирования классифицируются следующим образом:

E. алгоритмы без переключения и с переключением
F. планирование бесприоритетное и с приоритетами
G. если у нас приоритетное планирование, то алгоритмы могут быть с вытеснением и без вытеснения

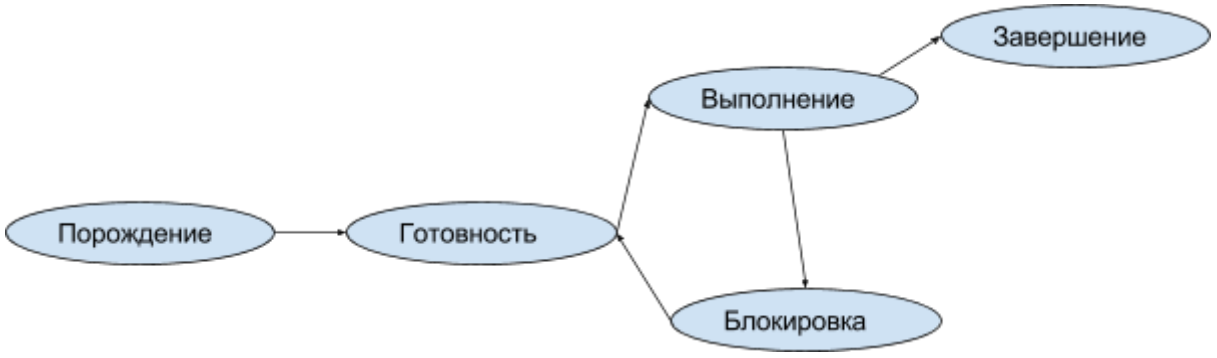
Н. приоритеты могут быть статистические и динамические, абсолютными и относительными.

На планирование, прежде всего, оказывает влияние тип ОС, т.е. в разных ОС реализовываются разные принципы планирования.

Для систем пакетной обработки рассмотрим следующие алгоритмы и классифицируем их:

5. **fifo - first in first out -** без переключения, без приоритетов. В соответствии с таким алгоритмом процесс будет выполняться от начала до конца.
6. **shortest job first -** наикратчайшее задание выполняется первым. это приоритетное планирование без переключения и вытеснения. Этот алгоритм имеет очень негативное свойство - бесконечное откладывание. Длинные задания, требующие много процессорного времени, могут постоянно откладываться.
7. **shortest remaining time -** задание с наименьшим временем оставшегося выполнения выполняется в первую очередь. Выполняющийся процесс может быть прерван, если в очередь поступит процесс с меньшим оценочным временем выполнения, чем то время, которое процессу осталось до его завершения. Это алгоритм с вытеснением без переключения. Для реализации этого алгоритма надо следить за текущим временем обслуживания. Очевидно, что такой алгоритм требует дополнения в диаграмму состояний - выполнение может быть прервано вытеснением в состояние готовности.

// Если вдруг попросит диаграмму состояний

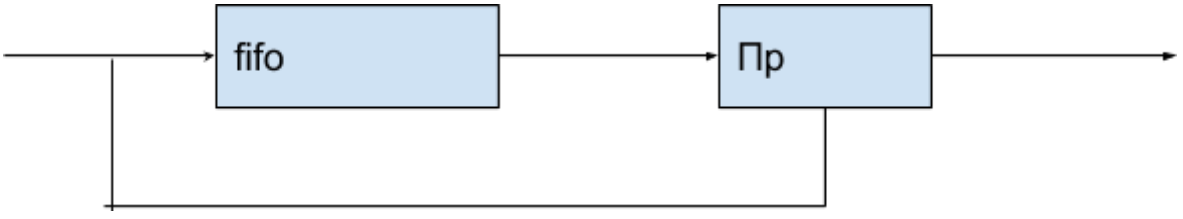


8. **HRN - highest response ratio next -** наибольшее относительное время ответа. Учитывается время ожидания готовых процессов. Чем выше время ожидания, тем выше приоритет процесса, т.е. это динамические процессы. *p=(tw-ts)/ts,* где *tw* - время ожидания, *ts* - запрошенное время.

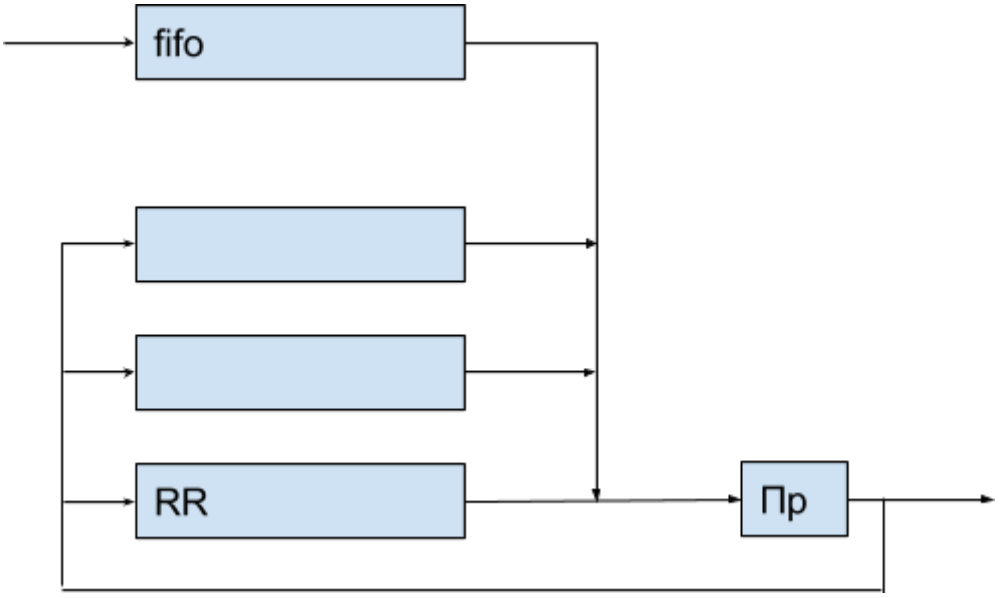
Рассмотренные алгоритмы являются алгоритмами, используемые в системах пакетной обработки.

Алгоритмы систем разделения времени

3. RR. С переключением, без вытеснения, без переключения.



4. Адаптированное планирование. На каждом уровне приоритета может находиться одновременно большое количество процессов. В нее попадают вновь созданные процессы и процессы после ожидания операций ввода-вывода. Квант для этой очереди выбирается таким образом, чтобы наибольшее количество процессов успело или завершить или выдать запрос ввода-вывода. Если за выделенный квант процесс не сделал ничего, то он помещается в низкий уровень.



Windows и Unix - операционные системы разделения времени с динамическими приоритетами и вытиснением, т.е. более приоритетный процесс вытесняет менее приоритетный. В Windows происходит путем явного выявления...

2. Файловая подсистема proc. Загружаемые модули ядра: правила программирования и сборки. Перемещение данных между ядром и пользовательским пространством с помощью API - функций. Пример

ФС /proc фактически представляет из себя интерфейс ядра, позволяющий читать данные в адресном пространстве ядра, управлять процессами, получать информацию о процессах и ресурсах, которые эти процессы используют. При этом используется стандартный интерфейс ФС и системных вызовов. /proc виртуальна, потому что файлы этой ФС не существуют на диске, а генерируются кодом ядра при каждом обращении для чтения файла /proc. Это текстовый файл, который отображает информацию об определенных частях ядра. По умолчанию запись/чтение файлов /proc разрешены только для владельцев. Указав PID процесса, мы перейдем в поддиректорию, в которой хранятся данные о данном процессе: /proc/<PID>.

В ней находятся файлы и следующие поддиректории:

	тип	содержание
cmdline	файл	Арг. Командной строки
cwd	Симв. ссылка	Директория процесса

environ	файл	Список окружения процесса
exe	Симв. ссылка	Образ процесса
fd	директория	Ссылки на файлы, исп. процессом
root	Симв ссылка	Указывает на корень ФС процесса
stat	файл	Информация о процессе

Еще один способ: /proc/self. В этой поддиректории находится каталог с данными.
Пример программы, печатающей информацию об окружении:

```
#include <stdio.h>
#define BUF_SIZE 0x100
int main (int argc, char *argv[])
{
        char buf [BUF_SIZE];
        int l, i;
        FILE *f;
        f=fopen(«/proc/self/environ», «r»);
        while ((len = fread(buf,1,buf_size,f)>0)
        {
                for (i=0;i<len;i++)
                if (buf[i]==0)
                buf[i]=0;
                buf[len]=0;
                printf(«%s»,buf);
        }
        fclose(f);
        return 0;
}
```

Разделение слов происходит по символу 10.
Команды: ls pci или apm
/proc/pci – PCI
/proc/apm – информация о состоянии батарей.
Для работы с ФС /proc используются модули ядра.

```
Struct proc_dir_entry
{
        unsigned short low_ino; //номер индекса inode-файла.
        unsigned short namelen;
        const char *name;
        mode_t mode;
        nlink_t nlink;
        uid_t uid;
        …
        struct proc_dir_entry *next, *parent, *subdir;
        void *data;
        ...
```

}

Копия индекса в памяти содержит поля, которых нет в дисковом inode: логический номер устройства, ответственные за блокировки поля, номер индекса (в дисковом индексе это поле не нужно, так как они хранятся в линейном порядке, и ядро идентифицирует его по смещению), ссылки на другие индексы для организации в ядре хеш-очереди. Кроме этого, ведется список свободных индексов, и счетчик, определяющий, сколько раз был открыт файл.

Функции для работы с /proc

Для создания файла типа proc_dir_entry используется функция create_proc_entry, или, в более поздних версиях, create_proc.

Struct proc_dir_entry *create_proc_entry(const char *name,
mode_t mode,
struct proc_dir_entry *base); // вызов

struct proc_dir_entry *entry = creat_proc_entry(name, mode, parent);

В результате создается запись с именем name, может быть указан относительный путь. Parent может быть указателем на proc_dir_entry, где может появиться новая запись (?).

Начиная с версии 3.10 стали использовать функцию proc_create_data(). В современных версиях функция переписана и называется proc_create().

Struct proc_dir_entry *kmod;

kmod = proc_create(kmod_proc_ent, 0644, NULL, &fops);

1 поле – static const char *Kmod_proc_ent= «kmod»;

В отличие от предыдущей функции есть опция &fops – это поле типа file_operations

Поддиректория создается proc_mkdir(), а символическая ссылка – proc_symlink().

(access_ok()) – проверяется возможность доступа

(copy_from_user()) – копирование информации из userspace в ядро

copy_to_user() – наоборот.

В последних версиях можно использовать sprintf();

kproc_dir = proc_mkdir(kproc_proc_dir, NULL);

kproc_symlink = proc_symlink(kproc_symlink, NULL);

Необходимо определить callback-функции, такие как open, read, write, iocontrol. Полный перечень функций в struct file_operations. Как правило, в модулях определяются не все функции, только нужные. Неопределенные функции из структуры никуда не деваются.

Загружаемые модули (отсебятина)

Необходимо определить функции инициализации и выгрузки модуля. Подключаются они двумя макросами:

module_init(my_init);

module_exit(my_cleanup);

где my_init и my_cleanup - функции загрузки и выгрузки модуля соответственно.

Перемещение данных между ядром и пользовательским пространством с помощью API - функций

Если посмотреть на данные с точки зрения внешнего устройства, то главной проблемой, связанной с этими двумя методами, является необходимость передачи данных между адресными пространствами ядра и пользоваетеля. Эти действия не могут осуществляться по указателям с использованием memcopy. Userspace не может

использоваться напрямую в режиме ядра по ряду причин, одной из которых является существенное отличие этих двух адресных пространств, связанное с тем, что АП пользователя (память в userspace) может быть выгружена. Для взаимодействия userspace с kernelspace необходимые спецсисвызовы opy_from_user(), copy_to_user().

Дальше с сайта ibm

access_ok	Проверяет, является ли указатель пространства пользователя действительным.
get_user	Получает простую переменную из пространства пользователя.
put_user	Помещает простую переменную в пространство пользователя.
clear_user	Очищает или обнуляет блок памяти в пространстве пользователя.
copy_to_user	Копирует блок данных из пространства ядра в пространство пользователя.
copy_from_user	Копирует блок данных из пространства пользователя в пространство ядра.
strnlen_user	Определяет размер строкового буфера в пространстве пользователя.
strncpy_from_user	Копирует строку из пространства пользователя в пространство ядра.

Функция access_ok

Функция access_ok используется для проверки действительности указателя пространства пользователя, по которому необходимо получить доступ. Входными аргументами этой функции являются указатель на начало блока памяти, размер блока и тип доступа (чтение или запись). Прототип функции определен следующим образом.

access_ok(type, addr, size);

Аргумент type может принимать значения VERIFY_READ и VERIFY_WRITE. Значение VERIFY_WRITE означает, что область памяти доступна не только для записи, но и для чтения. Если доступ к указанной области памяти разрешен, функция возвращает ненулевое значение (однако при попытке доступа может возникнуть ошибка -EFAULT). Эта функция просто проверяет, что адрес находится в пространстве пользователя, а не в пространстве ядра.

Функция copy_to_user

Функция copy_to_user копирует блок данных из пространства ядра в пространство пользователя. Входными аргументами этой функции являются указатель на буфер пространства пользователя, указатель на буфер ядра и размер блока (в байтах). В случае успеха функция возвращает нулевое значение, а в случае неудачного выполнения – числовое значение, равное количеству байтов, которые не удалось передать.

copy_to_user(to, from, n);

Функция проверяет (с помощью access_ok), доступен ли указатель пространства пользователя для записи, после чего выполняет вызов внутренней функции

__copy_to_user, которая, в свою очередь, вызывает функцию __copy_from_user_inatomic, определенную в заголовке ./linux/arch/x86/include/asm/uaccess_XX.h, где XX – разрядность процессора (32 или 64). Наконец, после определения размера блоков для копирования (1, 2 или 4 байта), происходит вызов функции __copy_to_user_ll, которая и выполняет основную работу. На старых архитектурах процессоров, предшествующих i486 (в которых WP-бит не принимал прерывания на обработку в привилегированном режиме), страничные таблицы могли меняться в произвольные моменты времени, поэтому требовалось закреплять в памяти требуемые страницы, чтобы они не могли быть выгружены в момент обращения к ним. В архитектурах от i486 и выше этот процесс представляет собой простое оптимизированное копирование.

Функция copy_from_user

Функция copy_from_user копирует блок данных из пространства пользователя в буфер ядра. Входными аргументами этой функции являются указатель на буфер назначения в пространстве ядра, указатель на буфер источника в пространстве пользователя и размер копируемого блока (в байтах). Так же как и функция copy_to_user, в случае успеха эта функция возвращает нулевое значение, а в случае неудачного выполнения – числовое значение, равное количеству байтов, которые не удалось скопировать.

copy_from_user(to, from, n);

Функция проверяет (с помощью access_ok), доступен ли указатель на буфер в пространстве пользователя для чтения, после чего выполняется вызов внутренней функции __copy_from_user и последующий вызов функции __copy_from_user_ll. Далее в зависимости от архитектуры процессора выполняется копирование из буфера пользователя в буфер ядра (с обнулением отсутствующих байтов). Оптимизированные функции, написанные на ассемблере, имеют возможности управления.

Пример из лабы с модулем ядра

```
ssize_t fortune_write(struct file *file, const char *buf, size_t count, loff_t *f_pos)
{
	int free_space = (COOKIE_BUF_SIZE - write_index) + 1;
	if (count > free_space) {
		printk(KERN_INFO "fortune: Cookie pot full.\n");
		return -ENOSPC;
	}
	if (copy_from_user(cookie_buf+write_index, buf, count)) {
		printk( KERN_ERR "fortune: ERROR. couldn't copy data from userspace\n" );
		return -EFAULT;
	}
	write_index += count;
	cookie_buf[write_index-1] = 0;
	return count;
}
```

Билет 21

1. Взаимодействие процессов в Unix - сигналы, очереди сообщений, программные каналы, разделяющие область памяти: системные вызовы, отличия, примеры.

Средства взаимодействия процессов в Unix System V.
Inter-Process Communicaion.
К IPC SV относятся
1) Сигналы
2) Семафоры
3) Программные каналы
4) Очереди сообщений
5) Разделяемая память

Сигналы: это программное средство, с помощью которого может быть прервано функционирование процесса в ОС Unix. Задачей сигнала является информирование процесса о ожидаемом (или нет) событии. Сигналы возникают в системе в результате выполнения определенных событий и информирование процесса совершенно не зависит от того, хочет ли процесс получать информацию или нет.

Как правило, получение некоторым процессом сигнала означает для него необходимость завершить свое функционирование. В то же время, реакция процесса на принимаемый сигнал зависит от того, как сам процесс определяет свое поведение в случае приема им данного сигнала.

В классическом Unix имеется 20 сигналов с закодированным числом и мнемонической записью, все это хранится в файле signal.h.

```
#define SIGHUP 1        /* разрыв связи с терминалом демоны игнорируют
                           получение данного сигнала  */
#define sigint 2 /* ctrl/c */
....
#define sigkill 9 /* уничтожение процесса */
....
#define sigsegv 11 /* segmentation fault */
....
#define sigpipe 13 /* запись в канал есть, а чтения нет */
....
 #define sigalarm 14 /* прерывание от таймера */
....
#define SIGUSR1 16 /* пользовательский сигнал */
....
#define sigcld 18 /* потомок завершился */
....
//------
//не сигналы
#define sig_dfl 0 /* установки по умолчанию */
```

#define sig_ign 1 /* игнорирование сигнала */

Kill() и Signal()

int kill (int pid, int sig);

kill (pid, sig);

Сигнал sig будет послан всем процессам, родственным процессу pid.

Процессы одной группы могут получать одни и те же сигналы, но эти установки могут быть отменены, например, если

1) pid <= 1, то сигнал sig будет послан группе процессов

2) pid = 0, то сигнал sig будет послан всем процессам с id группы, совпадающими с id группы процесса, который вызвал kill, кроме процессов с id 0 и 1.

3) Pid = -1, то сигнал sig будет послан процессам, который имеют тот же UID, что и процесс, вызвавший kill.

Например, если вызвать kill(37,sigkill), это приведет к завершению процесса с id 37, а если kill(getpid(), sigalarm); то процесс сам для себя получит sigalarm и завершится.

Системный вызов signal() входит в библиотеку signal.h, не входит в стандарт POSIX1, но он входит в ANSI C, а он есть в любом Unix/Linux. void (*signal(int sig, void (*handler)(int)))(int);

Из прототипа функции следует, что реакцией на signal с аргументом handler на сигнал sig будет вызов обработчика сигнала handler. Сигнал может быть обработан разными способами.

1) Стандартным

2) Сигнал может быть замаскирован (signal SIGINT, SIG_IGN);

3) Собственный способ реакции на сигнал (должен быть свой обработчик сигнала)

Signal возвращает указатель на предыдущий обработчик сигнала, что можно использовать для восстановления стандартного обработчик сигнала.

```
#include <signal.h>
int main()
{
        void (*old_handler)(int)=signal(SIGINT, SIG_IGN);
        //обработка
        signal(SIGINT, old_handler);
}
#include <signal.h>
#include <string.h>
#include <unistd.h>
/*handler*/
void handler (int signum)
{
        char buf[200], *cp;
        int offset;
        strcpy(buf, "handler: caught signal \n");
        cp = buf+strlen(buf); // cp указывает на /0 -- конец строки
        if (signum > 100) // маловероятно
                offset = 3;
        else if (signum > 10)
                offset = 2;
```

```
        else offset = 1;
        cp+=offset;
        *cp--='\0';
        while (signum>0)
        {
                *cp --= (signum % 10) + '0';
        signum /= 10;
        }
        strcat(buf, "\n");
        (void) write (2, buf, strlen(buf));
}
// handler
int main (void)
{
        (void) signal (SIGINT, handler);
        for (;;)
        pause(); // подать сигнал
        return 0;
}
```

Pause() возвращается только тогда, когда сигнал был перехвачен и был выполнен возврат из функции, перехватывающей сигнал; возвращается -1. Это написано в man. В противном случае возвращается errno.

```
#include <iostream.h>
#include <signal.h>
void catch_sig(int sig_num)
{
        signal(sig_num, catch_sig);
        cout << "catch_sig: " << sig_num << endl;
}
int main (void)
{
        signal(SIGTERM, catch_sig);
        signal(SIGINT, SIG_IGN);
        signal(SIGSEGV, SIG_DFL);
        sleep(3);
        return 0;
}
```

В POSIX определена функция sigaction(). Она подобно signal() задает метод обработки сигнала и также возвращает указатель.

Прототип:

```
int sigaction(int sig_num, struct sigaction *action, struct sigaction *old_action);
```

Важно отметить, что сигналы сопровождают внешние по отношению к нашей выполняемой программа события (т.е. абсолютно асинхронные). Используя это, мы можем менять поведение нашей программы.

Sigsetjmp используется для отметки одной или нескольких позиций, а siglongjump – только в определенную позицию.

Программные каналы

В программе именованные pipes можно создавать:

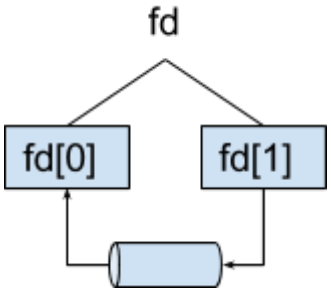
Mknod(name, IFIFO|ACCESS, 0);

int fd[2]; // fd – file descriptor pipe(fd);

inode (index node) – индексный узел. Без имени. Имеет дескриптор (файловый).

Процесс-потомок наследует дескрипторы программных каналов, открытых предком.

```
if (child_pid == 0)
{
        …
        close(fd[1]);
        read(fd[0], line, 100);
        …
}
else
{
        close(fd[0]);
        write(fd[1], test, strlen(test));
        …
}
```



Буферизуется на 3 уровнях: 1. В системной области памяти (при переполнении буфера имеющие наибольшее время существования записываются на диск). Ограничение размера – повышение эффективности обмена.

Ни один процесс не может влезть в адресное пространство другого. Они могут обмениваться данными только через АП системы.

Доступ к БД – запрос через именованный программный канал.

Разделяемая память

В других средствах межпроцессового взаимодействия (IPC) обмен информацией между процессами проходит через ядро, что приводит к переключению контекста между процессом и ядром, т.е. к потерям производительности.

Техника разделяемой памяти позволяет осуществлять обмен информацией через общий для процессов сегмент памяти без использования системных вызовов ядра. Сегмент разделяемой памяти подключается в свободную часть виртуального адресного пространства процесса. Таким образом, два разных процесса могут иметь разные адреса одной и той же ячейки подключенной разделяемой памяти.

После создания разделяемого сегмента памяти любой из пользовательских процессов может подсоединить его к своему собственному виртуальному пространству и работать с ним, как с обычным сегментом памяти. Недостатком такого

обмена информацией является отсутствие каких бы то ни было средств синхронизации, однако для преодоления этого недостатка можно использовать технику семафоров.

shmget — создание сегмента разделяемой памяти с привязкой к целочисленному идентификатору, либо анонимного сегмента разделяемой памяти (при указании вместо идентификатора значения IPC_PRIVATE);

shmctl — установка параметров сегмента памяти;

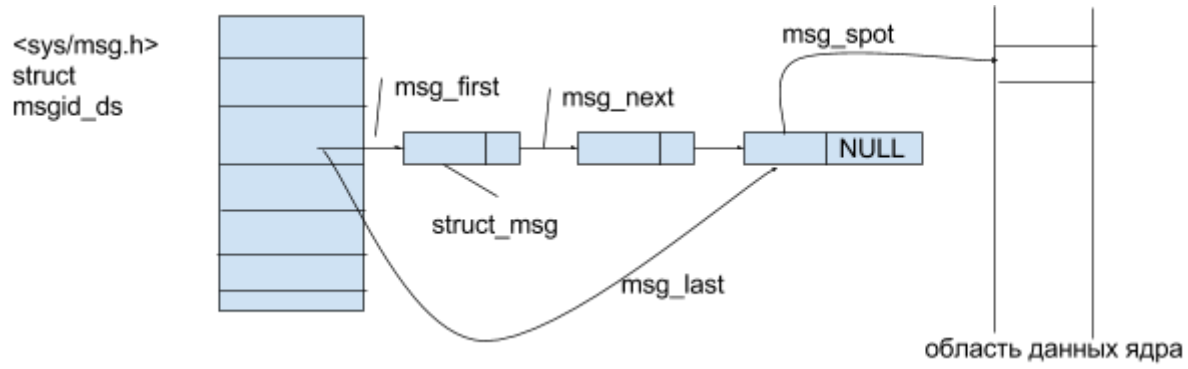
shmat — подключение сегмента к адресному пространству процесса;

shmdt — отключение сегмента от адресного пространства процесса.

```
#include <string.h>
#include <sys/stat.h>
#include <sys/ipc.h>
#include <sys/shm.h>
int main()
{
        int perms = S_IRWXU | S_IRWXG | S_IRWXO;
         int fd = shmget(100, 1024, IPC_CREATE | perms);
        if (fd == -1)
        {
                perror(“shmget”); exit(1);
        }
        char *addr = (char *)shmat(fd, 0, 0);
        if (addr == (char *)(-1))
        {
                perror(“shmat”);
                exit(1);
        }
        strcpy(addr, “Hello”);
        if (shmdt(addr) == -1)
        {
                perror(“shmdt”);
        }
        return 0;
}
```

Системные ограничения: SHMMNI – число разделяемых областей SHMMIN – минимальный размер РП в байтах SHMMAX – максимальный размер РП в байтах

Очереди сообщений



При помещении сообщения в очередь создается новая запись и помещается в конец списка записей указанной очереди. В каждой записи: тип сообщения, число байт данных, указатель на сообщение. Когда процесс выбирает сообщение из очереди, ядро копирует его в адресное пространство процесса, после чего запись удаляется.

msgget(), msgctl(), msgsud(), msgrev()

Пример

новая очередь сообщений stomp и устанавливает следующие права: rw-r---w-. Если msgget() успешен, то сообщение hello, иначе -- 15 (???); этот вызов неблокирующий.

```
#include <string.h>
#include <syslipc.h>
#include <sys/msg.h>
#define MSGMAX 1024
struct mbuf
{
   long mtype;
   char mtext[MSGMAX];
} mobj={15,"hello"};
int main()
{
   int fd = msgget(100,IPC_CREATE|IPC_EXCL}
   if (fd ==-1 || msgsnd(fd, &mobj, strlen(mobj.text)+1, IPC_NOWAIT))
      perror("message");
   return 0;
}
```

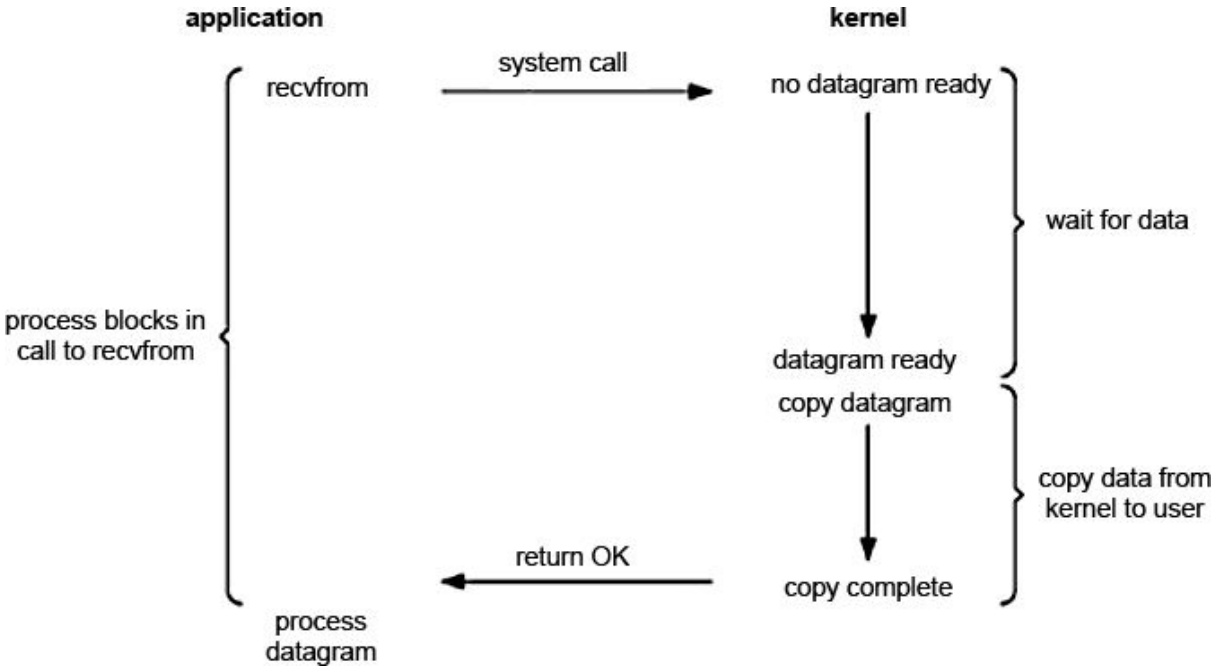
2. Модели ввода-вывода, анализ, схемы

5+1 моделей ввода/вывода

Все эти модели в самом общем виде могут быть представлены следующей диаграммой:

	Блокирующие	Нелокирующие
Синхронные	Read/Write	Read/Write (O_NONBLOCK)
Асинхронные	I/O multiplexing (select/poll)	AIO

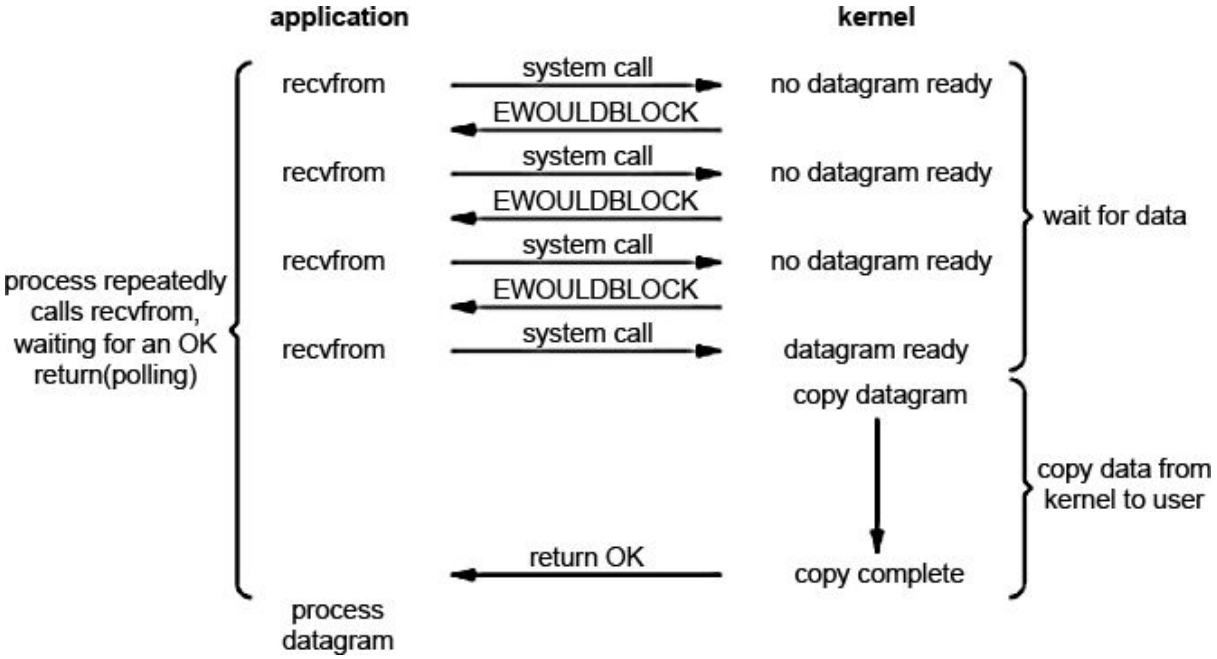
1) Блокирующий ввод вывод, по сути являющийся синхронным.



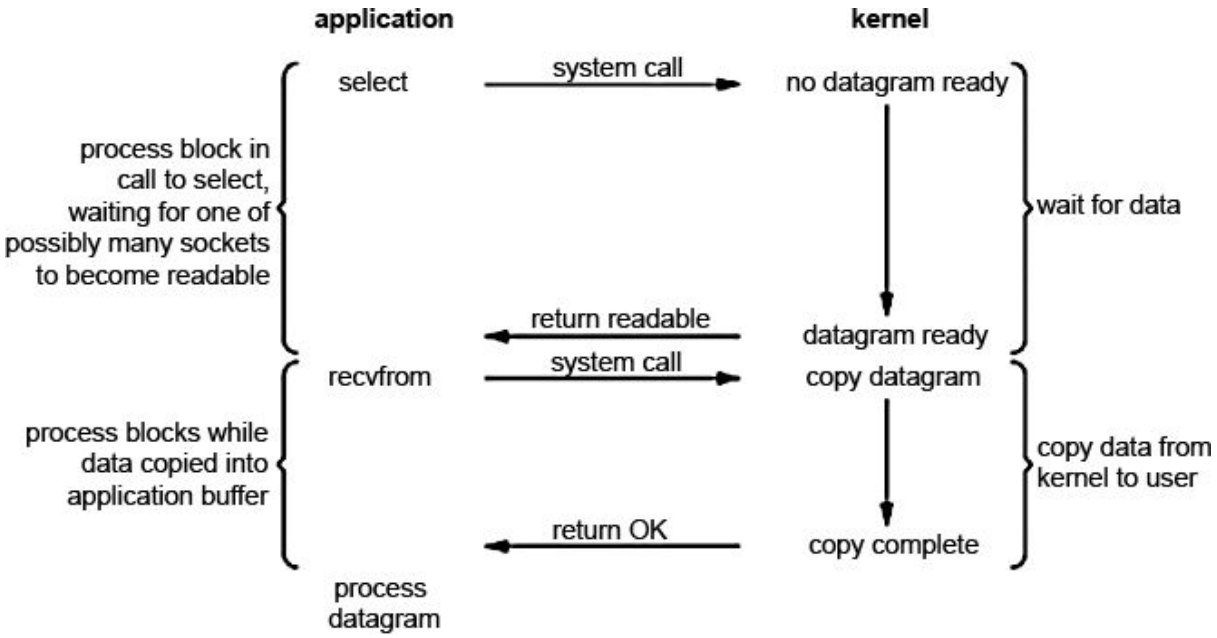
Понятно, что супервизор вызовет соответствующий драйвер устройства, он инициализирует процесс ввода/вывода, и при завершении IO это будет отмечено сигналом прерывания, …, будут вызваны callback-функции драйвера, и данные будут переданы приложению.

2) non-blocking IO (polling)

На этой основе можно создать цикл, который постоянно вызывает recvfrom (обращается за данными) для сокетов, открытых в неблокирующем режиме. Этот режим называется опросом (поллинг/polling) т.к. приложение все время опрашивает ядро системы на предмет наличия данных.

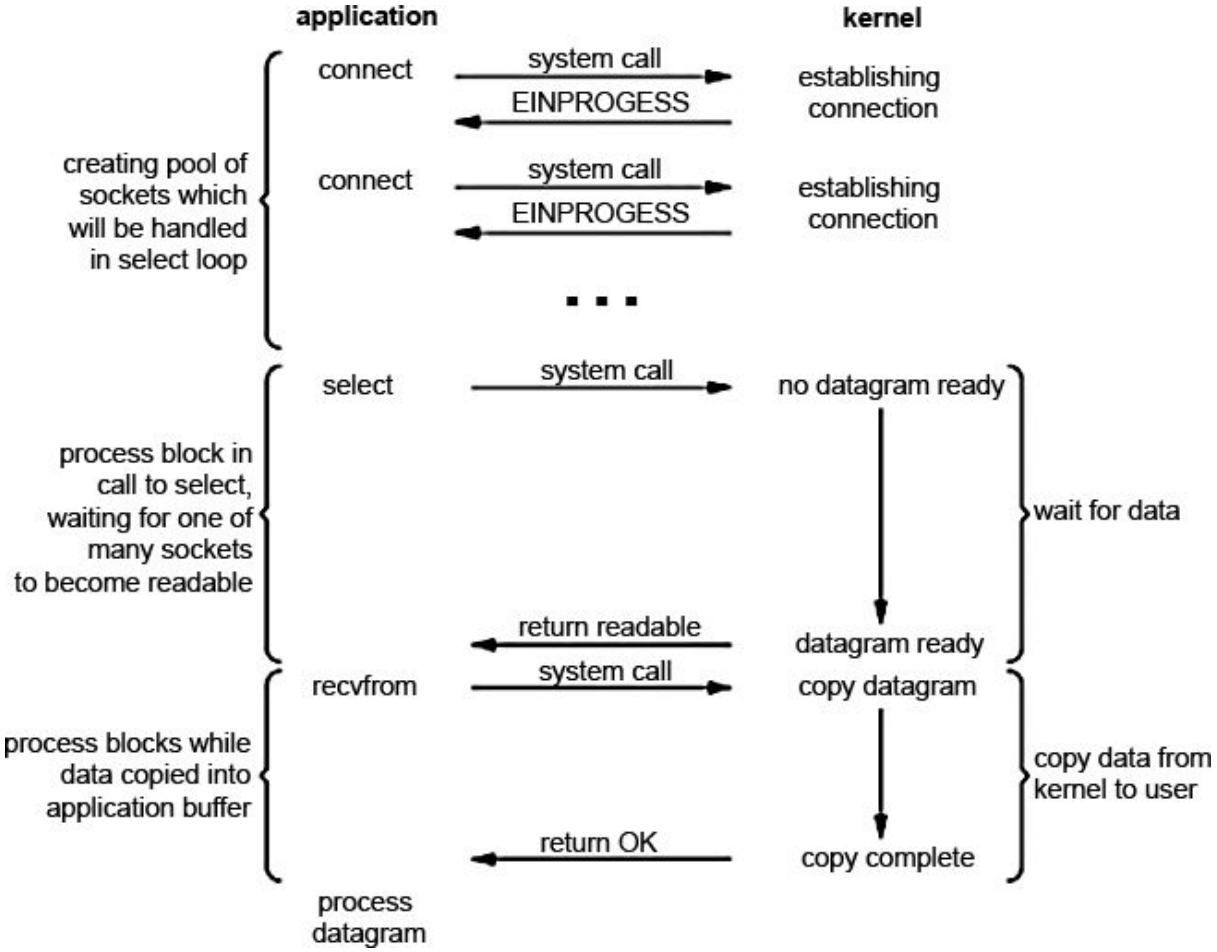


3) мультиплексирование (уплотнение) ввода/вывода



Приложение блокируется при вызове select'а ожидая когда сокет станет доступным для чтения. Затем ядро возвращает нам статус readable и можно получать данные помощью recvfrom. На первый взгляд — сплошное разочарование. Та же блокировка, ожидание, да и еще 2 системных вызова (select и recvfrom) — высокие накладные расходы. Но в отличии от блокирующего метода, select (и любой другой мультиплексор) позволяет ожидать данные не от одного, а от нескольких файловых дескрипторов. Надо сказать, что это наиболее разумный метод для обслуживания множества клиентов, особенно если ресурсы достаточно ограничены. Почему это так? Потому что мультиплексор снижает время простоя (сна).

Пояснение:



Пояснения слева буквально дословно можно перевести.

Преимущества такого способа перед блокирующим в том, что мы можем обрабатывать не один, а несколько дескрипторов. После получения статуса readable мы видим блокировку, но мультиплексирование снижает время блокировки, так как время ожидания готовности каждого сокета будет больше, чем время ожидания готовности первого сокета из пула сокетов, т. е. Вероятность готовности сокета в пуле выше, чем вероятность готовности одного сокета.

Если исключить мультиплексирование, то IO будет осуществляться с помощью обычной блокировки, при этом случайным образом выбирается соединение, из которого будет выполняться чтение. Поскольку первый сокет обрабатывается, могут подоспеть другие соединения, в результате время простоя сокращается.

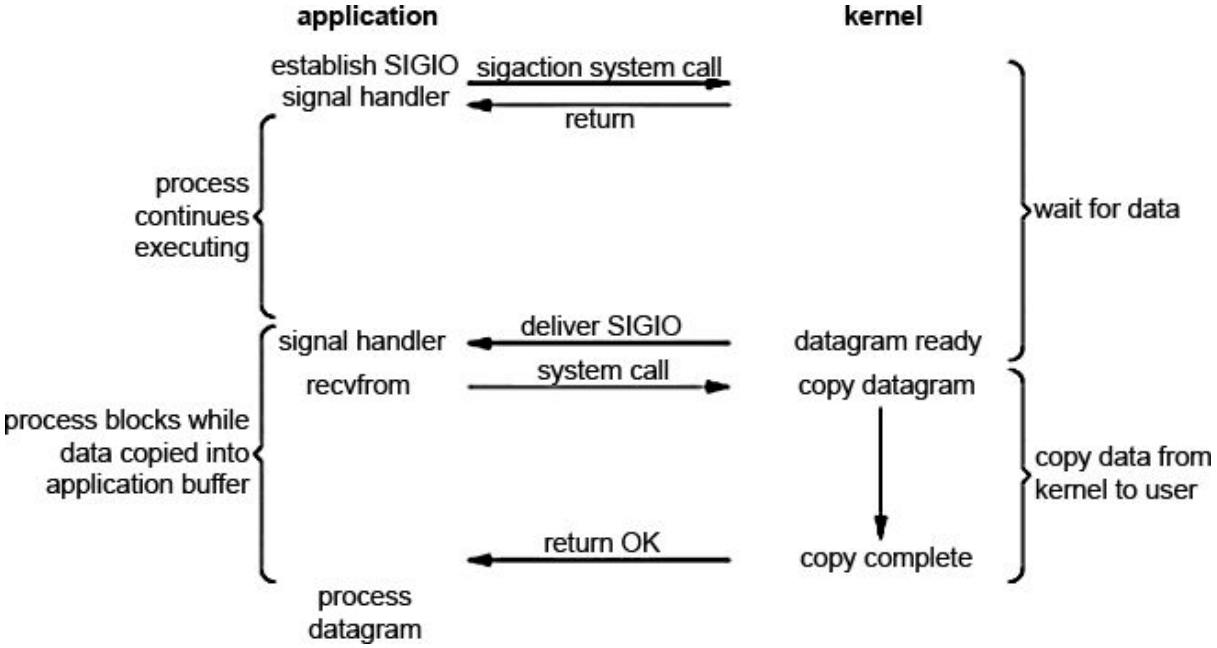
Способ, похожий на мультиплексирование, заключается в том, что несколько процессов или потоков пытаются выполнить тоже самое. Каждый из них при этом выполняет блокирующий IO.

Недостатки такого вида:

1) «дорогие» потоки с точки зрения команд

2) если взять Python, то в нем существует GIL – global interpreter lock, накладывающая некоторые ограничения на потоки, а именно – нельзя использовать несколько процессоров одновременно

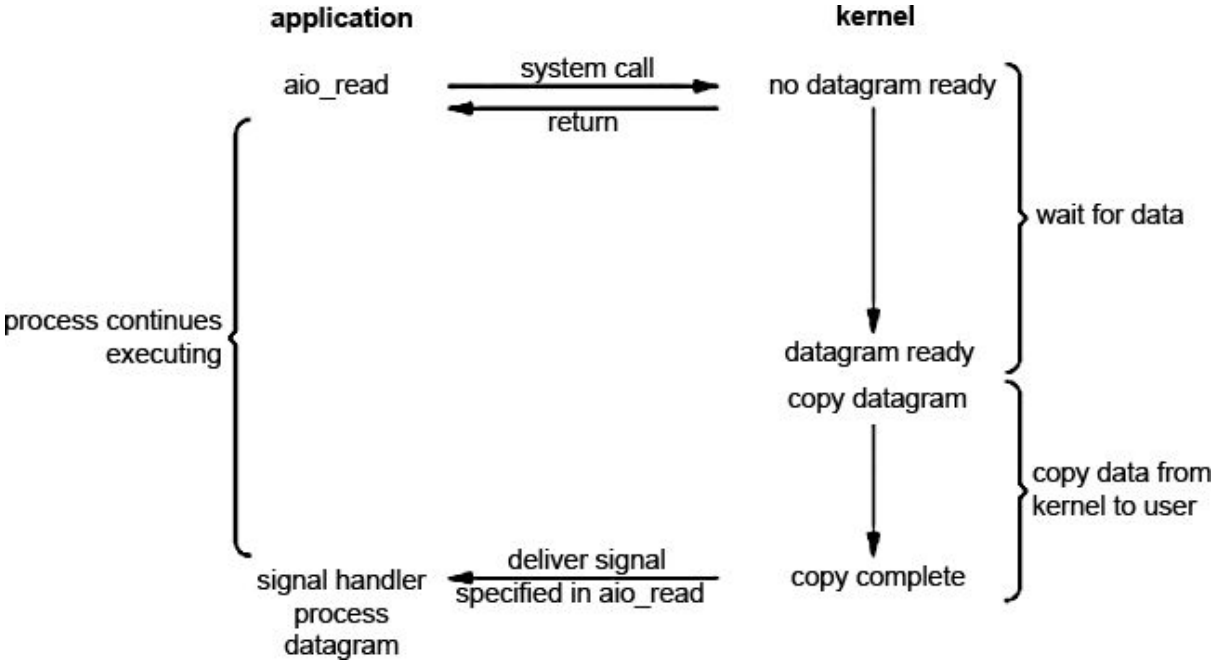
4) SIGIO (IO, управляемый сигналами)



Вначале необходимо установить параметры сокета для работы с сигналами и назначить обработчик сигналов (signal handler) с помощью системного вызова sigaction. Результат возвращается мгновенно и приложение, следовательно, не блокируется. Фактически, всю работу на себя берет ядро, т.к. оно отслеживает когда данные будут готовы и посылает нам сигнал SIGIO, который вызывает установленный на него обработчик (функция обратного вызова, callback). Соответственно сам вызов recvfrom можно сделать либо в обработчике сигнала, либо в основном потоке программы. Насколько я могу судить, Здесь есть одна проблема — сигнал для каждого процесса такого типа может быть только один. Т.е. за раз мы можем работать только с одним fd.

5) Async IO

Определен спецификацией POSIX, которая согласовала различия в функциях реального времени, появившиеся в разных стандартах, и объединила их. Основная особенность функций – сообщение ядру о начале операции и уведомление приложения о том, что вся операция завершена incl. Копирование данных



Очевидно, что все это относится только к системам массового обслуживания.

В AIO внимание сосредоточено на 2 моментах

1) на возможности определить, что IO можно выполнить быстро

2) на завершении IO-операции в любом случае (в случае невозможности выполнения IO сразу – возврат ошибки, и в случае возможности выполнения IO)

Основное отличие от 4й модели – в том, что в модели, управляемой сигналом: в ней ядро сообщает, когда операция IO может быть инициирована, а в AIO ядро сообщает о завершении операции IO. Системный вызов AIO возвращается сразу и приложение не блокируется, пока ждет завершения операции IO. Таким образом мы можем выступить критиками самой первой диаграммы.

Билет 22.

1. Монитор: определение, причины возникновения. Простой, кольцевой. Пример реализации монитора читатели-писатели на винде

Мониторы

Для структурирования средств взаимоисключения были введены мониторы.

Идея его заключается в создании механизма, который унифицировал бы взаимодействие параллельных процессов по разделяемым данным, кодам и обеспечивал правильное функционирование и не приводил к искажению данных.

Монитор – это структура, состоящая из данных и подпрограмм, которые могут изменять или обращаться к этим данным. Монитор начинается ключевым словом monitor, они похожи на классы, хоть и появились они раньше. Монитор защищает свои данные, это заключается в том, что к данным можно обратиться только через подпрограммы монитора, т.е. монитор сам является ресурсом. Он гарантирует, что в каждый момент времени ресурсы монитора могут использоваться только одним процессом – захвативший монитор процессор – это процесс, находящийся в мониторе. Все другие процессы, пытающиеся захватить монитор – они вне монитора и стоят к нему в очереди.

Использование монитора приводит к снижению производительности – снижение может достигать 70ти %.

Определены две операции – wait и signal.

Введен специальный тип переменных – condition, условие. Для каждой отдельной взятой причины, по которой процесс может быть переведен в состояние ожидания, назначается свое “условие”.

Рассмотрим: простой монитор, монитор читателя-писателя, кольцевой буфер

4. Простой монитор: обеспечивает выделение ресурсов произвольному числу процессов.

```
resource: monitor
var
   busy: logical;
   x: condition;
```

procedure acquire;
begin
  if busy then wait(x);
  busy := true;
end;
	procedure release;
	begin
	  busy:=false;
	  signal(x);
	end;

```
begin
   busy:=false;
end;
```

5. Кольцевой буфер: такой образ характерен для spooler-буферов.

```
resource: monitor;
```

```
var
   bcircle: array[0,...,n-1] of type;
   pos: 0..n; //тек.поз
   j: 0..n-1; // заполняемая позиция
   k: 0..n-1; // освобождаемая позиция
   bufferfull, bufferempty:condition;
```

procedure producer (data:type);
begin
  if pos=n then wait(bufferempty);
  bcircle[j]:=data;
  pos+=1;
  j:=(j+1) mod n;
  signal(bufferfull);
end;
	procedure consumer(var data: type);
	  begin
	  if pos = 0 then wait(bufferfull);
	  data := bcircle[k];
	  pos := pos-1;
	  k := (k+1) mod n;
	  signal (bufferempty);
	end;

```
begin
   pos:=0;
   j:=0;
   k:=0;
end.
```

6. Монитор читателя-писателя (Хоара) (Джеффри Рихтер “Windows для профессионалов” для лабораторных): там 4 функции: startread, stopread, startwrite, stopwrite. Два типа процессов: читатели (могут только читать) и писатели (могут изменять данные). Задача важна: в реальной жизни она обеспечивается в билетных системах, там, где есть конкретное указание “день-время-место”. Характерна для ядра ОС.

```
   resource:monitor;
   var
      nr: integer; //читатели
      wrt: logical; //писатель
      c_read, c_write : condition;
```

procedure startread;
begin
  if wrt or turn(c_write) then
  wait(c_read);
  nr+=1;
  signal(c_read);
end;
	procedure startwrite;
	begin
	  if hr > 0 or wrt then wait(c_write);
	  wrt := true;
	end;

procedure stopread;
begin
  nr-=1;
  if nr = 0 then
  signal(c_write);
end;
	procedure stopwrite;
	  begin
	  wrt:=false;
	  if turn(c_read) then
	  signal(c_read);
	  else signal(c_write);

	end;

```
begin
   nr:=0;
   wrt:=false;
end.
```

2. Буферизация и кэширования. Управление буферами. Распределение Slab

SLAB-распределение.

SLAB – пластина, лист, брусок. Такое распределение – это механизм управления памятью, обеспечивающий устранение фрагментации → достижения более эффективного использования памяти. Основой этого способа является хранение информации о ранее размещенных, а затем удаленных из памяти объектов. Такой подход основывается на наблюдениях, показывающих: часто удаленные из памяти объекты загружаются снова.

Загрузка и выгрузка объектов приводит к фрагментации, поэтому это распределение при уничтожении объектов не приводит к мгновенному освобождению памяти, а открывает слот, который помещается в список свободных слотов. Следующий вызов для выделения памяти того же размера вернет указатель на слот памяти из списка свободных слотов. Такой подход снижает время распределения памяти и сокращает фрагментацию. Система предоставляет в ядре спецсистемные вызовы для работы с кешами SLAB.

SLAB представляет собой непрерывный участок памяти, обычно состоящий из нескольких смежных страниц. Кеш состоит из одного или нескольких SLAB.

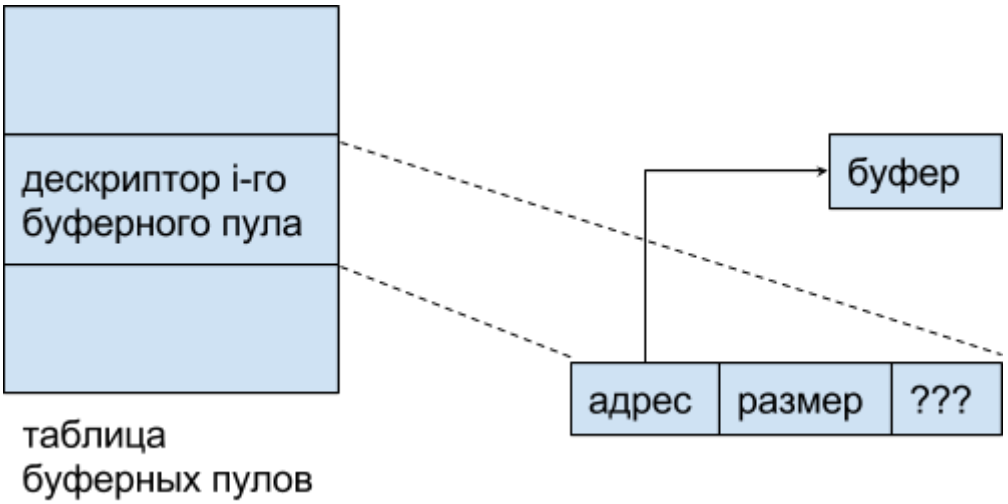
Linux/slab.h

Проблема буферизации и кеширования

Буфер – это область RAM для промежуточного хранения информации, обеспечивающая быстродействие(?) в процессе передачи данных между двумя процессами, процессом или устройством или двумя устройствами.

В отличие от буфера кеш-память – это быстродействующая RAM, в которую записывается копия команд или данных из более медленной памяти для ускорения процесса их дальнейшей обработки. Примером такого кеша является кеш TLB (ассоциативный по множеству буфер), предназначенный для хранения физических адресов таблиц, к которым было последнее обращение.

Использование кеш-памяти повышает или быстродействие системы в целом или какой-либо ее части. В базовой подсистеме смешивать понятие буферизации и кеширования не следует. Буфер часто содержит единственный набор данных, сгенерированный в системе. Кеш же по своему назначению всегда содержит копию данных, которые еще где-то хранятся в системе. При рассмотрении различных методов буферизации необходимо учитывать, что существует два типа устройств: блок-ориентированные (например, диски) и байт-ориентированные (терминалы, коммуникационные порты, позиционирующие устройства, другие устройства – не внешние запоминающие устройства) – в них передача информации осуществляется неструктурированным потоком информации. Особую роль буферизация играет в операциях IO.



Для управления буферами создается таблица, каждый элемент которой описывает 1 буфер. Буферный пул (буфер?) может быть создан

1) статически – в задаче при выделении памяти -- такой пул будет существовать все время существования задачи (процесса)
2) динамически – перед началом обмена с внешним устройством. Память освобождается, когда обмен завершен – в результате имеем более эффективную работу памяти.

Буферный пул формируется одним из следующих способов

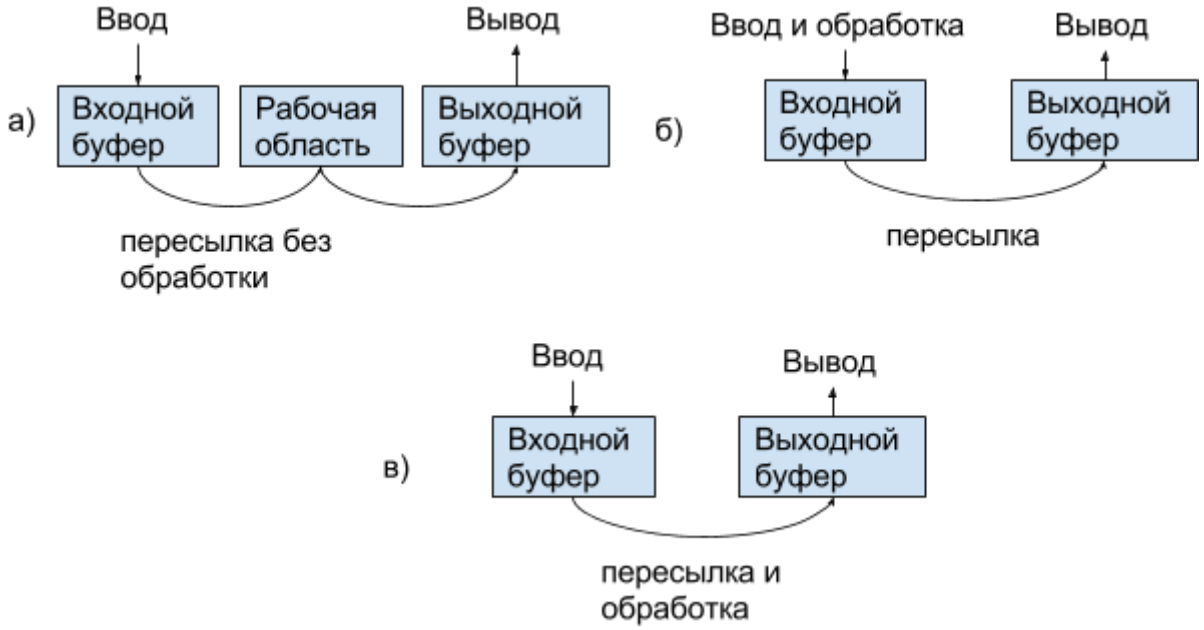
1) Созданием в программе спецобласти в памяти и последующим выполнением макрокоманды BUILT
2) выполнением макрокоманды GETPOOL (противоположная команда – FREEPOOL)
3) разрешением ОС автоматически создавать буферный пул при открытии обмена данных

Говоря об ОС, буферный пул может быть создан транслятором или в процессе работы самой ОС. ОС может создавать пул в результате соотв.

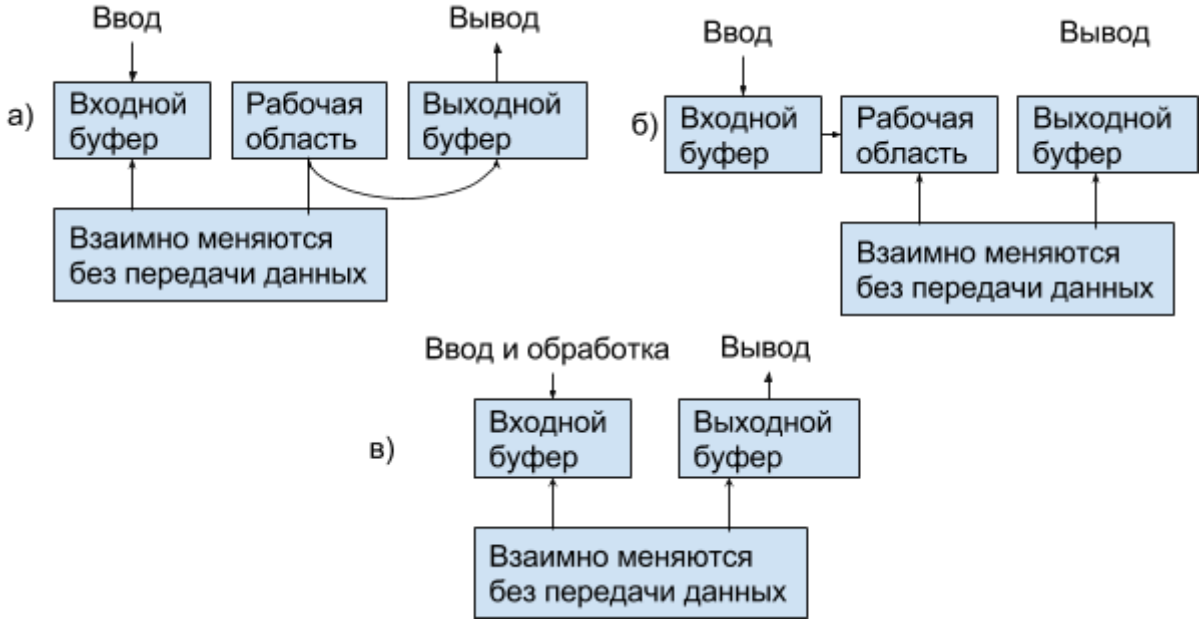
Запросов или автоматически.

При управлении буферами в самой ОС используются 2 системы управления

1) простая – при ней осуществляется значительное количество пересылок данных – это ведет к высоким накладным расходам, но достоинство схемы – в ее простоте. Применяется, когда объем обрабатываемых данных относительно невелик и обработка в оперативной памяти может занять большое время.



2) обменная (exchange buffering)



При обменной буферизации устраняются некоторые недостатки (...) за счет того, что буфера меняются функциями (на схеме А это видно). (надо будет, возможно, проговорить алгоритм на этой схеме). Очевидно, что обменная буферизация требует внимания со стороны ОС (ОС должна следить за функцией обмена буферов), но кроме этого, ОС должна менять адреса ввода у подпрограмм ввода и (...) обрабатывающего процесса пользователя. Это непростая задача.

При простой буферизации также может быть только 2 пересылки, но разница в том, что там буфер и рабочая область совмещаются физически – пока не закончится обработка, новая порция данных не может быть загружена.

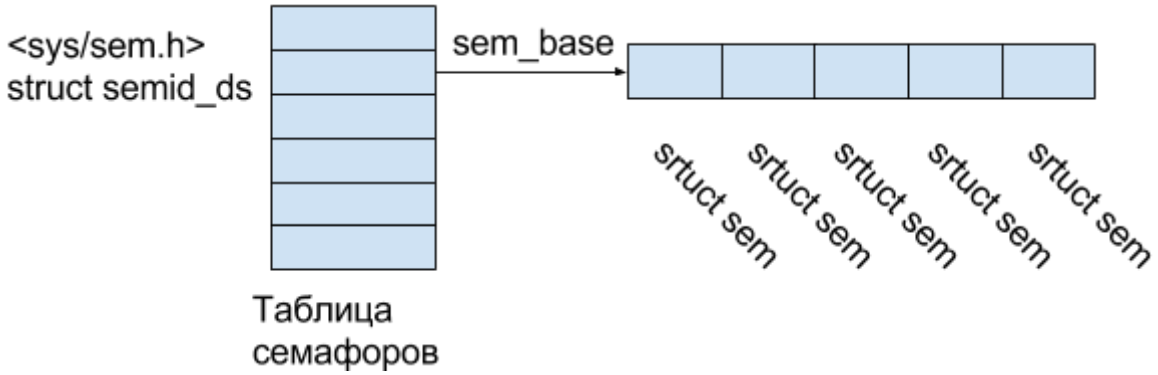
Билет Биня (N ??)

1. Процессы, семафоры и тупиковые ситуации. Примеры.

Семафоры

В Unix/Linux и Windows они объединяются в группы. Одним важнейшим свойством группы семафоров является такое: одним неделимым действием можно изменить часть или все семафоры набора. Семафоры поддерживаются в адресном пространстве ядра таблицей семафоров, в которой отслеживаются все создаваемые наборы семафоров. Каждый набор описывается соотв. Структурой:

1. Имя – целое число, присваивается процессом, создавшим набор
2. UID создателя набора семафоров. Процесс, эфф. UID которого совпадает с UID создателя может изменять параметры набора или удалять.
3. Права доступа. В Unix/Linux всегда они устанавливаются единообразно: rwxrwxrwx
4. Количество семафоров в наборе
5. Время изменения одного или нескольких значений семафоров каким-либо процессом
6. Время последнего изменения управляющих параметров набора каким-либо процессом
7. Указатель на массив семафоров.



На семафорах определены следующие вызовы:

1) Semget() – процесс может создать набор семафоров.

2) Semctl() -- изменение параметров

3) Semop() -- изменение состояния наборов семафоров

```
struct sembuf
{
        ushort sem_num;
        short sem_op;
        short sem_fl;
}
```

Sem_op > 0 // инкремент, освобождение ресурса

Sem_op = 0 // процесс, совершивший сисвызов, будет ожидать, но не пытается захватить семафор

Sem_op < 0 // декремент, захват ресурса

Sem_fl: на семафорах определены флаги:

1) IPC_NOWAIT – нежелание процесса переходить в состояние ожидания. Это во избежание блокировки всех процессов в очереди к семафору. В силу того, что сигнал kill нельзя перехватить, то убиваемый процесс не может выполнить освобождение семафоров, и соотв., ресурса, доступ к которому управляется семафору.
2) Sen_undo – указывает ядру, что следует отслеживать изменения семафора в результате сисвызова semop(). При завершении вызывающего процесса ядро ликвидирует сделанные изменения, чтобы процессы не были заблокированы на данном семафоре навечно. Например:

```
#include <sys/types.h>
#include <sys/ipc.h>
#include <sys/sem.h>
struct semid_ds sbuf[2]={{0, -1, sem_undo|ipc_nowait},
                         {1,0,1}};
int main()
{
        int perms = s_irwxv | s_irwxg | s_irwxo;
        int fd = semget(100,2,ipc_create|perms);
        if (fd == -1) {perror("semget); exit(1);}
        if (semop(fd,sbuf,2)==-1) perror ("semop");
        return 0;
}
```

В примере создается набор семафоров с идентификатором 100. Если сисвызов semget был завершен успешно, то нам удалось создать набор семафоров, потом вызывается semop, который декрементирует значение первого семафора.

Прототип semop:

```
Int semop(int semid, struct sembuf *opsPtr, int len);
```

Тупики – это ситуация, возникающая в результате монопольного использования разделяемых ресурсов, когда процесс, владея ресурсом, запрашивает другой ресурс, занятый непосредственно или через цепочку запросов другим процессом, который со своей стороны запрашивает и ожидает освобождения ресурса, занятого первым процессом.

Простейший пример: модель с двумя процессами p1 и p2 и двумя единичными ресурсами r1 и r2. Последовательность запросов выписывается следующим образом:

```
p1: …
    //запросил r1
    //получил r1
    …
    //запросил r2
p2: …
    //запросил r2
    //получил r2
    …
    //запросил r1
```

Процессы таким образом попадают в deadlock, не могут выполняться, так как пытаются взять ресурс, взятый другим.

Типы ресурсов

В теории тупиков выделяются два типа ресурсов

- повторно используемые
- потребляемые

К первым относятся аппаратная часть компьютера и реэнтерабельное ПО.

К вторым относятся любые сообщения.

Из самого названия следует: первые – используются многократно и это на них никак не влияет. Вторые – как булочка – съел и больше её нет. Получил сообщение – оно перестает существовать.

Повторно используемые ресурсы характеризуются следующими способами:

1. Число ресурсов единиц ограничено
2. Изменение числа единиц ресурса является исключением и следствием особых ситуаций в системе, например, выхода из строя внешнего устройства

Потребляемые ресурсы отличаются от повторно используемых в следующих отношениях:

1. Единиц ограниченное количество
2. Отправка неограниченна, а потребление приводит к уменьшению числа единиц.
3. Возможна система разной известности потребителей-производителей – от неограниченного количества тех и других до ограниченного.

Условие возникновения тупиков

Условия были сформулированы Ханвендером и являются ключом к борьбе с тупиками. Для возниковения тупиковой ситуации необходимо и достаточно 4 условия:

1. Взаимоисключение, mutual exclusion – когда процессы монопольно используют предоставляемые им ресурсы.
2. Ожидание, hold and wait – процессы удерживают занятые ресурсы, ожидая получения дополнительных ресурсов, которые им необходимы для продолжения выполнения.
3. Неперераздеяемость – ресурсы нельзя отобрать у процесса до его завершения или освобождения ресурса самим процессом.
4. Круговое ожидание – возникновение в замкнутой цепи запросов процессов, в которой каждый процесс занимает ресурс, необходимый следующему в цепочке процессу для продолжения выполнения.

Методы борьбы с тупиками

Существуют три основных подхода к проблеме борьбы с тупиками:

1. Исключение самой возможности возникновения тупиков
2. Недопущение или предотвращение или обход тупиков
3. Обнаружение и восстановление работоспособности системы

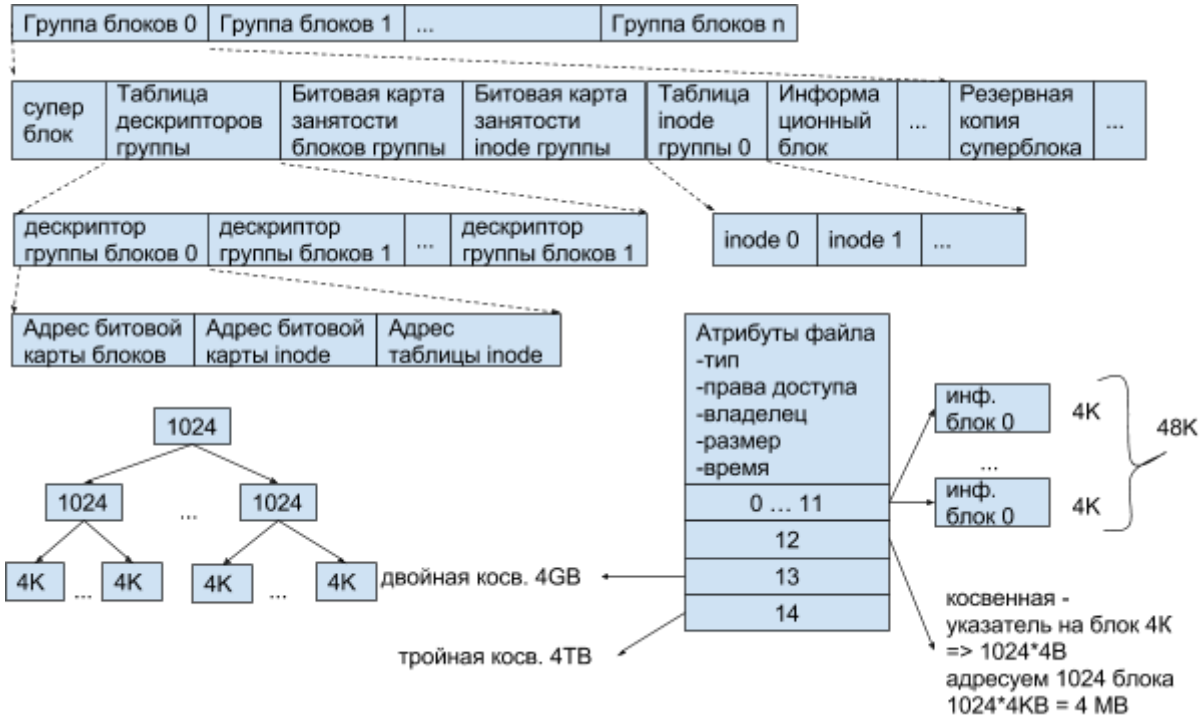
2. Файловые системы. Схема inode, схема суперблоков, связь между ними, особенности ФС в Linux.

Объект «суперблок» является начальным блоком любой ФС и содержит информацию, описывающую конкретную ФС. Суперблок хранится на диске в спецсекторе.

Struct super_blockuct

```
{
        list_head s_list;       //список суперблоков
        …
        struct super_operation s_op; // операции, определённые на суперблоке
        struct dentry *s_root; //точка монтирования
        …
        struct list_head s_inode; // список индексов
        struct list_head s_dirty; // список изменённых индексов
}
#define <fs/super.c>
```

Суперблок создается функцией alloc_super(), которая вызывается при монтировании ФС. Она заполняет поля структуры после считывания суперблока.




```
struct super_block
{
        struct list_head s_list;
        unsigned long s_blocksize;
        struct file_system_type *s_type;
        struct super_operations *s_op;
        struct semaphore s_lock;
        …
}
```

I-ноды.

Информация о файле часто называется метаданными. Иными словами, inode – метаданные. Система при доступе ищет точный inode в таблице inode. (...) В результате пользователь получает доступ к файлу по его inode, но пользователь указывает имя файла.

Для доступа к inode не требуется имя файла, но для того, чтобы по имени получить доступ, надо сопоставить имя с его inode. Посмотрим структуру айнод-каталога:

Inode number: 3470036	
. (DOT)	3470036
.. (DOT)	3470017
Folder1	3470031
File1	3470043
File2	3470023
Filder2	3470024
File3	3470065
...	

Структура inode-каталога состоит просто из имен к inode, соотв. этим файлам и поддиректориям.

Возможные допы

Пересчет динамических приоритетов

Планировщик Windows периодически настраивает текущий приоритет потоков, используя внутренний механизм повышения приоритета. Во многих случаях это делается для уменьшения различных задержек (чтобы потоки быстрее реагировали на события, в ожидании которых они находятся) и повышения их восприимчивости. Windows может динамически повышать значение текущего приоритета потока в одном из пяти случаев:

- Повышение вследствие событий исполнительной системы (сокращение задержек).
- Повышение вследствие завершения ввода-вывода (сокращение задержек).
- Повышение вследствие ввода из пользовательского интерфейса (сокращение задержек и времени отклика).
- Повышение вследствие окончания операции ожидания потоками активного процесса.
- Повышение в случае, если поток, готовый к выполнению, задерживается из-за нехватки процессорного времени (предотвращение зависания и смены приоритетов).

Но, как и любые другие алгоритмы планирования, эти настройки не совершенны, и они могут не принести пользы абсолютно всем приложениям.

Windows никогда не увеличивает приоритет потоков в диапазоне реального времени (от 16 до 31). Поэтому планирование таких потоков по отношению к другим всегда предсказуемо. Windows считает: тот, кто использует приоритеты реального времени, знает, что делает.

Повышение вследствие событий исполнительной системы

Когда ожидание потока на событии исполнительной системы или объекте «семафор» успешно завершается (из-за вызова SetEvent, PulseEvent или Relea-seSemaphore), его приоритет повышается на 1 уровень. Причина повышения приоритета потоков, закончивших ожидание событий или семафоров, та же, что и для потоков, ожидавших окончания операций ввода-вывода: потокам, блокируемым на событиях, процессорное время требуется реже, чем остальным. Такая регулировка позволяет равномернее распределять процессорное время.

В данном случае действуют те же правила динамического повышения приоритета, что и при завершении операций ввода-вывода.

К потокам, которые пробуждаются в результате установки события взовом специальных функций NtSetEventBoostPriority и KeSetEventBoostPriority повышение приоритета применяется особым образом. Если поток с приоритетом 13 или ниже, ждущий на событии, пробуждается в результате вызова специальной функции, его приоритет повышается до приоритета потока, установившего событие, плюс 1. Если длительность его кванта меньше 4 единиц, она приравнивается 4 единицам. Исходный приоритет восстанавливается по истечении этого кванта.

Повышение вследствие завершения операции ввода-вывода

Windows дает временное повышение приоритета при завершении определенных операций ввода-вывода, при этом потоки, ожидавшие ввода-вывода, имеют больше шансов сразу же запуститься и обработать то, чего они ожидали. Хотя рекомендуемые величины повышения можно найти в заголовочных файлах Windows Driver Kit (WDK), подходящее значение для увеличения зависит от драйвера устройства(см. таблицу). Именно драйвер устройства указывает повышение при завершении запроса на ввод-вывод в своем вызове функции ядра IoCompleteRequest. В таблице следует обратить внимание на то, что запросы ввода-вывода к устройствам, гарантирующим наилучшую отзывчивость, имеют более высокие значения повышения приоритета.

Таблица - Рекомендованные приращения приоритета

Устройство	Приращение
Жесткий диск, привод компакт-дисков, параллельный порт, видеоустройство	1
Сеть, почтовый слот, именованный канал, последовательный порт	2
Клавиатура, мышь	6
Звуковое устройство (плата)	8

Следует иметь в виду, что значения повышения, которые были даны в предыдущей таблице, это всего лишь рекомендации от компании Microsoft, разработчики драйверов могут при желании спокойно их проигнорировать.

Приоритет потока всегда повышается относительно базового, а не текущего уровня. Как показано на рисунке, после динамического повышения приоритета поток в течение одного кванта выполняется с повышенным уровнем приоритета, после чего приоритет снижается на один уровень и потоку выделяется еще один квант. Этот цикл продолжается до тех пор, пока приоритет не снизится до базового. Поток с более высоким приоритетом все равно может вытеснить поток с повышенным приоритетом, но прерванный поток должен полностью отработать свой квант с повышенным приоритетом до того, как этот приоритет начнет понижаться.

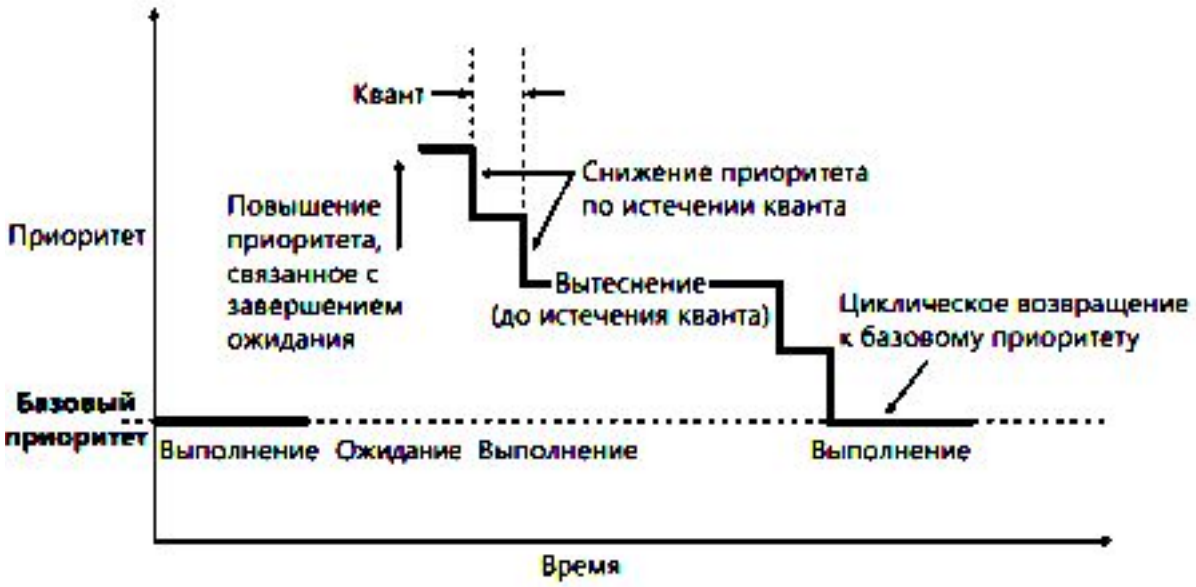


Рисунок - Динамическое повышение приоритета

Как уже отмечалось, динамическое повышение приоритета применяется только к потокам с приоритетом динамического диапазона (0-15). Независимо от приращения приоритет потока никогда не будет больше 15. Иначе говоря, если к потоку с приоритетом 14 применить динамическое повышение на 5 уровней, его приоритет возрастает лишь до 15. Если приоритет потока равен 15, он остается неизменным при любой попытке его повышения.

Повышение вследствие ввода из пользовательского интерфейса

Потоки-владельцы окон получают при пробуждении дополнительное повышение приоритета на 2 из-за активности системы работы с окнами, например поступление сообщений от окна. Система работы с окнами (Win32k.sys) применяет это повышение приоритета, когда вызывает функцию KeSetEvent для установки события, используемого для пробуждения GUI-потока. Смысл этого повышения — содействие интерактивным приложениям.

Повышение вследствие окончания операции ожидания потоками активного процесса

Всякий раз, когда поток в активном процессе завершает ожидание на объекте ядра, функция ядра KiUnwaitThread динамически повышает его текущий (не базовый) приоритет на величину текущего значения PsPrioritySeparation. (Какой процесс является активным, определяет подсистема управления окнами.).

Это увеличивает отзывчивость интерактивного приложения по окончании ожидания. В результате повышаются шансы на немедленное возобновление его потока — особенно если в фоновом режиме выполняется несколько процессов с тем же базовым приоритетом.

Повышение в случае, если поток, готовый к выполнению, задерживается из-за нехватки процессорного времени

Раз в секунду диспетчер настройки баланса (balance set manager) (системный поток, предназначенный главным образом для выполнения функций управления памятью) сканирует очереди готовых потоков и ищет потоки, которые находятся в состоянии

Ready в течение примерно 4 секунд. Обнаружив такой поток, диспетчер настройки баланса повышает его приоритет до 15. В Windows 2000 и Windows XP квант потока удваивается относительно кванта процесса. В Windows Server 2003 квант устанавливается равным 4 единицам. Как только квант истекает, приоритет потока немедленно снижается до исходного уровня. Если этот поток не успел закончить свою работу и если другой поток с более высоким приоритетом готов к выполнению, то после снижения приоритета он возвращается в очередь готовых потоков. В итоге через 4 секунды его приоритет может быть снова повышен.

Чтобы свести к минимуму расход процессорного времени, диспетчер настройки баланса сканирует лишь 16 готовых потоков. Если таких потоков с данным уровнем приоритета более 16, он запоминает тот поток, перед которым он остановился, и в следующий раз продолжает сканирование именно с него. Кроме того, он повышает приоритет не более чем у 10 потоков за один проход. Обнаружив 10 потоков, приоритет которых следует повысить (что говорит о необычайно высокой загруженности системы), он прекращает сканирование. При следующем проходе сканирование возобновляется с того места, где оно было прервано в прошлый раз.

Unix

Приоритет процесса задается любым целым числом, лежащим в диапазоне от 0 до 127. Чем меньше такое число, тем выше приоритет. Приоритеты от 0 до 49 зарезервированы для ядра, следовательно, прикладные процессы могут обладать приоритетом в диапазоне 50-127. Структура proc содержит следующие поля, относящиеся к приоритетам:

p_pri	Текущий приоритет планирования
p_usrpri	Приоритет режима задачи
p_cpu	Результат последнего измерения использования процессора
p_nice	Фактор «любезности», устанавливаемый пользователем

Поля p_pri и p_usrpri применяются для различных целей. Планировщик использует p_pri для принятия решения о том, какой процесс направить на выполнение. Когда процесс находится в режиме задачи, значение его p_pri идентично p_usrpri. Когда процесс просыпается после блокирования в системном вызове, его приоритет будет временно повышен для того, чтобы дать ему предпочтение для выполнения в режиме ядра. Следовательно, планировщик использует p_usrpri для хранения приоритета, который будет назначен процессу при возврате в режим задачи, а p_pri — для хранения временного приоритета для выполнения в режиме ядра.

Ядро системы связывает приоритет сна с событием или ожидаемым ресурсом, из-за которого процесс может заблокироваться. Приоритет сна является величиной, определяемой для ядра, и потому лежит в диапазоне 0-49. Когда замороженный процесс просыпается, ядро устанавливает значение его p_pri, равное приоритету сна события или ресурса, на котором он был заблокирован. Поскольку приоритеты ядра выше, чем приоритеты режима задачи, такие процессы будут назначены на выполнение раньше, чем другие, функционирующие в режиме задачи. Такой подход

позволяет системным вызовам быстро завершать своб работу, что является желательным, так как процессы во время выполнения вызова могут занимать некоторые ключевые ресурсы системы, не позволяя пользоваться ими другим.

Когда процесс завершил выполнение системного вызова и находится в состоянии возврата в режим задачи, его приоритет сбрасывается обратно в значение текущего приоритета в режиме задачи. Измененный таким образом приоритет может оказаться ниже, чем приоритет какого-либо иного запущенного процесса; в этом случае ядро системы произведет переключение контекста.

Приоритет в режиме задачи зависит от двух факторов: «любезности» (nice) и последней измеренной величины использования процессора. Степень любезности (nice value) является числом в диапазоне от 0 до 39 со значением 20 по умолчанию. Увеличение значения приводит к уменьшению приоритета. Фоновым процессам автоматически задаются более высокие значения. Уменьшить эту величину для какого-либо процесса может только суперпользователь, поскольку при этом поднимется его приоритет. Степень любезности называется так потому, что одни пользователи могут быть поставлены в более выгодные условия другими пользователями посредством увеличения кем-либо из последних значения уровня любезности для своих менее важных процессов.

Системы разделения времени пытаются выделить процессорное время таким образом, чтобы конкурирующие процессы получили его примерно в равных количествах. Такой подход требует слежения за использованием процессора каждым из процессов. Поле p_cpu структуры proc содержит величину результата последнего сделанного измерения использования процессора процессом. При создании процесса значение этого поля инициализируется нулем. На каждом тике обработчик таймера увеличивает p_cpu на единицу для текущего процесса до максимального значения, равного 127. Более того, каждую секунду ядро системы вызывает процедуру schedcpu() (запускаемую через отложенный вызов), которая уменьшает значение p_cpu каждого процесса исходя из фактора «полураспада» (decay factor). В системе SVR3 используется фиксированное значение этого фактора, равное 1/2. В 4.3BSD для расчета фактора полураспада применяется следующая формула:

decay = (2*load_average)/(2*load_average + 1)

где load_average — это среднее количество процессов, находящихся в состоянии готовности к выполнению, за последнюю секунду.

Процедура schedcpu() также пересчитывает приоритеты для режима задачи всех процессов по формуле

p_usrpri = PUSER + (p_cpu/4) + (2*p_nice)

где PUSER — базовый приоритет в режиме задачи, равный 50.

В результате, если процесс в последний раз, т.е. до вытеснения другим процессом, использовал большое количество процессорного времени, его p_cpu будет увеличен. Это приведет к росту значения p_usrpri и, следовательно, к понижению приоритета. Чем дольше процесс простаивает в очереди на выполнение, тем больше фактор полураспада уменьшает его p_cpu, что приводит к повышению его приоритета. Такая схема предотвращает бесконечное откладывание низкоприоритетных процессов. Ее применения предпочтительно процессам, осуществляющим много операций ввода-вывода, в противоположность процессам, производящим много вычислений.

Что такое система разделения времени?

Такие системы обеспечивают одновременное обслуживание многих пользователей, позволяя каждому пользователю взаимодействовать со своим заданием в режиме диалога. Эффект одновременного обслуживания достигается разделением процессорного времени и других ресурсов между несколькими вычислительными процессами, которые соответствуют отдельным заданиям пользователей. Операционная система предоставляет ЭВМ каждому вычислительному процессу в течение небольшого интервала времени; если вычислительный процесс не завершился к концу очередного интервала, он прерывается и помещается в очередь ожидания, уступая ЭВМ другому вычислительному процессу. ЭВМ в этих системах функционирует в мультипрограммном режиме.

Операционная система разделения времени может применяться не только для обслуживания пользователей, но и для управления технологическим оборудованием. В этом случае “пользователями” являются отдельные блоки управления исполнительными устройствами, входящими в состав технологического оборудования: каждый блок взаимодействует с определенным вычислительным процессом в течение интервала времени, достаточного для передачи управляющих воздействий на исполнительное устройство или приёма информации от датчиков.

Почему в системе может быть только одна таблица процессов?

Ответ в гугле не нашел, но предположительно это, отчасти, из-за ограничения на максимальное количество возможных процессов. Так же, возможно, это сделано для того, чтобы не было путаницы во время планирования процессов, т.к. они все имеют уникальный PID. Основная догадка - проблема организации общения между процессами, т.к. если они будут в разных таблицах, то они будут изолированы друг от друга.

Почему надо иметь стек ядра и стек пользователя?

Потому что мы с вами говорили, когда рассматривали диаграмму состояний: часть времени процесс выполняется в режиме пользователя (задачи), и тогда процесс выполняет собственный код. А часть времени процесс выполняет реентерабельный код ОС. Когда процесс переключается в режим ядра, с ним происходят всякие метаморфозы, при этом он запрашивает дополнительные ресурсы – например, запрашиваются устройства ввода-вывода. После этого возникает аппаратное прерывания и вызывается соответствующий обработчик прерывания.

Адресное пространство: что значит «создать адресное пространство»?

Это значит «создать для данного процесса таблицы, описывающие данное адресное пространство». В случае со страничной виртуальной памятью – это будет таблица страниц, в которой будет храниться дескрипторы страниц. Каждый такой дескриптор содержит информацию об одной странице, и они описывают виртуальное пространство, так как для процессов выделяется виртуальное адресное пространство.

Что такое стек ядра в контексте ядра? Стек пользователя?

Стек ядра, в котором хранятся записи процедур ядра, если процесс выполняется в режиме ядра. Несмотря на то, что все процессы пользуются одними и теми же программами ядра, каждый из них имеет свою собственную копию стека ядра для хранения индивидуальных обращений к функциям ядра. Пусть, например, один процесс вызывает функцию create и приостанавливается в ожидании назначения нового индекса, а другой процесс вызывает функцию read и приостанавливается в ожидании завершения передачи данных с диска в память. Оба процесса обращаются к функциям ядра и у каждого из них имеется в наличии отдельный стек, в котором хранится последовательность выполненных обращений. Ядро должно иметь возможность восстанавливать содержимое стека ядра и положение указателя вершины стека для того, чтобы возобновить выполнение процесса в режиме ядра. В различных системах стек ядра часто располагается в пространстве процесса, однако этот стек является логически-независимым и, таким образом, может помещаться в самостоятельной области памяти. Когда процесс выполняется в режиме задачи, соответствующий ему стек ядра пуст.

Стек пользователя видимо необходим для хранения значений автоматических переменных и параметров функций, а также для организации рекурсивных вызовов функций.

Отличия семафоров от мьютиксов

1. У мьютикса есть владелец и это процесс, который захватил мьютикс. Только процесс, захвативший мьютикс, может его разблокировать, в отличие от мьютиксов, семафоры такого владельца не имеют, и разблокировать семафор может другой процесс.

P1: … … … P(S) ...	P1: … … … V(S) ...

2. Для обеспечения нормальной работы процесс, удерживающий мьютикс, не может быть случайно, в отличие от семафора, удален.
3. Для мьютиксов характерна инверсия приоритетов. Учитывается приоритет владельца мьютикса и это связано с тем, когда какая-либо высокоприоритетная задача переходит в состояние ожидания мьютикса.

Цикл запроса на ввод/вывод

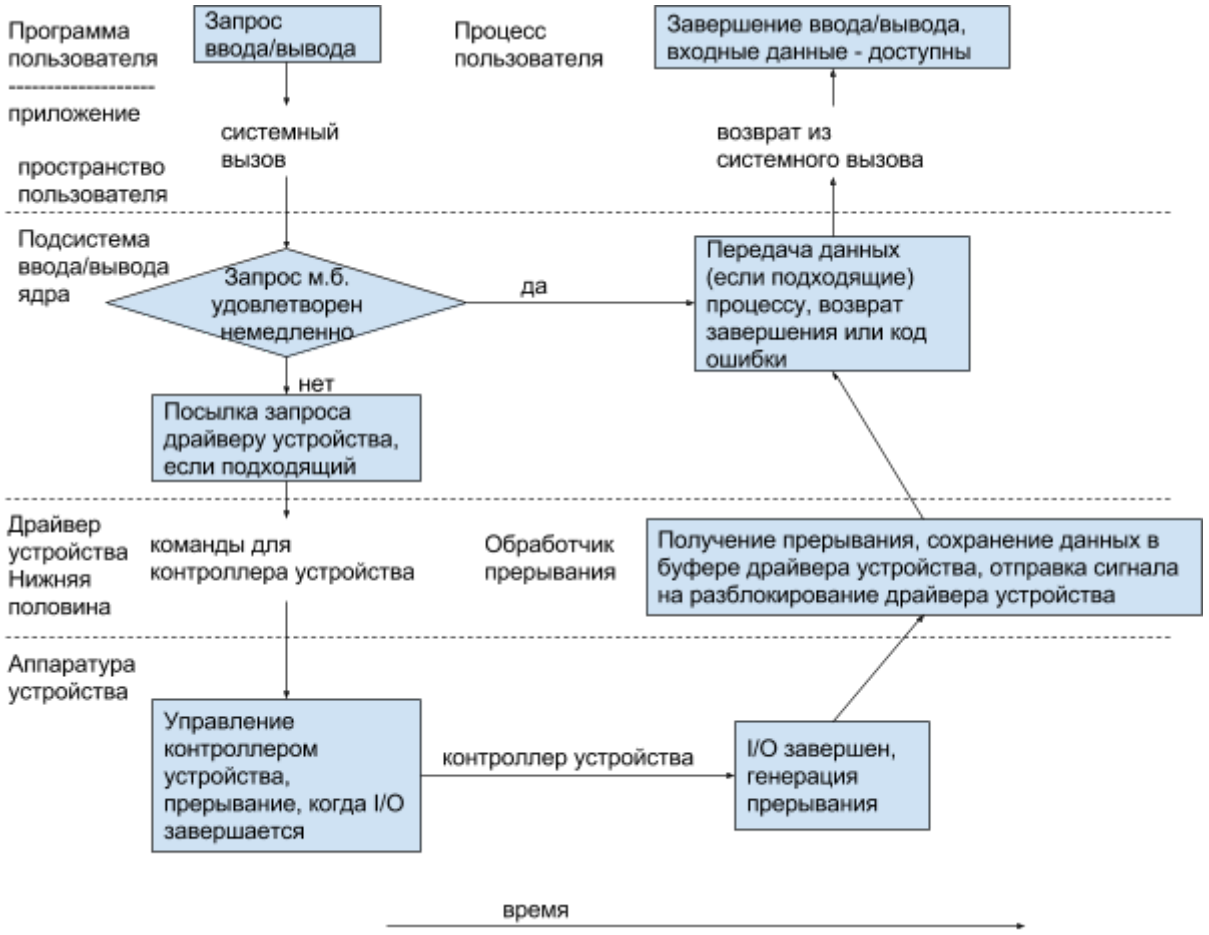
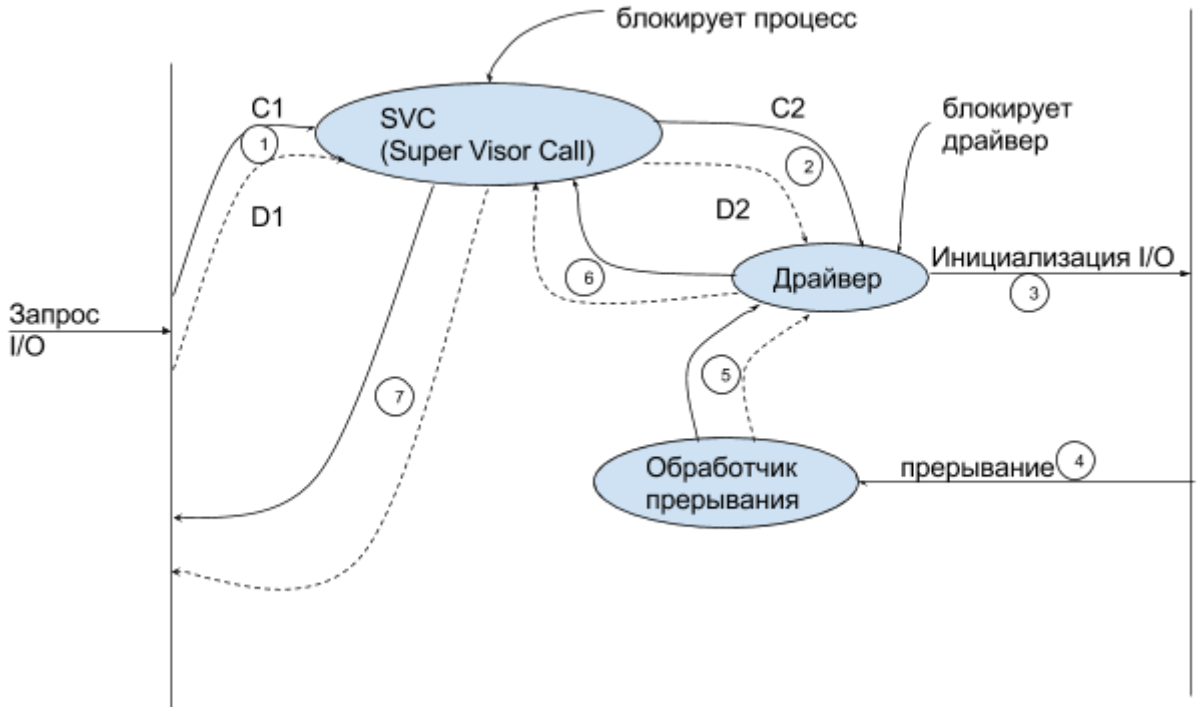


Диаграмма Шоу

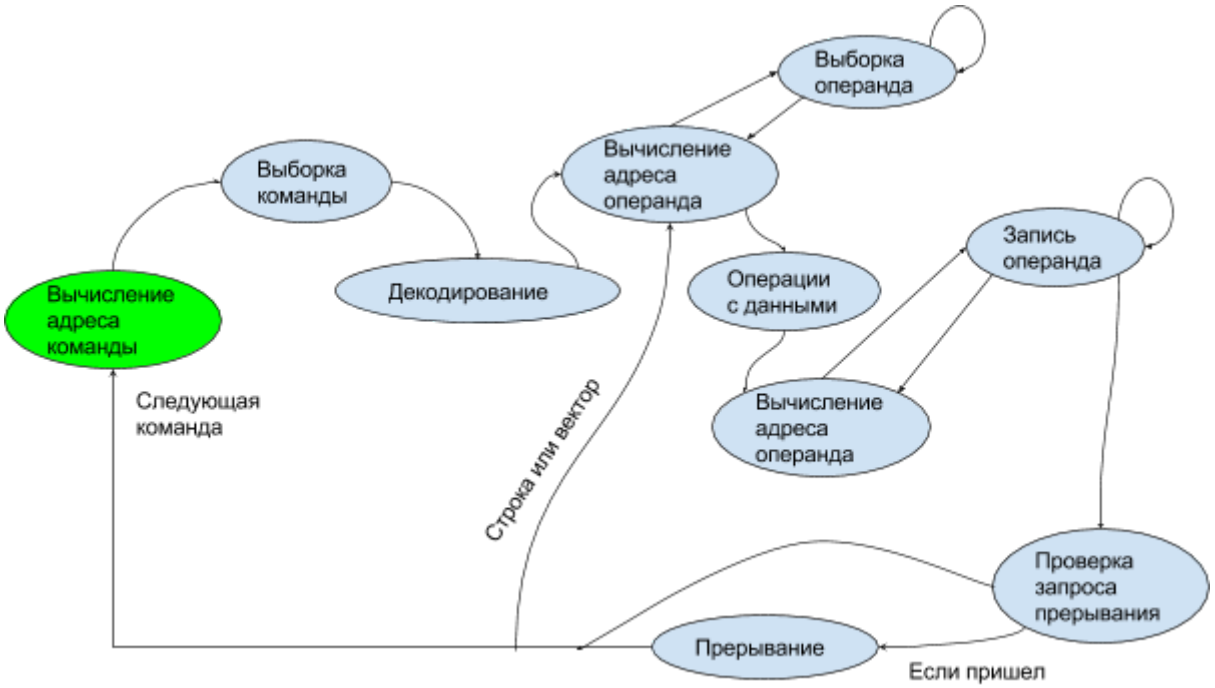


Методы выхода из тупика

Восстановление можно интерпретировать как запрет постоянного пребывания в опасном состоянии. Существуют следующие методы восстановления:

- принудительный перезапуск системы, характеризующийся потерей информации обо всех процессах, существовавших до перезапуска
- принудительное завершение процессов, находящихся в тупике
- принудительное последовательное завершение процессов, находящихся в тупике, с последующим вызовом алгоритма распознавания после каждого завершения до исчезновения тупика
- перезапуск процессов, находящихся в тупике, с некоторой контрольной точки, то есть из состояния, предшествовавшего запросу на ресурс
- перераспределение ресурсов с последующим перезапуском процессов, находящихся в тупике

Диаграмма состояния цикла выполнения команды



Режимы ядра и задачи переключения в режим ядра - классификация и события.

Ядра делятся на монолитные и микроядра.

Все ОС общего назначения – с монолитным ядром.

Монолитное ядро – программа, имеющая модульную структуру, т.е. состоящая из подпрограмм. Современные ядра – многопоточные, но это не значит, что они перестали быть монолитными. Любой запущенный драйвер – отдельный поток. В драйвере мы также можем дополнительный поток запустить. Так как монолитное ядро является единой программой, то единственным способом изменения кода ядра является его перекомпиляция.

Микроядро -- ядро, в котором оставлены основными действиями организация взаимодействия между программами, планирование использования процессора, первичную обработку прерываний, операции ввода-вывода и базовое управление памятью. Все остальные команды выполняются в ОС в unprivileged mode. В этом случае процессы ОС работают по модели Клиент-Сервер, и взаимодействуют при помощи сообщений.

Пример монитора “читатели - писатели” на мьютексах (семафорах по сути)

```
#include "stdafx.h"
#include <stdio.h>
#include <windows.h>
#include <time.h>
#pragma hdrstop
#define COUNT_READERS 4
#define COUNT_WRITERS 4
#define TIME_READING 2000
#define TIME_WRITING 3000

```

```
long wr = 0;  // Waiting Reader
long ww = 0;  // Waiting Writer
long ar = 0;  // Active Reader
long aw = 1;  // Active Writer

HANDLE mutex, read, write;
int buffer = 0;

void startW()
{
	InterlockedIncrement (&ww);
	if (aw)
	   WaitForSingleObject (write, INFINITE);
	WaitForSingleObject (mutex, INFINITE);  // ожидание освобождение
                                                // мьютекса, затем lock()
	SetEvent (write); // Перевести событие в свободное состояние
	InterlockedDecrement (&ww);
	aw = 1;
	ReleaseMutex (mutex); //unlock()
}

void stopW()
{
	aw = 0;

	if (wr)
	   SetEvent(read); // Перевести событие в свободное состояние
	else
	{
	   ResetEvent (read); // Перевести событие в занятое состояние
	   SetEvent (write); // Перевести событие в свободное состояние
	}
}
// запись
unsigned long int writer ()
{
   srand (time (NULL));
   for (int i = 0; i < 5; i++)
   {
      startW();
      buffer++;
      HANDLE hConsole = GetStdHandle(STD_OUTPUT_HANDLE);
      SetConsoleTextAttribute(hConsole, (WORD) ((0 << 4) | 2));
      printf ("WRITER [%ld] write: %ld\n", GetCurrentThreadId (), buffer);
      SetConsoleTextAttribute(hConsole, (WORD) ((0 << 4) | 9));
      stopW();

      Sleep (rand () % TIME_WRITING);
   }
   return 0;
}


void startR()
```

```
65. {
66.       InterlockedIncrement (&wr);
67.       if (aw || ww)
68.          WaitForSingleObject (read, INFINITE);
69.
70.       WaitForSingleObject (mutex, INFINITE); // ожидание освобождение
71.                                              // мьютекса, затем lock()
72.       InterlockedDecrement (&wr);
73.       InterlockedIncrement (&ar);
74.       SetEvent (read);  // Перевести событие в свободное состояние
75.       ReleaseMutex (mutex);  // unlock()
76. }
77.
78. void stopR()
79. {
80.       InterlockedDecrement (&ar);
81.       if (!wr)
82.          SetEvent (write); // Перевести событие в свободное состояние
83.       else
84.       {
85.          ResetEvent (write);  // Перевести событие в занятое состояние
86.          SetEvent (read); // Перевести событие в свободное состояние
87.       }
88. }
89.
90. // чтение
91. unsigned long int reader ()
92. {
93.    srand (time (NULL));
94.    while (buffer <  20)
95.    {
96.       Sleep (rand () % TIME_READING);
97.       startR();
98.       printf ("    READER [%ld] read: %ld\n", GetCurrentThreadId (), buffer);
99.       stopR();
100.          Sleep (rand () % TIME_READING);
101.       }
102.       return 0;
103.    }
104.
105.    int main (int argc, _TCHAR* argv[])
106.    {
107.       HANDLE write_threads[COUNT_WRITERS];
108.       HANDLE read_threads[COUNT_READERS];
109.
110.       //  Создание мьютекса
111.       mutex = CreateMutex (NULL, FALSE, NULL);
112.       //  Создание объектов - событий
113.       read = CreateEvent (NULL, TRUE, FALSE, NULL);
114.       write = CreateEvent (NULL, TRUE, TRUE, NULL);
115.
116.       // чтение
117.       for (int i = 0; i < COUNT_READERS; i++)
118.       {
```

```
119.          read_threads[i] = CreateThread (NULL, 0, (LPTHREAD_START_ROUTINE)&reader,
    NULL, 0, NULL); // создание потока
120.          Sleep(rand () % TIME_READING);
121.       }
122.
123.       // запись
124.       for (int i = 0; i < COUNT_WRITERS; i++)
125.       {
126.          write_threads[i] = CreateThread (NULL, 0, (LPTHREAD_START_ROUTINE)&writer,
127.                                           NULL, 0, NULL);  // создание потока
128.          Sleep(rand () % TIME_WRITING);
129.       }
130.
131.       // Ожидание завершения потоков
132.       WaitForMultipleObjects (COUNT_WRITERS, write_threads, TRUE, INFINITE);
133.       WaitForMultipleObjects (COUNT_READERS, read_threads, TRUE, INFINITE);
134.
135.       // Закрытие декскрипторов объектов
136.       CloseHandle (mutex);
137.       CloseHandle (read);
138.       CloseHandle (write);
139.
140.       system("pause");
141.       return 0;
142.    }
```