

Билеты по ОСям

[Вместо предисловия](#)

[Полезные ссылки](#)

[Дополнительно](#)

[ГЛОССАРИ](#)

[CISC- и RISC-архитектуры](#)

[Transputer](#)

[Вопросы на консультацию](#)

Билет 1

Вопрос 1: Как происходит загрузка операционной системы? Что такое первичный загрузчик? Вторичный? Как происходит загрузка бездисковых машин?

Вопрос 2: *организация файловой системы HPFS*

[Монтирование ФС](#)

[Операции над файлом](#)

[RT-11 Иртегов лекция 16 48 минута](#)

[ISO-9660 \(CDFS\) Иртегов 16 лекция 53 минута](#)

[FAT Иртегов лекция 16 57 минута](#)

[Сложные ФС лекция 16 108 минута](#)

[HPFS Иртегов лекция 16 118 20](#)

[FFS \(UFS\)](#)

[Данные в иноде](#)

[NTFS Иртегов 16 лекция 1 16](#)

Билет 2

Вопрос 1: Распределение памяти алгоритмами близнецов и парных меток. Ограничения этих алгоритмов.

Вопрос 2: Аутентификация и проверка подлинности кода в Apple iOS.

[Загрузка iOS](#)

[Обязательная подпись кода](#)

[Процесс подписания приложений разработчиками](#)

[Аутентификация](#)

[Списки контроля доступа](#)

Билет 3

Вопрос 1: Алгоритмы поиска жертвы при страничном обмене и кэшировании. Критерии выбора и влияние алгоритма на производительность. Что такое рабочее множество страниц?

Вопрос 2: Сигналы в системах семейства Unix.

[Функция обработчик](#)

Билет 4

Вопрос 1: Инверсия приоритета. Способы ее предотвращения и способы обхода этой проблемы.

[Способы ее предотвращения и способы обхода этой проблемы](#)

Вопрос 2: Линки в транспьютере

Билет 5

Вопрос 1: Определение задачи реального времени. Чем системы РВ отличаются от систем разделенного времени? Пример архитектуры ОС реального времени.

[Отличия от Систем Разделенного Времени \(by Inkpv\):](#)

[Другое описание отличия систем \(перевод\)](#)

[Системы реального времени](#)

[Архитектура](#)

Вопрос 2: Сборщик мусора Java HotSpot

Билет 6

Вопрос 1: Журнальные файловые системы. Принципы работы. Для чего это нужно?

Вопрос 2: Почтовые ящики (mailbox) в VAX/VMS

Билет 7

Вопрос 1: Семафоры Дийкстры. Мутексы, двоичные семафоры и семафоры общего вида. Мертвая блокировка и способы избежать ее.

[Мёртвые и живые блокировки](#)

[Примитивы синхронизации с ожиданием](#)

Семафоры и мутексы

Флаги событий RSX-11 и VMS (AST)

Блокировка чтения-записи

Копирование при записи

Захват участков файлов

Мониторы и серверы транзакций

Вопрос 2: Файловая система NetApp WAFL

Билет 8

Вопрос 1: Как реализуется многопоточность на однопроцессорной машине.

Что такое контекст процесса? Какие особенности процессора влияют на скорость переключения процессов?

Какие особенности процессора влияют на скорость переключения процессов?

Вопрос 2: Формирование запросов на ввод/вывод в RSX-11, VMS, OpenVMS.

Какие преимущества предоставляет этот метод?

Преимущества

Билет 9

Вопрос 1: Что такое гармонически взаимодействующие последовательные процессы? Средства для реализации этой дисциплины в существующих системах.

Программные каналы Unix (трубы)

Почтовые ящики VMS

Линки в транспьютере

Системы, управляемые событиями

Вопрос 2: Организация страничного обмена в VMS, OpenVMS и Windows NT

Поиск жертвы

Поиск жертвы в VAX/VMS и Windows NT/2000/XP

LRU и clock-алгоритм поиска жертвы

Управление swap-файлом

Билет 10

Вопрос 1: Методы реализации виртуальной памяти. Базовая адресация, сегментная и страничная виртуальная память

Базовая адресация

Сегментная и страничная адресация

Вопрос 2: Программные каналы (трубы) в системах семейства Unix

Билет 11

Вопрос 1: Что такое абсолютный и относительный загрузчики? Структура абсолютного и перемещаемого загрузочных модулей.
Что такое позиционно-независимый код?

Запуск задач в ОС семейства Unix

Абсолютная загрузка

Относительная загрузка

Позиционно-независимый код

Вопрос 2: Семафоры Unix System V IPC. Наборы семафоров.

Виды семафоров:

Семафоры System V - кратко

Семафоры System V - подробно

Билет 12

Вопрос 1: Устойчивые к сбоям файловые системы. Методы реализации устойчивых ФС.

Устойчивость к сбоям системы

Журнализование и журнальные ФС

JFS как пример журнальной ФС (многие моменты справедливы и для NTFS)

Устойчивость к сбоям диска

ФС с копированием при записи (NetApp WAFL)

Вопрос 2: Диспетчер задач в транспьютере.

Билет 13

Вопрос 1: Сборка в момент загрузки. Преимущества и недостатки этого метода. Чем отличаются DLL Win32 и разделяемые библиотеки ELF

Сборка программ

Объектные библиотеки

Сборка в момент загрузки

Динамические библиотеки

Вопрос 2: Динамическое выделение памяти в ОС семейства Unix и стандарте POSIX

Билет 14

Вопрос 1: Драйвер устройства. Функции драйвера в ОС семейства Unix.

Функции драйверов

Архитектура драйвера

Запрос к драйверу

Синхронный ввод-вывод

Асинхронный ввод-вывод

Вопрос 2: Файловая система FAT.

Билет 15

Вопрос 1: Динамическое выделение памяти. Методы борьбы с фрагментацией. Основные алгоритмы выделения памяти

Выделение памяти для систем с открытой памятью

Алгоритм парных меток

Алгоритм близнецов

Слабовые аллокаторы

Вопрос 2: Флаги событий RSX и VMS. Что такое AST?

Билет 16

Вопрос 1: Мертвая и живая блокировки. Способы их предотвращения. Преимущества и недостатки каждого из методов

Вопрос 2: Разделяемые библиотеки формата ELF

О разделяемых библиотеках Win32/Win64 (PE)

Наконец, разделяемые библиотеки формата ELF

Билет 17

1. Разделяемая память. Преимущества и недостатки по сравнению с другими методами межпроцессного взаимодействия.

2. Механизм setuid в ОС семейства Unix.

Общее про setuid

sudo в современных Unix-системах

Полезные источники

Билет 18

Вопрос 1: Событийно-ориентированные системы. Обязательно ли такая система является многопоточной?

Вопрос 2: Понятия инода и связи в файловых системах ОС семейства Unix.

Билет 19

Вопрос 1: Реентерабельная программа. Техника реализации реентерабельных программ. Всегда ли это возможно? Что такое критическая секция?

Вопрос 2: Загружаемые модули и разделяемые библиотеки Win32/Win64 (PE).

Билет 20

Вопрос 1: Прерывания в классических процессорах (PDP-11, 8086, x86). Внешние прерывания и исключения (exceptions).

Опрос

Канальные процессоры и прямой доступ к памяти

Прерывания

Классический подход к прерываниям на примере PDP-11

Исключения

Вопрос 2: Сборщик мусора Java G1.

Отличия малой сборки

Отличия большой сборки

Билет 21

Вопрос 1: Объектный модуль. Объектная библиотека. Структуры данных, содержащиеся в объектном модуле, в общих чертах.

Алгоритм работы сборщика и выбора модулей из архивной библиотеки.

Вопрос 2: Структура и принципы работы файловой системы NTFS.

Билет 22

Вопрос 1: Приоритеты процессов и нитей. Управление приоритетами для нитей реального и разделенного времени. Где используется и для чего нужно динамическое изменение приоритета?

Вопрос 2: Права доступа к файлам в ОС семейства Unix.

Билет 23

Вопрос 1: Системы управления доступом. Полномочия и списки контроля доступа. Кольца доступа.

Системы управления доступом

Списки контроля доступа Иртегов с 773

Полномочия Иртегов с 783

Билет 24.

Вопрос 1: Планировщики разделенного времени. Динамическое управление приоритетами в системах разделенного времени.

Вопрос 2: Структура и особенности организации файловой системы UFS (FFS).

Билет 25

Вопрос 1: Кооперативная и вытесняющая (preemptive) многозадачность. Преимущества и недостатки обоих архитектур.

Кооперативная многозадачность, преимущества и недостатки.

Вытесняющая многозадачность

Вопрос 2: Файловая система ISO 9660 (CDFS).

Билет 26

Вопрос 1: Троянские программы и способы их внедрения. Меры по защите от троянских программ.

Способы их внедрения

Описание и защита

ЗАЩИТА

Вопрос 2: Асинхронный ввод-вывод в стандарте POSIX.

Билет 27

Вопрос 1: Сборка мусора. Основные стратегии сборки мусора, их преимущества и недостатки

Основные стратегии сборки мусора:

(Иртегов 4.3.1 с. 255) Подсчет ссылок

(Иртегов 4.3.2 с. 256) Просмотр ссылок (mark and sweep)

(Иртегов 4.3.3 с. 259) Генерационная сборка мусора.

Репликационный или инкрементальный сборщик мусора(Иртегов с. 264).

Билет 28

Вопрос 1: Ввод-вывод в режиме опроса и по прерываниям. Преимущества и недостатки.

Прерывания

Вопрос 2: Уровни RAID.

Билет 29

Вопрос 1: Спинлоки и их применение. Их преимущества и недостатки по сравнению с другими средствами взаимоисключения.

Их преимущества и недостатки по сравнению с другими средствами взаимоисключения.

Вопрос 2: Структура файловой системы RT-11.

Вместо предисловия

азазазазазазаза, жесть вы лохи, надо было криты сдавать. Выпью за ваше здоровье, не чокаясь.



Ну а если серьёзно, то вот пара важных моментов, которые лучше учитывать с самого начала курса.

Во-первых, эта ПДФка не спасёт вас, если вы решили за день до экза, что пойдёте в полноценный бой против Папы ФИТА - нужно готовиться хотя бы за 10 дней, но даже это успеха не гарантирует. Оценки писавших это всё распределились так: 3 за криты (4 за семестр, избрал лёгкий путь... можем понять, но не простить), 3 за билет (4 за семестр), 4 за билет (4 за семестр) и 5 за билет (5 за семестр)

Статистика маловата, но мне кажется, можно сделать из неё вывод, что работа в семестре определённо играет роль. По крайней мере, получив 5 за семестр, прохождение экзамена может стать сильно проще.

Если Папа к нашим отзывам прислушается, получение пятёрки за лабы может стать чуть легче и/или приятнее, но в текущих реалиях выигрышной будет стратегия залутать по лабам тройку, а потом пилить шелл (автор вообще не ручается, что у вас получится, так что есть риск остаться с тройкой, что, конечно, не приговор, но сильно грустнее сделает положение).

Написать что-нибудь о тактике с лутанием четвёрки за семак

Как сдавать лабы? Ну... Пока пишешь код, придерживайтесь рекомендаций по его виду от самого Иртегова и НЕ ИСПОЛЬЗОВАТЬ ТО, БЕЗ ЧЕГО МОЖНО ОБОЙТИСЬ (а то вопрос ограбёте по самое горло). Читать маны по всему, что используете. Готовиться ко всем возможным вопросам по теме лабы на основании лекций (правда на практике они безумно запаздывают от необходимого темпа сдачи лаб), методички Иртегова, его книги, а также других источников (нашей группе очень помогала книга Вахалии "UNIX изнутри")

Всё сказанное выше справедливо, если ваш семинарист - не Иртегов. Если же вам так "повезло", то...

Некоторые советы по сдаче лаб Иртегову.

Данный текст не является исчерпывающим гайдом на оценку 5, 4 или 3. Это всего лишь рекомендации, они не дают никаких гарантий.

В наш год была введена система учёта задач при использовании github. Инструкция Иртегова имеет свои проблемы, в частности скорее всего после её прочтения вы и половины слов не поймёте.

Нормальную инструкцию мы не написали(или мне о ней не известно), но есть пару документов, которые возможно вам немного помогут:

[https://vk.com/doc341849136_672561266?](https://vk.com/doc341849136_672561266?hash=tz29TIPqMhWofWb993JQy65CW2ulzZE9pBJUVwpaU3T&dl=5hgd2btYFyUIHR6SRyuVvo80hl1rRODDiV1TPA0le3k)

[https://vk.com/doc341849136_672561271?](https://vk.com/doc341849136_672561271?hash=laMvZTILF7zy1b9DcmnPm31vd8K704pTsXaXZkeXXlw&dl=2bVdS1uOq1uzvmZ32O1eEDwE8Fh5PPnAkVUnLUG3TXc)

https://vk.com/doc341849136_672561271?hash=laMvZTILF7zy1b9DcmnPm31vd8K704pTsXaXZkeXXlw&dl=2bVdS1uOq1uzvmZ32O1eEDwE8Fh5PPnAkVUnLUG3TXc

За правдивость того что внутри документов я опять же не ручаюсь, но меня второй документ выручил.

Об отсутствии нормального документа по гит системе Иртегова деканату было донесено, но изменится ли что-либо это уже вы узнаете.

Другие семинаристы имеют другие требования к гиту. И то что работает у одного семинариста, с большой вероятностью Иртегов не примет.

Без настроенного гита, Иртегов отказывается принимать задачи. Так что гит это первоначальная вещь с которой вам нужно разобраться.

"Твои знания не всегда пропорциональны твоей оценке"

Примерно к такой мысли придёт большинство людей после курса ОСей. Стоит себя сразу готовить к тому, что за ОСи 3 это не такая уж и плохая оценка.

Я думаю кто-то из читающих сейчас усмехнулся, но попомните моё слово в конце 3-го семестра))).

Ну собственно по сдачам задач. Если вы в группе Иртегова, то приплыли.

Привыкайте к тому, что однокурсники будут сдавать по 3 лабы за семинар, пока вы сдаёте 1 лабу 32 семинара. Будет тяжело получить даже 3, не говоря уж о чём-то большем. Собственно этот док и пишется, для облегчения жизни 2 курсу.

Оценка 5 за лаб. работы. На 5 надо сдать 23 лабы или написать shell.

Сдать 23 лабы идея гиблая, ибо вы будете сдавать 1 лабу в семинар при удачном стечении обстоятельств, а семинаров 16.

Можете попробовать, я не то чтобы отговариваю, но за всё время существования нового потока ни один человек

не сдал Иртегову 23 лабы.

Shell - если вы собираетесь писать только shell, не сдавая задач, то первую часть shell'a нужно сдать до первой зачетной недели,

если вы собираетесь набрать сначала задач на 3 или 4, то shell можно начать сдавать в любой момент.

Оценка 4 и 3. На 4 надо сдать 11 лаб, на 3 надо сдать 6 лаб.

И тут мы плавно перетекаем в сдачу задач. Сдавать задачи Иртегову для вас будет невероятным новым экспириенсом,

это не похоже ни на что, что у вас было до этого. Каждый мердж задачи в векту Irtegov-accepted для вас будет сровни празднику.

Что стоит знать сразу, чем больше раз сдавали задачу, тем сложнее сдать задачу.

Если ты сдаёшь задачу первым(т.е. её никто ещё не сдал), то Иртегов может её принять без теоретических вопросов, или спросит 1-2 вопроса.

Если ты сдаёшь задачу четвертым, то в течении диалога, вы можете прийти к любому вопросу. Абсолютно любому вопросу. Иртегов считает, что раз вы отучились первый курс,

то вы все знаете программу первого курса(не важно что у вас 3 с 2 пересдачами за предмет или 5 автоматом), как следствие он может спрашивать вопросы и по другим дисциплинам, которые вы прошли.

Собственно перед сдачей задач, лучше бы повторить какую-то часть, где пробелы курса ЦП и С(к ним Иртегов особенно часто возвращается).

Также Иртегов считает, что если это было озвучено на лекции, то вы это должны знать. Соответственно, даже если сдаёте задачу, которая никак не относится к теме лекции(ну это только вы так думаете), велик шанс, что Иртегов спросит и по лекции.

Когда вы всё же пришли к обсуждению задачи, то не надо перебивать Иртегова(он же это будет делать постоянно, тут просто терпим).

Сто процентов будут ситуации, когда вы нашли какую-либо информацию, а Иртегов говорит, что это не верно.

Иртегов признаёт только мануал и официальную документацию, ну и методичку со своей книгой, но там с натяжкой.

Наверно есть ещё какие-то сайты. Но это уже лучше у Иртегова спросить.

И если вы уверены, что вы правильно поняли информации из документации, то тут даже можно поспорить.(у меня два раза так было, обы успешно)

Ещё постоянно будут ситуации, когда Иртегов уходит от вас к другому сдающему, а вы даже не поняли что он хотел. Берём волю в кулак и спрашиваем: "Можете ещё раз повторить, на какой вопрос я должен дать ответ" (или что-то подобное).

Очень облегчает жизнь. Собираем оставшуюся волю в кулак и уточняем:"А где я могу найти ответ на данный вопрос, на каких ресурсах?". Этот вопрос вам ещё сильнее облегчит жизнь.

Иртегов очень любит, когда вы думаете, мыслительный процесс очень важен в диалоге. Если не знаете, что Иртегов хочет услышать, то лучше сказать, что ты не знаешь ответ, чем сидеть гадать, попутно углубляясь в граф знаний Иртегова и отнимая время у одногруппников.

Сразу наизусть заучиваем все ошибки в задачах прошлых лет(этот док вам Иртегова сам выкатит) и не допускаем их. Сэкономит много времени. Ибо если у тебя есть ошибки и ты сдаёшь не первый, то велик шанс, что ты отправишься переписывать задачу.

Лучше бы выучить криты заранее. Иртегов любит и к ним обращаться.

В целом, на этом наверное и всё. Я эту дисциплину закрыл и вам того же желаю. Kiss.

Полезные ссылки

- [Криты дедов](#)
- [Криты нашего homie](#)
- [Конспект лекций с доп. материалами](#)
- [Материалы курса](#)
- [Книга Иртегова](#)

- [Книга Танненбаума](#)
- [Билеты основного потока](#)

Дополнительно

Здесь будут собраны темы, которые, как мне кажется, достойны упоминания, но явно не относятся к каким-то вопросам и не только.

ГЛОССАРИ

Аллокатор - библиотека, которая выделяет и освобождает память.(Иртегов лекция 5),

Объект - это нечто, что обладает идентичностью, состоянием и поведением. (Иртегов лекция 8)

Примитив - кусок данных с которым надо работать определённым образом, то есть у него определены операции, и для манипуляции этой сущностью нужно использовать только эти операции.(Иртегов лекция 8)

POSIX - (Portable Operating System Interface [based on] uniX, переносимый интерфейс операционной системы, основанный на Unix) стандарт, описывающий интерфейс системных вызовов (книга Иртегова с 39)

Процесс - объект ОС, который обладает виртуальным адресным пространством и контекстом безопасности. То есть процесс имеет программу и может исполнять эту программу с какими-то правами. (Иртегов лекция 9)

Нить - это для чего создают иллюзию последовательного выполнения. (звучит странно, но если вдуматься, то смыслол имеется)(Иртегов лекция 8)

Нить - это единица исполнения внутри программы. (Иртегов лекция 9)

Системная нить - это объект ос, но у нее нет своего адресного пространства и она принадлежит процессу. (Иртегов лекция 9)

Примечание: есть процессы без нитей - процессы зомби. Есть нити без процессов - нити ядра.(Иртегов лекция 9)

.Диспетчер памяти — это устройство, осуществляющее трансляцию виртуальных адресов в физические.

Race condition(ошибка соревнований) - зависимость результата исполнения программы от её порядка исполнения, но чтобы это стало ошибкой, надо чтобы некоторые результаты были правильные а некоторые неправильные. (Иртегов лекция 8)

Устройство -

Драйвер -

Стэковый(стековый) кадр - область памяти в пользовательском стэке, куда сохраняются параметры функции (хотя и не всегда и не все), адрес возврата, сохранённые регистры, локальные переменные, а также дополнительно в С указатели на обработчики исключений и деструкторы локальных переменных.

Прямой доступ к памяти (DMA) - работа устройств с памятью напрямую через магистральную шину (говоря проще и точнее, **без использования ЦП**)

Ядро - комплекс программ исполняющихся в привилегированном(системном) режиме (Иртегов лекция 6 вроде бы)

Чистые функции - функции, которые не имеют внутреннего состояния и не имеют побочных эффектов.

CISC- и RISC-архитектуры

Стр. 129

RISC сейчас расшифровывается как *reduced instruction set of commands*, хотя изначально он означал совсем иное - *rational instruction set of commands*. Рациональность набора инструкций процессора заключалась в нескольких отличиях от CISC (*complex instructions set of commands*):

- Отказ от обеспечения обратной бинарной совместимости, за счёт чего количество команд могло ощутимо сократиться

- Отказ от комплексных команд, которые усложняли архитектуру процессора и на практике за счёт оптимизации кода трансляторами использовались не так уж часто (в CISC архитектуре встречались инструкции, которые могли копировать строку или вставлять/удалять элемент списка)
- Третьей отличительной особенностью стало требование большого количества регистров, без которых оптимизации могли работать не всегда

Transputer

Transputer - процессор, использующий стековую или аккумуляторную систему команд. Суть такой систему заключается в использовании в первую очередь без- и одноадресных команд, которые используют в качестве неявных операндов значения из особых регистров либо стэка (тут почему-то и стэк называется особым регистром, но такая формулировка мне совершенно не нравится). В настоящее время стековая система команд широко используется в байт-кодах виртуальных машин Java и в некоторых других языках, использующих JIT-компиляцию.

Вопросы на консультацию

- Кольца доступа и кольца защиты - одно и то же?
 - Существуют источники, где это разные вещи, но в смысле книги Иртегова и его "контекста" они рассматриваются как синонимы
- Может ли мы с уверенностью утверждать, что не любую программу можно сделать реентерабельно? Или всё же стоит сказать, что реентерабельной будет может быть любая программа, но в некоторых случаях это реализуется очень сложно?
 - Обработчик сигнала - не нить!
 - Thread-safe функции - не async signal safe!
 - Некоторые архитектуре по своей природе не позволяют сделать программу реентерабельной
 - Пример ситуации, когда не все примитивы взаимоисключений работают: нить заблокирована мутексом (или другим семафором), получает сигнал, обрабатывает его и проходит в критическую секцию с закрытым мутексом, то есть рискует получить дедлок
- Как в алгоритме близнецов будет расширяться куча? (сложная реабалансировка дерева или это вообще невозможно?)
 - Будет одна неплоная ветка с которой достаточно неудобно работать. В связи с этим зачастую близнецы используются для распределения физической памяти, общий объём которой также определён статически сразу, то есть мы можем всю её разбить на дерево
- Про коды аутентификации Apple iOS
 - Надо сказать, зачем нужна аутентификация и проверка подлинности кода
 - Рассказать про алгоритмы шифрования из приватного и публичного ключа
 - Суть в том, что Apple требует, чтобы все программы в iOS были подписаны Apple, то есть у них полный контроль над тем, что будет на ваших яблочных устройствах. Дополнительная необходима проверка при загрузке iOS на то, что проверка ключей не вырезана из системы
 - *Рассказ про StucksNet - загрузочный вирус, который подписывался одним из производителей железа (realtek), который Microsoft предоставляла для установки драйверов*

Ответы на другие вопросы:

- Что такое файл?
 - Файл - совокупность данных на носителе (совокупность данных, адресуемых по имени)

- Файл – объект в ядре, который может в зависимости от своей специфики использоваться для разных целей.
Такое определение для файла справедливо лишь в течение некоторого времени.
- Ну дальше про определение ФС - ничего интересного
- Разделение привилегий
 - Заключается не только в классических правах Юникса, но и в целом комплексе методов, предназначенных для уменьшения возможного вреда от любого пользователя или процесса
- Третичный загрузчик
 - Это всё может и интересно, но нам явно не нужна инфа глубже книги...
- Немного про RAID-массивы
 - Пятый уровень RAID использует XOR для восстановления данных
 - Шестой уровень RAID использует другой математический метод, который тут приводить не буду, но позволяет он восстановить даже 2 потерянных диска

Билет 1

Вопрос 1: Как происходит загрузка операционной системы? Что такое первичный загрузчик? Вторичный? Как происходит загрузка бездисковых машин?

Из Вахалии: "Unix изнутри", Глава 2 - Ядро и процессы

Само по себе **ядро** – особая программа, работающая непосредственно с аппаратурой. Ядро хранится в файле на диске и загружается процедурой **начальной загрузки (bootstrapping)**, после чего само ядро инициализирует систему и устанавливает среду для исполнения процессов (все глобальные структуры и т.п.), затем запускает несколько начальных процессов, которые в дальнейшем будут порождать остальные процессы. Ядро будет находиться в памяти до момента завершения работы системы. Ядро обеспечивает свою функциональность 4-мя основными способами:

- API системных вызовов
- Отлавливание исключительных ситуаций
- Аппаратные прерывания
- Некоторые системные процессы, которые отвечают за базовые обширные системные задачи (управление активными процессами, пулом памяти и т.п.)

Книга Иртегова, страница 200: 3.11 "Загрузка самой ОС"

Загрузочный монитор – программа, расположенная в ПЗУ по определённому адресу (чаще всего на самой материнке), с которого процессор начнёт исполнение при подаче питания. Эта программа проведёт первичную инициализацию процессора, диагностику ОЗУ и обязательных периферийных устройств. *Можно сказать про консольный монитор, дающий больше свободы на этом этапе.* Первичный загрузчик запускает загрузку самой ОС, причём загружаться она может даже (и чаще всего) с другого ПЗУ.

Чтобы загружать ОС с ПЗУ, загружчик должен обладать модулем этого ПЗУ. Зачастую ПЗУ сами содержат у себя плату с таким программным модулем, что обеспечивает высокую совместимость.

С перфокарт и других последовательных устройств просто читаются все возможные данные, а потом регистр исполнения ставится на начало считанной области.

На ПЗУ с произвольным доступом (чаще всего дисках) загрузочный монитор чаще всего загружает нулевой сектор нулевой дорожки, содержимое которого называется **первичным загружчиком**. Загружчик ищет на диске начало родной ФС этой ОС и загружает из неё определённый файл, которому передаёт управление. В самом простом случае этот файл будет ядром. Если ФС сложная и загружчик размером в 512 байт (стандартный размер

сектора) не сможет вместить всю логику запуска ядра, то он будет загружать вторичный загрузчик, который по размеру значительно больше, а значит, умнее. В особых случаях может существовать также третичный загрузчик.

Возможен вариант также загрузки по сети, который позволяет удобно загружать множество машин и незаменим для бездисковых машин. ПЗУ сетевой карты посыпает на сервер запрос и получает в ответ вторичный загрузчик (*а где мы ищем ядро? Полагаю, что также делаем запросы на сервер*)

После загрузки файла ядра начинается инициализация модулей, которые могут загружаться статически или динамически. Загруженные модули запускают свои подпрограммы инициализации. Завершением инициализации ядра считается запуск определённой программы. В Unix это будет `init`.

В том же разделе книге пишется и про загрузку ОС в PC-совместимых машинах (фактически, много в чём она тут и описана), так что будет достаточно прочесть этот участок из книги и выжимку для ответа и на второй вопрос 27-го билета

ВОЗМОЖНЫЕ ВОПРОСЫ:

1. Что такое ФС? (КРИТ)
 2. Что такое ядро?
-
1. Что такое ОС?
 2. Что такое модуль? (странный немного вопрос, но может быть и он)
 3. Что такое загрузчик в целом? (не только для ОС) (о нём есть 2 КРИТА)
 4. Явно имеет смысл хотя бы мельком пролистать всю эту главу из книги Иртегова

Вопрос 2: организация файловой системы HPFS

Презентация Иртегова

Глава 11 "Файловые системы" (стр. 631).

Для начала вот выдержка из конспект с лекции 23.12.18. Тут определение ФС несколько отличается от того, что записывал в критах Антон, но Иртегов его упоминал минимум дважды, так что имею основания считать надёжным:

- ***Файл**** - системный объект, который можно читать и писать
- ***Регулярный файл**** - совокупность данных, которая адресуется по имени

Файлы на диске переживают включение и выключение системы, то есть сохраняются без питания. Отсюда следует, что файлы на диске лишь на некоторое время (в Unix на время открытия) становятся объектами системы, а в остальное время - это структуры данных на диске

- ***Метаданные**** - данные о данных. К ним можно отнести ***атрибуты файлов****, ***каталоги****, ***информация о занимаемых файлом участках диска****.

3 определения ****файловой системы****:

- документ, описывающий формат метаданных
- драйвер ФС, то есть программный комплекс, позволяющий работать с метаданными этого формата
- экземпляр файловой системы - носитель, размеченный и организованный таким образом, что с ним может работать драйвер ФС

Ну а теперь по книге

Файл — это совокупность данных, доступ к которой осуществляется по ее имени (кусок крита)

Таким образом, файл противопоставляется другим объектам, доступ к которым осуществляется по их адресу. При этом в ОС семейства Unix под файлом понимается любой объект, имеющий имя в файловой системе, однако

файлы, не представляющие собой совокупность данных (каталоги, внешние устройства, псевдоустройства и т.п.) обычно называются **специальными файлами**.

Каталог (директория) - таблица преобразования имён файлов в адреса (**второй кусок крита**)

Кроме имени файла и адреса на диске, в каталоге может храниться дополнительная информация о файле, такая как дата создания и модификации, размер файла, идентификатор владельца и права доступа и т.п. Обычно информация о файле хранится в специальной структуре - **иноде**. Таким образом, в каталоге будет только имя файла и указатель на иноду.

В системах, поддерживающих вложенные каталоги, **полным** или **путевым** именем файла является цепочка вложенных каталогов и имя файла в последнем из них.

Таким образом, **файловая система** - совокупность каталогов и других метаданных (данных о данных), представляющих собой системные структуры данных для отслеживания размещения файлов на диске и свободного дискового пространства. (**последняя часть крита**)

Раздел или слайс - фиксированная непрерывная часть диска, на которой располагается ФС. Обычно разбиение на них производится на уровне драйвера диска, поэтому они называются **логическими дисками**.

ФС может располагаться на нескольких логических дисках, которые могут даже располагаться на разных дисках, за такое объединение отвечает промежуточный слой ОС, обычно называемый **LVM (Logical volume manager - менеджер логических томов)**. Всё пространство, занимаемое ФС, можно называть **томом**.

Пространство имён файлов - совокупность всех допустимых в файловой системе имён файлов. В первую очередь накладываются ограничения на длину имени и используемые в нём символы

Монтирование ФС

Перед использованием ФС, необходимо провести над ней операцию **монтирования**, обычно включающую:

- Проверка типа ФС
- Проверка целостности ФС
- Считывание структур данных ФС и инициализация драйвера ФС
- Установка дополнительных данных вроде dirty-flag (см. далее)
- Включение смонтированной ФС в пространство имён

Далее со стр. 634 идёт инфа про специфику монтирования в ДОС-системах (дисковая ОС - как по мне, понятие **странные**) имени ФС при монтировании в разных системах. Если первое очень редко сейчас встречается, то второе достаточно любопытно и почитать можно.

К интересной, но не очень важной инфе также отнесу главу 11.1.2. "Формат имен файлов" на стр. 637

Операции над файлом

Большинство современных ОС рассматривают файл как неструктурированную последовательность байтов переменной длины. По стандарту POSIX для файлов определены следующие операции:

- `open()` - тут я оставлю [ссылку на конспект](#) и дополнительно дам определение дескриптора из книги
 - "ручка" или дескриптор файла - целое число, являющееся индексом в таблице открытых файлов данной задачи и позволяющее обращаться к этому файлу
 - Про `lseek()`, `read()`, `write()` и `fcntl()` читай в том же конспекте сразу после `open()`
 - Про `mmap()` также [из конспекта](#)

В ОС не из семейств Unix файлы могли рассматриваться как набор записей постоянной или переменной длины. Хотя такой подход мог быть удобен для программирования, его поддержка на уровне ядра часто оказывалась довольно затратной

Глава 11.1.4. "Тип файла" (стр. 642) тоже не представляется особо интересной, поэтому будет лишь краткая выжимка

Определение того, какая программа должна работать с файлом, было бы очень удобно, но создавало и некоторые проблемы:

- При определении типа файла по расширению ОС должна где-то хранить связи определённых программ с расширениями. Проблема - конфликты расширений
- Магические числа или сигнатуры - короткие записи в начале файлов, которые позволяли ассоциировать их с программами. Проблема всё та же - конфликты магических чисел. Впрочем, для сигнатур такая проблема встречается реже (современные Unix-сигнатуры имеют вид `#!<путь/к/исполняемому/файлу>`), однако возможны ситуации, когда начало текстового файла будет распознано как сигнтура
- Расширенные данные в инодах, которые содержат дополнительную ОС-специфичную информацию о том, какой программой должен обрабатываться этот файл и, возможно, о чём-то ещё

Сейчас опишу здесь ВСЕ ФС из глав 11.2, 11.3, которые упоминаются в билетах, а потом буду раскидывать их по соответствующим билетам

RT-11 Иртегов лекция 16 48 минута

Одна из немногих ФС, которая успешно применялась и на ленточных устройствах, и на устройствах с произвольным доступом.

Файл занимает непрерывную область на диске в виде блоков по 512 байт (минимальный сектор диска)

Существует корневой каталог, который содержит имена файлов и их длину в блоках (в том числе информацию о свободном месте). Следующий файл начинается после конца предыдущего (то есть нам достаточно знать лишь адрес блока начала первого файла в каталоге). Такая организация без указания адреса первого блока файла возможна за счёт одной корневой директории и расположения файлов на диске в том же порядке, в каком они расположены в каталоге.

Из-за такой организации мы получаем ряд недостатков:

- Необходимо заранее указывать длину создаваемого файла, а расширение в привычном понимании невозможно (текущее содержимое файла будет копироваться в новый файл большего размера)
- После удаления файлов мы можем столкнуться с фрагментацией, решить её могут утилиты дефрагментации, которые "сдвинут" файлы к началу диска, но в далёкие времена из-за аппаратных сбоев во время этого долгого процесса файлы могли быть безвозвратно повреждены

Достоинства:

- Простота
- Реализация влезает в 64 кбайта оперативки.

Сейчас почти нигде не используется за исключением разве что устройств с очень малой ОЗУ либо устройств с разовой записью (компакт-диски)

ISO-9660 (CDFS) Иртегов 16 лекция 53 минута

Файлы занимают непрерывную область на диске. В каталоге хранятся атрибуты файла, имя файла, адрес его первого блока и длина файла.

Расширение файлов не допускается, а также никак не обозначается внутреннее свободное место. Свободным считается только место после конца последнего файла. При добавлении новых файлов в каталог он также будет перемещён, в новое место и расширен

В этой ФС могут быть вложенные каталоги, которые также являются файлами ФС во всех смыслах этого слова (на них есть ссылка из каталога-родителя, а их содержимое - таблица файлов каталога. Корневых каталогов может быть 16 (их адреса располагаются в системных секторах диска 0-15).

Формат имён файлов: `32 символа имя . 32 символа расширение ; версия` (все символы в ASCII)

У каждой корневой директории может быть свой формат имён и некоторые другие различия в хранимых данных (см. стр. 652)

Данная ФС достаточно удобна для записи на неизменяемые носители

FAT Иртегов лекция 16 57 минута

Файл представлен в виде связанного списка, но указатели списка лежат не в блоках файла, а в таблице **FAT (file allocation table) (далее - "таблица")**, в которой есть по одной записи на каждый блок(сектор) диска (512 байт). Если блок принадлежит файлу, то запись в таблице будет содержать адрес следующего блока того же файла, в ином случае блок может быть помечен как свободный (0) или плохой. Последний блок файла помечается записью 1

Самый старый вариант - FAT12 - 12 бит длины на запись ⇒ 4096 блоков ⇒ 2 МБ (дискеты)

Далее появился FAT16 на 32 МБ

FAT32 было поначалу очень проблемно создавать из-за малой ОЗУ, так что следующим этапом стала FAT16 с логическими секторами или кластерами, размер которых мог быть степенью двойки старше 512 байт. Благодаря этому размер памяти для носителя ФС удалось увеличить до 2 ГБ

В полноценной FAT32 обычно берутся кластеры по 4 КБ

Одной из самых больших проблем FAT стала необходимость при работе с файлами постоянно обращаться к таблице, из-за чего магнитная головка постоянно прыгала и скорость работы с файлом значительно падала. Решением стала комбинация из создания кластеров различных размеров и кэширования уменьшенной за счёт этого таблицы (иногда её просто грузили в ОЗУ), однако и этот вариант был лишь компромиссным. (подробнее см. стр. 655)

Сложные ФС лекция 16 108 минута

В большинстве сложных ФС в начале идёт заголовок или суперблок, который содержит информацию о файловой системе, размер выделенного для неё тома, адреса системных структур и другую информацию.

Большая часть метаданных о файлах и адреса их блоков хранятся в таблицах инодов, а в каталогах содержатся только имена файлов и ссылки на эту таблицу (либо таблицы, не всегда таблица монолитна). Информация о свободных и занятых блоках на диске чаще всего хранится в битовых масках

HPFS Иртегов лекция 16 118 20

Иноды (лучше говорить **файловые записи**) здесь называются *fnode*, в каталогах хранятся ссылки на них и имена файлов. Каталоги организованы через В-деревья.

В-дерево - структура данных, оптимизированная для хранения логически отсортированных данных на дисковых носителях. Выделяют также В+ и В*-деревья.

В-дерево представляет собой бинарное дерево поиска, в котором каждый узел представляет собой массив длиной с сектором диска или кластер. Все массивы отсортированы, левый потомок узла содержит элементы меньше левого конца массива, а правый - больше правого. По таким деревьям можно достаточно просто и быстро проводить поиск и добавление элементов, но вот удаление и ребалансировка являются затратными и сложными операциями.

В+-дерево состоит из узлов с массивами индексных блоков и блоков данных. Индексные блоки указывают на другие узлы (индексные либо с данными) Блоки данных всегда будут листьями и содержат собственно данные

Файловая запись занимает один блок на диске и содержит в самом базовом случае имя файла, время его создания и расширенные атрибуты. Далее идёт 10 экстентов - записей из двух чисел: начала непрерывной последовательности блоков на диске и её длины. Такой принцип называется

Выбор места для расположения файла происходит по методу *worst fit*, то есть выбирается самая большая свободная область. За счёт этого в обычной ситуации файлы фрагментируются слабо, однако при необходимости выделяются отдельные экстенты, в которых уже будут экстенты блоков файла, организованные в В-дерево.

Благодаря этому, размер файла становится теоретически неограниченным ничем, кроме количества свободного места на диске.

Экстенты открытых файлов и карта свободных секторов размещаются в ОЗУ, что обеспечивает высокую скорость работы с файлами, благодаря экстентам также упрощается произвольный доступ к файлу

FFS (UFS)

Каталоги содержат только имена файлов и номер инодов. Диск разделён на группы цилиндров, каждая группа цилиндров содержит копию суперблока с битовой маской этого участка диска и таблицей инодов файлов на этом участке. Такой метод значительно ускоряет работу ФС и повышает её надёжность - даже потеря суперблока приведёт к потере лишь небольшой части файлов ФС

Данные в иноде

Инфа отсюда играет большую роль даже вне рамок вопроса об этой ФС. Есть отдельный билет про иноды и ссылки, так что может потом ещё чутка дополню этот раздел

```
int stat(const char* path, const stat *buf) - возвращает 0 при успехе и -1 иначе и записывает атрибуты в особую структуру:
```

```
struct stat {
    dev_t st_dev; // Флаг файла-устройства
    ino_t st_ino; // Индекс inode
    mode_t st_mode; // Права доступа, но не та же маска, что обычно
    nlink_t st_nlink;
    uid_t st_uid;
    gid_t st_gid;
    dev_t st_rdev; // флаг драйвера файловой системы
    off_t st_size;
    timeb st_atim, st_mtim, st_ctim; // Время доступа, модификации и создания (на самом деле
    // время последнего изменения атрибутов). Создание и модификация хранятся с точностью до миллисекунд,
    // доступа - до шести часов (и обновиться он может не чаще чем раз в такой же период)
};

#define st_atime st_atim.tv_sec;
#define st_mtime st_mtim.tv_sec;
```

Старшие 4 бита mode_t кодируют тип файла:

```
#define S_IFMT 0xF000 /* type of file */
#define S_IFIFO 0x1000 /* fifo */
#define S_IFCHR 0x2000 /* character special */
#define S_IFDIR 0x4000 /* directory */
#define S_IFBLK 0x6000 /* block special */
#define S_IFREG 0x8000 /* regular */
#define S_IFLNK 0xA000 /* symbolic link */
#define S_IFNAM 0x5000 /* XENIX special named file */
#define S_IFSOCK 0xC000 /* socket */
#define S_IFDOOR 0xD000 /* Solaris door */
#define S_IFPORT 0xE000 /* Solaris event port */
```

Последующие 12 бит кодируются по классике

- lstat(const char* path, const stat *buf) - версия для получения атрибутов ссылки
- fstat(int fd, const stat *buf) - то же, что и stat, но по дескриптору

Псевдоустройства - говорят, что они устройства, имеют драйвера, но не имеют за собой реального устройства

В UFS размещение файла на диске описывается 13-ю числами: 10 - номера блоков, а ещё 3 используются при необходимости для косвенных блоков, которые также содержат номера блоков и, возможно, номера следующих косвенных блоков.

Интересной и достаточно полезной особенностью UFS является механизм ссылок, который позволяет обращаться к одной и той же иноде из разных мест по разным именам (в рамках одной ФС). Такой механизм называется жёсткой ссылкой и кроме многих удобств создаёт также и ряд проблем:

- Жёсткие ссылки на каталоги могут создать закольцованный путь, поэтому запрещены
- Реальное удаление файла произойдёт лишь когда счётчик ссылок в иноде будет равен нулю, что довольно сложно отслеживать. Отвязывание файла от иноды (aka удаление этого имени) происходит командой `unlink`, причём для её использования над файлом достаточно обладать лишь правами записи в каталог, где файл располагается (можно ужесточить проверку прав, установив на каталог sticky-bit)

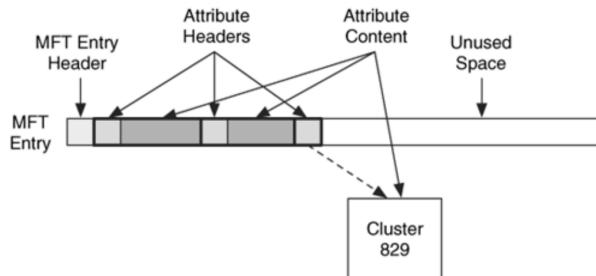
Решением этих недостатков стал механизм **символических ссылок**. В иноде этих специальных файлов хранятся не адреса блоков данных, а строка с именем того файла, на который указывает символическая ссылка. Ссылка может указывать на уже удалённый файл или и вовсе на несуществующий, никак не влияет на счётчик ссылок файла, на который указывает и может указывать на файл в других ФС.

NTFS Иртегов 16 лекция 1 16

Пример: структура записи MFT NTFS

- Запись MFT состоит из атрибутов.
- Атрибут может быть
 - резидентным (размещен в записи MFT)
 - нерезидентным (под него выделены экстенты данных)
 - Размещение больших атрибутов описывается в виде В-дерева
- Атрибуты:
 - Обязательные атрибуты файла: тип, дата, владелец, security id всегда резидентный
 - Имена файла (если у файла есть жесткие ссылки, может быть несколько)
 - ACL (список управления доступом)
 - Потоки данных (может быть несколько)

Структура записи MFT NTFS



NTFS построена на экстентах.

Тут возникает проблема в том, что чётко отделимого от других описания особенностей этой ФС нет, так что придётся собирать по крупицам...

ФС с журналированием системных данных

Иноды файлов хранятся в динамической таблице инодов, которая допускает расширение, но не сжатие
если найдете еще инфы, то допишите ([Inkpv](#))

- **Длинные имена файлов в ОС семейства СР/М**

Поддерживает имена файлов длиной до 256 символов, содержащие печатаемые символы и пробелы. Точка

считается частью имени, как и в UNIX, и можно создавать имена, содержащие несколько точек.

[\(Иртегов с.642\)](#)

- **Жесткие связи в VMS и Windows NT/2000/XP**

В

файловой системе VAX/VMS данные о размещении файлов на диске хранятся в специальном индексном (index) файле; каталоги же хранят только индексы записей в этом файле. Основное отличие этой структуры от принятой в Unix состоит в том, что вместо статической таблицы или набора таблиц используется динамическая таблица, пространство для которой выделяется тем же методом, что и для пользовательских файлов. Этот же подход реализован в файловой системе **NTFS**, используемой в Windows NT/2000/XP, но в ней индексный файл называется **MFT** (Main File Table — главная таблица файлов) (Аналог таблицы инодов).

VAX/VMS и Windows NT позволяют создавать дополнительные имена для файлов, хотя в VMS утилиты восстановления ФС выдают предупреждение, обнаружив такое дополнительное имя. Все имена файла в этих ФС обязаны находиться в одной файловой системе. Кроме того, операция удаления файла в VMS ведет себя не так, как в Unix: применение операции удаления к любому из имен приводит к удалению самого файла, даже если существовали и другие имена.

[\(Иртегов с.671\)](#)

- NTFS в Windows NT/2000/XP — устойчивая к сбоям файловая система. Практически все такие ФС основаны на механизме, который по-английски называется intention logging (регистрация намерений). [\(подробно про ФС с регистрацией намерений Иртегов с.681 в пдф\)](#)

- Поддерживает транзакции при работе с системными данными [\(Иртегов с.684\)](#)

- Ряд других ФС, использующих динамическую таблицу инодов, в том числе NTFS, допускают расширение этой таблицы (у NTFS она называется MFT — Main File Table), но никогда не сжимают ее. Действительно, сжатие такой таблицы может быть сложной задачей из-за ее фрагментации. Это приводит к определенной проблеме — создав много коротких файлов и затем удалив их, злоумышленник может уничтожить часть объема ФС.

[\(Иртегов с.689\)](#)

Билет 2

Вопрос 1: Распределение памяти алгоритмами близнецов и парных меток. Ограничения этих алгоритмов.

При алгоритме парных меток мы тратим некоторое количества памяти на доп. информацию, а также получаем внешнюю фрагментацию, от которой можем +- эффективно избавляться (объединяя свободных соседей либо проводя дефрагментацию), но тем не менее, она возникает

При алгоритме близнецов мы вынуждены использовать дополнительную структуру для бинарного дерева, пусть и простую + не можем объединить освободившиеся участки из разных ветвей, пусть они и лежат рядом + будет возникать внутренняя фрагментация

Вопрос 2: Аутентификация и проверка подлинности кода в Apple iOS.

Найс вопрос. Главное полезный.

Ну будем работать с тем, что имеем.

А имеют здесь только нас.

Будем использовать официальное руководство APPLE.

Кстати, в нём есть ответ на любой вопрос по безопасности iOS

Загрузка iOS

Обязательная подпись кода

После запуска ядро iOS определяет, какие процессы пользователей и приложения могут быть запущены в системе. Чтобы помочь проверить, что все приложения получены из известного и утвержденного источника и не подделаны, iOS требуют, чтобы весь исполняемый код был подписан с помощью выпущенного компанией Apple сертификата. Установленные на устройстве приложения, такие как Почта и Safari, подписаны Apple. Приложения сторонних разработчиков также должны быть проверены и подписаны с помощью выпущенного компанией Apple сертификата. Обязательная подпись кода расширяет концепцию цепочки доверия с операционной системы на приложения. Она помогает принять меры, чтобы приложения сторонних разработчиков не загружали неподписанные фрагменты кода или не использовали самомодифицирующийся код.

Процесс подписания приложений разработчиками

Разработчики могут подписывать свои приложения с помощью сертификатов. Они также могут встраивать в свои приложения различные программные среды и использовать сертификаты, выданные Apple, для проверки кода.

- *Проверка сертификата.* Для разработки и установки приложений на устройствах iOS или iPadOS разработчики должны зарегистрироваться в Apple и присоединиться к программе Apple Developer Program. Перед выдачей сертификата компания Apple проверяет личность каждого разработчика, будь то частное лицо или компания, в реальном мире. Используя эти сертификаты, разработчики могут подписывать приложения и отправлять их в App Store для распространения. В результате все приложения в App Store отправляются идентифицированными людьми и организациями, что выступает в качестве сдерживающего фактора для создания вредоносных приложений. Кроме того, Apple проверяет все приложения, что помогает выявить явные ошибки или другие заметные проблемы и определить, соответствуют ли приложения своему описанию. Эта проверка дает пользователям дополнительную уверенность в качестве приложений, которые они покупают.
- *Проверка подписи кода.* Разработчики приложений для iOS и iPadOS могут встраивать в свои приложения различные программные среды, используемые самим приложением или встроенные в него расширениями. Чтобы защитить систему и другие приложения от загрузки стороннего кода в их адресное пространство, в момент загрузки система выполняет проверку подписи кода для всех динамических библиотек, на которые ссылается процесс. Эта проверка выполняется с помощью идентификатора команды (Team ID), который извлекается из выпущенного компанией Apple сертификата. Идентификатор команды представляет собой десятизначную буквенно-цифровую строку, например 1A2B3C4D5F. Приложение может ссылаться на любую библиотеку платформы, поставляемую вместе с системой, и любую библиотеку с таким же идентификатором команды в подписи кода, как у основного исполняемого модуля. Поскольку исполняемые модули, поставляемые с системой, не имеют идентификатора команды, они могут ссылаться только на библиотеки, которые также поставлялись с системой.

Аутентификация

Единый вход

iOS и iPadOS поддерживают аутентификацию в корпоративных сетях посредством единого входа в систему (SSO). Технология SSO работает с сетями на основе протокола Kerberos, обеспечивая аутентификацию пользователей в службах, к которым у них есть доступ. SSO можно использовать для целого ряда различных сетевых операций, начиная с безопасных сеансов Safari и заканчивая приложениями сторонних разработчиков. Также поддерживается аутентификация на основе сертификатов, такая как PKINIT.

macOS поддерживает аутентификацию в корпоративных сетях посредством Kerberos. Приложения могут использовать Kerberos для аутентификации пользователей в службах, к которым им разрешен доступ.

Для взаимодействия со шлюзами аутентификации на основе Kerberos технология SSO в iOS, использует токены SPNEGO и протокол HTTP Negotiate.

В iOS, iPadOS и macOS поддерживаются следующие типы шифрования:

- AES-128-CTS-HMAC-SHA1-96
- AES-256-CTS-HMAC-SHA1-96
- DES3-CBC-SHA1

- ARCFour-HMAC-MD5

Расширяемая технология единого входа

Разработчики приложений могут предоставлять собственные реализации технологии единого входа на основе расширений SSO. Расширения SSO вызываются, когда нативному приложению или веб-приложению необходимо обратиться к какому-либо поставщику услуг идентификации для аутентификации пользователя. .

Списки контроля доступа

Данные связки ключей хранятся(видимо это про цифровую подпись и шифрование) в отдельных разделах и защищены с помощью списков контроля доступа (ACL). Благодаря этому доступ к учетным данным, которые хранятся в приложениях сторонних разработчиков, не могут получить приложения с другими учетными данными. Этот механизм защищает учетные данные на устройствах Apple, используемые для аутентификации в различных приложениях и службах внутри организации.

Может вообще Иртегов хочет про Secure Enclave услышать

Билет 3

Вопрос 1: Алгоритмы поиска жертвы при страничном обмене и кэшировании. Критерии выбора и влияние алгоритма на производительность. Что такое рабочее множество страниц?

Вопрос 2: Сигналы в системах семейства Unix.

Методичка Иртегова НАМНОГО ПОНЯТНЕЕ И ИНФОРМАТИВНЕЕ ТОГО ЧТО НАПИСАНО ДАЛЬШЕ, КРАЙНЕ РЕКОМЕНДУЮ, КОЛИЧЕСТВО СТРАНИЦ ПУГАЕТ, НО ПО ФАКТУ ТАМ ПРОСТО ДОХЕРА ПРИМЕРОВ КОДА

Из конспекта:

- Теория про сигналы
- Использование системных вызовов для разных сигналов

Иртегов лекция 11

Сигналы - некое событие, которое можно послать от одного процесса другому, и это событие имеет некую принудительность, то есть это событие нельзя игнорировать.

Все сигналы шлются ядром, то есть сигналы это чисто программная абстракция.

Сигналы могут доставляться процессу, но не ядру.

Посылка сигналов всегда происходит только через ядро.

У процессов есть маска пендинг с сигналами которые пришли но не обработаны, ну и когда процесс получает управление, переходя из системного режима в пользовательский, он смотрит эту маску и делает действия по обработки.

Если процесс сидел в ожидании, то его будят, и он получает ошибку EINTR что системный вызов был прерван сигналом то есть системный вызов возвращает ошибку но ваш процесс будится и возвращается из системного вызова в контекст пользовательский и в пользовательском контексте уже зовётся обработчик.

signal - когда запускается обработчик и прилетает тот же сигнал то обработчик этого сигнала сбрасывается на SIG_DFL, если это не то чего мы хотим, то мы можем в обработчике засунуть обработчик на себя.

sigset - запускает sighold он запоминает сигналы которые ему приходят пока сидит в обработчике и при выходе получается эти сигналы.

Иртегов лекция 12

Сигналы и прерывания возникают асинхронно но они не являются нитями с точки зрения планировщика. То есть когда вам прилетает обработчик сигнала он вам прилетает, обрабатывается в той нити которая была.. на самом деле там хитро

в случае синхронных сигналов обработчик сигналов зовётся в той нити в которой произошёл источник сигнала в случае асинхронных сигналов вообще говоря он может позваться в любой нити вашего процесса, тем не менее планировщик будет считать что он по-прежнему сидит в этой же нити. Хотя зовется совсем другой код.

Наиболее очевидная проблема к которой это может привести состоит в том что если например вы зовёте какую-нибудь функцию и у этой функции внутри есть блокировка и вы сидели внутри этой функции, Потом вам прилетает обработчик сигнала и вы зовете из обработчика сигнала функцию которая пытается захватить эту же блокировку, тогда произойдёт дедлок.

В обработчике сигналов нельзя использовать незащищённые критические секции.

Презентация

ТАБЛИЦА СИГНАЛОВ В ВИКИПЕДИИ

Иргетов с 359

Многие процессоры используют механизм, родственный прерываниям, для обработки не только внешних, но и внутренних событий: мы с вами уже сталкивались с исключительными ситуациями (**exception**) отсутствия страницы и ошибки доступа в процессорах с виртуальной памятью.

Большинство современных процессоров предоставляют **исключения** при неизвестном коде операции, делении на ноль, арифметическом переполнении или, например, выходе значения операнда за допустимый диапазон в таких операциях, как вычисление логарифма, квадратного корня или арксинуса.

Исключительные ситуации обрабатываются аналогично внешним прерываниям: исполнение программы останавливается, и управление передается на процедуру-обработчик, адрес которой определяется природой исключения.

Иргетов с 733

Простой обработчик ошибок.

В системах семейства Unix этот механизм(видимо механизм обработки ошибок) называется **сигналами** (signal). Сигналы делятся на два класса:

- синхронные (т. е. возникающие в момент исполнения программой определенной операции, например деления на ноль);
- асинхронные (т. е. возникающие из-за внешних по отношению к программе событий).

Примерами синхронных сигналов являются:

- уже упоминавшееся деление на ноль и другие арифметические ошибки —

SIGFPE (SIGnal Floating Point Exception — исключение при работе с плавающей точкой), хотя он генерируется и при целочисленном делении на ноль;

- ошибка доступа к памяти — **SIGSEGV** (SIGnal Segmentation Violation — нарушение сегментации);
- ошибка шины — **SIGBUS**, обращение к невыровненным словам на процессорах, которые этого не допускают;
- разрыв трубы — **SIGPIPE**, попытка записи в трубу или сокет, противоположный конец которых закрыт;
- чтение с терминала— **SIGTTIN**, генерируется, если фоновый процесс пытается читать данные с терминала.

Примерами асинхронных сигналов являются:

- прерывание пользователем— **SIGINT** (SIGnal INTerrupt), генерируется при нажатии пользователем комбинации клавиш <Ctrl>+<C> (ASCII ETX, символ

прерывания передачи); в старых системах этот сигнал генерировался при получении символа ASCII DEL;

- завершение терминальной сессии — **SIGHUP** (SIGnal HangUP), при подключении терминала по модему генерируется при потере модемом несущей и/или разрыве телефонной линии, т. е. когда "повесили трубку";
- будильник— **SIGALARM**, генерируется в определенный момент времени по таймеру, устанавливаемому системным вызовом `alarm` ;
- потеря питания — **SIGPWR**, компьютерах общего назначения может рассыпаться демоном, наблюдающим за состоянием источника бесперебойного питания; на ноутбуках — при приближении батареи к разряду;
- исчерпание квоты времени центрального процессора — **SIGXCPU**.

Сигналы могут также генерироваться искусственно; для этого служит системный вызов **kill** (в действительности, это семейство системных вызовов с разными механизмами выбора процессов, которым будет послан сигнал) и одноименная команда `shell`.

Функция обработчик

Процесс может зарегистрировать для каждого из сигналов **функцию обработчика**. Эта функция вызывается при возникновении события, соответствующего сигналу. Точнее, при возникновении события ядро системы просто устанавливает флаг в дескрипторе процесса. При каждой передаче управления процессу — при возврате из системных вызовов, при возврате из обработчиков исключений и прерываний и при возврате управления вытесненному процессу — планировщик проверяет, есть ли у процесса необработанные сигналы.

Если они есть, то сначала вызываются обработчики и только потом — если эти обработчики не завершили процесс — управление передается коду процесса.

В старых версиях Unix флаги были реализованы в виде битовой маски, каждый бит которой соответствовал номеру допустимого сигнала. Таким образом, если до обработки сигнала произошло второе аналогичное событие, то информация о нем терялась. Современные системы поддерживают "надежные" сигналы, которые доставляются столько раз, сколько раз произошло событие. Если в момент возникновения сигнала процесс исполнял блокирующийся системный вызов, этот системный вызов прерывается.

Таким образом, если во время возникновения сигнала исполнялся системный вызов, то управление будет передано в его точку возврата; при этом код возврата будет сигнализировать об ошибке, а значение `errno` будет равно `EINTR`. Если разработчик программы все-таки желает исполнить системный вызов, он должен проверить код ошибки и, если он равен `EINTR`, повторить вызов с теми же параметрами.

Вместо обработчика можно также зарегистрировать "функции" **SIG_DFL** и **SIG_IGN**. Эти "функции" представляют собой численные значения, приведенные к типу указателя на функцию.

Тут можно *тап* почитать

SIG_IGN (от `IGNore`) обозначает игнорирование сигнала; при возникновении соответствующего события ничего не происходит, никакие обработчики не вызываются, системные вызовы не прерываются, информации о пришедшем сигнале не сохраняется,

SIG_DFL (DeFaUlт, по умолчанию) соответствует системному обработчику по умолчанию; для разных сигналов эти обработчики, вообще говоря, различны, но их общая номенклатура не так уж широка.

Для большинства сигналов обработка по умолчанию состоит в завершении процесса. Для некоторых сигналов, таких как **SIGFPE SIGSEGV**, процесс завершается с созданием посмертного дампа памяти. Дамп создается в текущем каталоге в файле с именем `core`

Есть группа сигналов, реакция на которые состоит в остановке процесса. Это такие сигналы, как:

- **SIGTSTP** — Terminal StoP, остановка с терминала, генерируется при нажатии `<Ctrl>+<Z>`;
- **SIGSTOP** — безусловная остановка, генерируется только программно;

- **SIGTTIN, SIGTTOOUT**— ввод и вывод с терминала, генерируются для фонового процесса, который пытается вести диалог с пользователем;
- **SIGTRAP**— точка останова в отладчике.

Процесс, остановленный такими сигналами, не завершается, но приостанавливается. Чтобы возобновить его исполнение, необходимо послать ему сигнал **SIGCONT**. Сигнал **SIGTRAP** используется отладчиками; остальные сигналы этой группы применяются **командными процессорами**, поддерживающими управление заданиями, такими как csh, ksh и bash.

Обработчик сигнала получает единственный параметр — **номер обрабатываемого сигнала**. Таким образом, один обработчик может использоваться для нескольких сигналов, но, с другой стороны, представления обработчика о том, в каком контексте произошла ошибка, следует описать как достаточно смутные.

Современные Unix-системы реализуют несколько типов параметризованных сигналов, которые, кроме номера сигнала, могут получить или запросить дополнительную информацию о контексте ошибки

Обработчик имеет тип `void`, т. е., в соответствии с синтаксисом языка C, не возвращает значения. Таким образом, набор средств, с помощью которых обработчик сигнала может сообщить основной нити программы об ошибке, достаточно узок; далее в этом разделе мы рассмотрим некоторые методы, с помощью которых можно достичь более тесной координации действий между обработчиками и основной нитью программы.

Одной из важных проблем при работе с сигналами является то, что сигнал происходит, вообще говоря, в произвольные моменты по отношению к исполнению основной нити программы, поэтому при взаимодействии обработчика с основной нитью мы сталкиваемся со всем тем букетом проблем, который описывался в главе 7, — критическими секциями, нереентерабельными функциями и т. д.

Основным средством решения этих проблем является маскирование сигналов.

Процесс имеет атрибут, называемый

маской сигналов (signal mask). Этот атрибут представляет собой битовую маску, в которой каждому типу сигнала соответствует один бит.

Общее количество допустимых сигналов невелико, в старых системах оно не превосходило 32, в новых — ненамного превосходит это значение, поэтому 64-разрядной маски вполне достаточно. Мaska проверяется при каждой попытке доставить сигнал; если соответствующий бит установлен, то сигнал не доставляется — ни установленный программой, ни системный обработчики не вызываются, системные вызовы не прерываются, но, в отличие от игнорируемых сигналов, информация о сигнале сохраняется: когда бит в маске будет сброшен, сигнал будет доставлен.

Современные системы предоставляют вызовы, которые реализуют атомарные блоки операций, например установку обработчика и **размаскирование** сигнала или, напротив, **размаскирование** сигнала и вход в системный вызов `pause()` (такой блок может быть полезен для избежания ошибок потерянного пробуждения).

Билет 4

Вопрос 1: Инверсия приоритета. Способы ее предотвращения и способы обхода этой проблемы.

Иртегов с 461

Читать про приоритеты.

В системах с приоритетным планированием при взаимодействии процессов с разными приоритетами возникает ряд специфических проблем, объединяемых названием **инверсия приоритета** (priority inversion).

Один из наглядных примеров этой проблемы мы видели в предыдущем разделе, когда обсуждали фоновое разбиение текста на страницы в текстовом процессоре Microsoft Word. Действительно, при сохранении документа Word должен завершить разбиение на страницы, а для этого он вынужден остановить редактирование, иначе есть риск, что переразбиение на страницы никогда не завершится. Таким образом, при редактировании большого

документа с включенным автосохранением пользователь большую часть времени взаимодействует с высокоприоритетным и действительно быстро реагирующими потоком, отвечающим за редактирование, — но иногда оказывается вынужден ждать завершения работы низкоприоритетного фонового потока. Это выглядит как внезапное "зависание" Word. В добавок ко всему, этот фоновый поток вынужден делиться ресурсами с не менее жаждым до процессорного времени потоком, который занимается фоновой проверкой орфографии...

Сейчас, скорее всего, эта проблема не актуальна, всё таки уже 24 век, поэтому лучше говорить о прошлом.

Если в интерактивных приложениях инверсия приоритета только раздражает, в системах реального времени она может приводить к действительно серьезным проблемам.

При расчете времени реакции на событие разработчик системы реального времени должен принимать во внимание не только время исполнения кода, непосредственно обрабатывающего это событие, но и времена работы всех **критических секций(КРИТ)** во всех нитях, которые могут удерживать **мутексы**, необходимые обработчику, — ведь обработчик не сможет продолжить исполнение, пока не захватит эти **мутексы**, а произвольно снимать их нельзя, потому что они сигнализируют, что защищаемый ими разделяемый ресурс находится

в несогласованном состоянии.

Если высокоприоритетная нить пытается захватить мутекс, занятый низкоприоритетной нитью, то в определенном смысле получится, что эта нить будет работать со скоростью низкоприоритетного процесса.

В условиях, когда планировщик не обеспечивает справедливого распределения времени центрального процессора, а в системе наравне с высокоприоритетной нитью (работа которой нас интересует) и низкоприоритетной (которая держит мутекс) существуют еще среднеприоритетные нити, может оказаться, что низкоприоритетная нить не будет получать управления в течение значительного времени. На это же время будет заблокирована и высокоприоритетная нить.

Особенно серьезна эта проблема, когда высоко- и низкоприоритетная нити относятся к разным

классам планирования — а в системах реального времени так оно и есть.

Аналогичная проблема может возникать также при работе с синхронными примитивами гармонического взаимодействия — линками компьютера и т. д.

Способы ее предотвращения и способы обхода этой проблемы

Основным средством борьбы с инверсией приоритета является **наследование приоритета** (priority inheritance). Обычно наследование контролируется флагом в параметрах мутекса или в параметрах системного вызова, захватывающего мутекс. Если высокоприоритетная нить пытается захватить мутекс с таким флагом, то приоритет нити, удерживающей этот мутекс, **приравнивается** приоритету нашей нити. Таким образом, в каждый момент времени реальный приоритет нити, удерживающей мутекс, равен **наивысшему** из приоритетов нитей, ожидающих этого мутекса.

Более радикальное решение называется потолком приоритета (priority ceiling). Оно состоит в том, что приоритет нити, удерживающей мутекс, приравнивается наивысшему из приоритетов нитей, которые **могут** захватить этот мутекс. Разумеется, во время исполнения определить потолок приоритета невозможно, он должен устанавливаться программистом как параметр мутекса.

Легко понять, что во время работы в критических секциях, защищенных такими мутексами, задачи — даже низкоприоритетные — должны подчиняться всем ограничениям, которые может накладывать исполнение в режиме реального времени. Если мы в каком-то смысле не доверяем низкоприоритетной нити (в частности, не можем оценить время, в течение которого она будет удерживать мутекс), нам следует отказаться от использования разделяемой памяти для взаимодействия с этой нитью и перейти к каким-либо буферизованным средствам обмена данными.

При этом достаточно обеспечить гарантированное время передачи данных в буфер; для того, чтобы избежать переполнения буфера, достаточно того, чтобы средний темп генерации данных высокоприоритетной нитью был ниже среднего же возможного темпа их обработки низкоприоритетным потребителем. Впрочем, если темп такой

обработки подвержен большим флюктуациям (например, из-за наличия в системе других процессов), может оказаться необходимо предусмотреть возможность сброса части данных для борьбы с переполнением буфера — именно так борются с перегрузками сетевые маршрутизаторы и коммутаторы.

Управление наследованием или потолком приоритета для мутексов в обязательном порядке предоставляется системами реального времени и некоторыми многопоточными API общего назначения, например POSIX Thread library.

Необходимо отметить, что и наследование, и потолок приоритета применимы только к мутексам и другим примитивам синхронизации, **для которых можно указать, какая именно нить их удерживает**. Точнее говоря, для борьбы с инверсией приоритета необходимо, чтобы приоритет повышался для нити, которая **будет освобождать семафор**. Для мутексов это всегда та же нить, которая его захватывала, но для семафоров-счетчиков это, вообще говоря, неверно.

Поэтому даже те системы, которые предоставляют наследование или потолки приоритетов для мутексов, не предоставляют аналогичных средств для семафоров-счетчиков. В книге приводятся результаты экспериментов над системой реального времени QNX, из которых видно, что эта ОС реализует наследование приоритетов на мутексах, но при использовании семафоров-счетчиков можно получить классический случай инверсии приоритета.

Вопрос 2: Линки в транспьютере

Билет 5

Вопрос 1: Определение задачи реального времени. Чем системы РВ отличаются от систем разделенного времени? Пример архитектуры ОС реального времени.

Иртегов с. 16

Одним из важнейших типов гарантий, которые интересны далеко не только системам коллективного пользования, является так называемый **режим реального времени**, при котором пользовательская программа получает управление в течении гарантированного (и, как правило, достаточно небольшого) времени после возникновения того или иного внешнего события. Такой режим наиболее важен для вычислительных систем, управляющих промышленным или исследовательским оборудованием.

Иртегов с.31

Системы реального времени (real time) — системы, которые оптимизируют (или, точнее, гарантируют) максимальное время реакции на внешнее событие. Такие системы используются, главным образом, в неинтерактивных приложениях. Дело в том, что пользователя-человека неожиданная задержка при выполнении его запроса может раздражать, но непосредственной опасности эта задержка представлять не может. Напротив, задержка при обработке сигнала от установленного на летательном аппарате датчика опасности столкновения может привести к катастрофическим последствиям.

Подробно про системы разделенного времени Иртегов с.31

Там же есть преимущества и недостатки по сравнению с ПК

Переключение процессов по таймеру в сочетании с некоторыми другими приемами, такими как **динамическая приоритизация** (рассматривается в разд. 8.2.7), позволяет реализовать параллельную работу нескольких пользователей или пользовательских задач так, что каждый пользователь получает определенную долю процессорного времени. Со многих точек зрения каждый из этих пользователей может считать, что ему доступен свой собственный компьютер, производительность которого составляет определенную долю от реального.

... Следовательно, хотя многопоточность обычно не может улучшить время обработки одного запроса, она вполне способна в несколько раз увеличить количество запросов, обрабатываемых в единицу времени, — а это именно

то, что требуется от сервера приложений.

Отличия от Систем Разделенного Времени (by Inkpv):

- В системах разделенного времени основной упор делается на максимизацию пропускной способности и обеспечение справедливого доступа к ресурсам для всех пользователей.
- В системах реального времени приоритет отдается соблюдению временных ограничений, даже если это приводит к менее эффективному использованию ресурсов.

Другое описание отличия систем (перевод)

<https://www.geeksforgeeks.org/difference-between-time-sharing-os-and-real-time-os/>

Операционная система с разделением времени позволяет выполнять программы одновременно посредством быстрого переключения, тем самым предоставляя каждому процессу одинаковое количество времени для выполнения. В этой операционной системе доступен метод / функция переключения. Это переключение невероятно быстрое, так что пользователи смогут выполнять свою программу так, как будто запущена только эта программа, не зная, что система является общей.

Операционная система реального времени невероятно полезна для приложений с временным порядком, другими словами, везде, где задачи должны быть выполнены в определенные сроки. Период времени в операционных системах требует не только правильных результатов (на самом деле это не обязательно, если под правильным результатом мы подразумеваем **успешное выполнение**, допустима ситуация и завршения с ошибкой, но главное, чтобы за гарантированное время - Кирилл), но и своевременных результатов, что подразумевает, помимо корректности результатов, что должно быть создано в предельно сжатые сроки, иначе система может выйти из строя.

Основное различие между операционной системой с разделением времени и операционной системой реального времени заключается в том, что в ОС с разделением времени ответ предоставляется пользователю в течение секунды. В ОС реального времени ответ предоставляется пользователю в течение ограниченного времени.

Итегров с.36

Системы реального времени

Многие системы разделенного времени обеспечивают удовлетворительно среднее время реакции на внешние события, однако есть приложения, для которых необходимо гарантированное максимальное время такой реакции. Иными словами, ОС, которая в среднем обеспечивает реакцию на событие в течении 0,1 миллисекунды, но допускает— пусть даже изредка— увеличение этого времени до 0,1 секунды, для таких приложений неприемлема. Большинство этих приложений связаны с управлением промышленным и исследовательским оборудованием, а также с управлением бортовым оборудованием транспортных средств и космических аппаратов.

Впрочем, при разработке системы реального времени главной проблемой оказываются не "плохие" в указанном ранее смысле алгоритмы, а тот факт, что многие операции ввода/вывода занимают очень большое время. Ядро системы реального времени должно уметь вытеснять само себя и передавать управление пользовательским задачам во время исполнения операций ввода/вывода. Проблемы, к которым это приводит, и решения, применяемые на практике, обсуждаются в **разделе 8.2.3 (с.464 в пдф)**. Так или иначе, архитектура наиболее успешных систем РВ, таких как **QNX** и VxWorks, достаточно сильно отличается от наиболее распространенных систем разделенного времени.

Архитектура

Насчет архитектуры в вопросе билета не совсем понимаю, что от нас хотят, так что вот два варианта:

1. Системная архитектура QNX

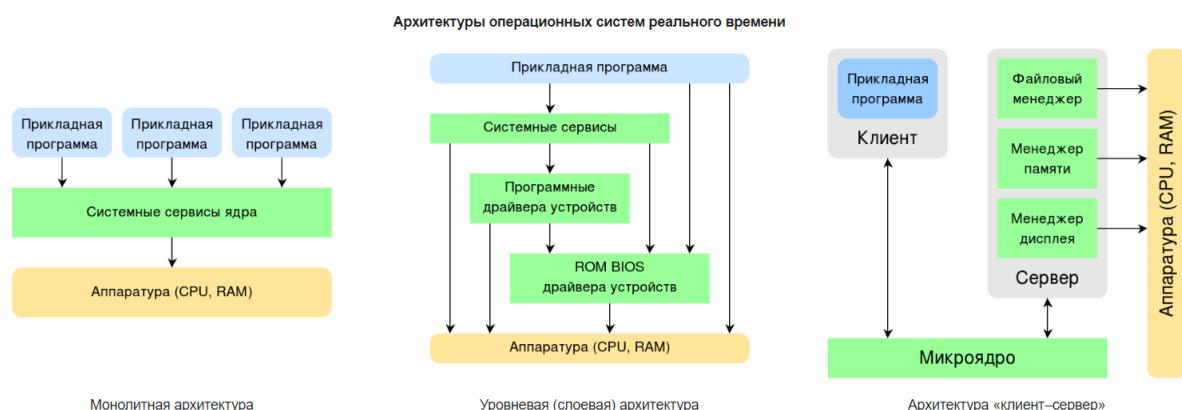
Концепция QNX. Главная обязанность ОС состоит в управлении ресурсами компьютера. Все действия в системе – диспетчеризация прикладных программ, запись файлов на диск, пересылка данных по сети и т.п. – должны выполняться совместно слитно и прозрачно. ОС QNX обеспечивает все неотъемлемые составляющие СРВ: **многозадачность, диспетчеризацию программ на основе приоритетов и быстрое переключение контекста.** Конфигурация QNX изменяется от ядра с несколькими небольшими модулями до полноценной сетевой системы, обслуживающей сотни пользователей. QNX достигает своего уникального уровня производительности, модульности и простоты благодаря двум фундаментальным принципам:

- архитектура на основе микроядра;
- связь между процессами на основе сообщений.

2. Архитектура ОС реального времени (https://ru.wikipedia.org/wiki/Операционная_система_реального_времени#Архитектуры_ОСРВ)

В своём развитии ОСРВ строились на основе следующих архитектур:

- Монолитная архитектура.** ОС определяется как набор модулей, взаимодействующих между собой внутри ядра системы и предоставляющих прикладному ПО входные интерфейсы для обращений к аппаратуре. Основной недостаток этого принципа построения ОС заключается в плохой предсказуемости её поведения, вызванной сложным взаимодействием модулей между собой.
- Уровневая (слоевая) архитектура.** Прикладное ПО имеет возможность получить доступ к аппаратуре не только через ядро системы и её сервисы, но и напрямую. По сравнению с монолитной такая архитектура обеспечивает значительно большую степень предсказуемости реакций системы, а также позволяет осуществлять быстрый доступ прикладных приложений к аппаратуре. Главным недостатком таких систем является отсутствие многозадачности.
- Архитектура «клиент–сервер».** Основной её принцип заключается в вынесении сервисов ОС в виде серверов на уровень пользователя и выполнении микроядром функций диспетчера сообщений между клиентскими пользовательскими программами и серверами — системными сервисами. Преимущества такой архитектуры:
 - Повышенная надёжность, так как каждый сервис является, по сути, самостоятельным приложением и его легче отладить и отследить ошибки.
 - Улучшенная масштабируемость, поскольку ненужные сервисы могут быть исключены из системы без ущерба к её работоспособности.
 - Повышенная отказоустойчивость, так как «зависший» сервис может быть перезапущен без перезагрузки системы.



Также про системы РВ (только мельком нашла у Иртегова, подробно было на [википедии](#))

Операционные системы реального времени иногда делят на два типа — системы жёсткого реального времени и системы мягкого реального времени[4].

Операционная система, которая может обеспечить требуемое время выполнения задачи реального времени даже в худших случаях, называется **операционной системой жёсткого реального времени**. Система, которая может обеспечить требуемое время выполнения задачи реального времени в среднем, называется **операционной системой мягкого реального времени**.

Вопрос 2: Сборщик мусора Java HotSpot

Презентация Иртегова

Читать Иртегов с.261

Виртуальная машина Java HotSpot была реализована в версии java 1.2.2 и стала стандартной в Java 1.3. В этой VM используется **генерационный сборщик мусора** (читать про сборку мусора), предполагающий разбиение объектов на четыре поколения: **перманентные, старые, молодые и вновь создаваемые**. Структура пулла:



Перманентные объекты – то системные объекты JVM, срок жизни которых всегда равен времени исполнения программы. При нормальной работе системы они не подвергаются сборке мусора.

Пул молодых объектов состоит из двух частей, называемых Semispace 1 и Semispace 2 (полупространства 1 и 2, сокращенно SS1 и SS2)

Все вновь создаваемые объекты создаются в "эдеме". При заполнении эдема инициируется **малая** сборка мусора.

При первой малой сборке мусора SS1 и SS2 пусты. Все используемые объекты из эдема перемещаются в SS1; все, что осталось в эдеме, объявляется мусором, эдем очищается.

При второй малой сборке мусора SS1 занят, но SS2 пуст. Используемые объекты из SS1 и эдема перемещаются в SS2, оставшееся содержимое эдема и SS1 очищается.

Объект, определенное число раз перемещавшийся между SS1 и SS2, признается **долгоживущим** (tenured) и перемещается в пул старшего поколения.

Соответствующий параметр не задается явно; вместо этого система динамически подбирает порог "зрелости" объекта, пытаясь за счет этого обеспечить определенный уровень заполнения пространства выживших.

Вместе с каждым долгоживущим объектом перемещаются и все объекты, на которые он ссылался, даже если эти объекты находились в эдеме. Это делается, чтобы защитить соответствующие объекты от потери при последующих малых сборках мусора.

Данная стратегия обоснована только статистикой исполнения реальных программ и гипотезой, что долгоживущие объекты не ссылаются на короткоживущие, однако практика подтверждает, что обычно она работает довольно хорошо.

При заполнении пула старшего поколения инициируется **большая** сборка мусора. Эта сборка затрагивает только объекты старшего поколения и производится в два этапа. На первом этапе сборщик помечает используемые объекты, как при обычном так mark and sweep. На втором этапе происходит дефрагментация — все используемые объекты копируются в начало пула. В зависимости от настроек JVM, после большой сборки мусора может быть запущена малая.

Таким образом, система имеет следующие параметры настройки:

- ms - начальный размер пула, который соответствует сумме размеров эдема и полупространств среднего поколения SS1 и SS2
- semispaces или SurvivorRatio - размер полупространств среднего поколения. Этот параметр может задаваться как абсолютным значением (semispace), так и относительно начального размера пула(SurvivorRatio).
- TargetSurvivorRatio - доля (в процентах) заполнения пространства выживших.
- Размер пространства старшего поколения. В параметрах командной строки вместо этого указывается общий максимальный размер пула mx. Пространство старшего поколения соответствует разности между максимальными и минимальными размерами пула.

Есть способы задавать эти параметры друг относительно друга; чаще всего используется параметр NewRatio, задающий отношения объемов старого и молодого поколения (сумма объемов эдема и полупространств). Так при NewRatio = 2 под пул молодого поколения занимается 1/3 памяти, а пул старого 2/3.

Пользователь может подбирать эти параметры самостоятельно, наблюдая за работой приложения, — для этого можно заставить JVM выдавать довольно подробную статистику работы сборщика мусора, в том числе:

- среднее и максимальное время работы сборщика (с разбивкой на малые и большие сборки);
- общий уровень заполнения пула и его разбивка по поколениям;
- порог "зрелости" объекта и т. д.

По умолчанию JVM реализует два режима: клиентский и серверный.

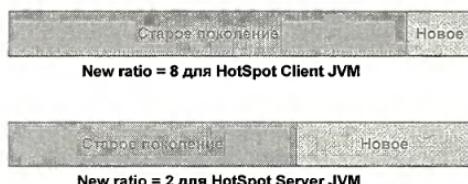


Рис. 4.17. Относительные размеры пулов младшего и старшего поколения в разных версиях HotSpot JVM

В клиентском режиме система выделяет 1/9 пула под новое поколение объектов и, соответственно, 8/9 под старое поколение. При этом малые сборки мусора происходят очень часто; клиентские программы обычно работают недолго, и потому до большой сборки мусора может вообще не дойти.

Для серверного режима под новое поколение выделяется 1/3 пула, а под старое — 2/3. Это приводит к значительному перераспределению времени между малыми и большими сборками мусора: малые сборки происходят реже, а большие — чаще, но зато каждая большая сборка занимает меньше времени.

Билет 6

Вопрос 1: Журнальные файловые системы. Принципы работы. Для чего это нужно?

Вопрос 2: Почтовые ящики (mailbox) в VAX/VMS

Билет 7

Вопрос 1: Семафоры Дийкстры. Мутексы, двоичные семафоры и семафоры общего вида. Мертвая блокировка и способы избежать ее.

По главе 7 "Параллелизм с точки зрения программиста"

В начале вводятся базовые понятия вроде целостности и как она может нарушаться, а также race condition.
Вставлю тут инфу из конспекта:

Race conditions (ошибки соревнования, "гонки"):

- Разные наложения параллельно работающих сопрограмм будут давать разный результат
- Если некоторые из результатов получились ошибочными, то мы и получили ошибку соревнования (важно, что именно некоторые результаты, потому что при ошибке при всех вариантах наложения это будет уже не race condition)
- Очень болезненно отлавливать такую ошибку тестированием

*Пример - покупка одного и того же билета несколькими пользователями**

Источником такой ошибки зачастую являются совместные данные, используемые в параллельно работающих участках кода.

Целостность или **согласованность** данных – неформализуемое понятие, которое включает в себя требования предметной области (сходится дебит и кредит, общая сила действия равняется сумме приложенных сил) и требования алгоритмов обработки:

- Записи должны иметь уникальный ID
- Строки должны заканчиваться нулями
- Массив должен быть отсортирован
- Дерево должно быть сбалансировано

Далее речь пойдёт о нитях исполнения процессов, которые в действительности могут быть вообще физическими процессами (перемотка ленты, биологические процессы в человеке-пользователе и т.п.)

Нить – то, что выполняется параллельно, но для чего создаётся иллюзия последовательного управления.

Дальше по книге

Асинхронное взаимодействие – взаимодействие между нитями.

Если нить работает с объектами, состояние которых не может быть изменено другими нитями либо объекты вообще не подвергаются модификации, проблем не будет. Проблемы начнутся в случае, если разделяемый несколькими нитями объект подвергнется модификации

КРИТ

Критическая секция - интервал, в течение которого нить модификацией нарушает целостность разделяемых данных, либо интервал, в течение которого нить полагается на целостность разделяемых данных

Таким образом, есть 2 пути написания корректного многонитевого кода: искоренение критических секций либо создание условий, при которых 2 нити не попадут в критические секции, связанные с одними и теми же разделяемыми данными. При невыполнении одного из этих вариантов, мы рано или поздно получим race condition.

Первый путь достаточно сложно реализуем и порой требует полной переработки логики программы. Второй путь - гарантит взаимоисключения - реализовать куда проще.

КРИТ

Транзакция - группа операций модификации разделяемой структуры данных, которая происходит атомарно, не прерываясь никакими другими операциями с той же структурой данных

ЛИБО (*первый вариант мне кажется более универсальным, но в критах дедов второе помечено зелёным*)

Транзакция - группа операций модификации разделяемой структуры данных, которые всегда либо происходят все вместе, либо не происходят вообще.

КРИТ

Реентерабельная программа - модуль, в котором нет критических секций либо для них гарантируется взаимоисключение, причём взаимоисключение не может быть нарушено никаким способом (в т.ч. сигналами). В современной литературе их называют *thread-safe*

Далее идёт инфа про критическую секцию из флага и спинлоки

Мёртвые и живые блокировки

При использовании любых механизмов взаимоисключения для доступа к различным разделяемым ресурсам возникает опасность мёртвой блокировки

КРИТ

Мёртвая блокировка (*deadlock*) - возникновение замкнутого цикла в графе ожидающих друг друга задач

Более человечно, но уже не на крит: ситуация, в которой несколько нитей заблокировались в своих критических секциях таким образом, что для разблокировки им необходима разблокировка хотя бы одной из них, что невозможно. Для возникновения достаточно, чтобы в одной нити поднимался сначала флаг взаимоисключения A, а затем внутри флаг B, а в другой нити - наоборот. Таким образом, может возникнуть ситуация, когда обе нити подняли первые флаги (A и B соответственно), но не могут пройти дальше, так как их вторые флаги уже подняты (B и A соответственно), а значит не могут снять свои первые поднятые флаги (A и B соответственно). Дэдлок двух нитей также может блокировать и множество других нитей до тех пор, пока один из флагов не будет опущен вручном режиме. В общем, ситуация страшная.

Одним из решений проблемы могли бы быть сообщения о возможности получения дэдлока, однако возникает вопрос, что в таком случае делать? Если 2 нити, получившие такое сообщение, вновь попытаются захватить ресурс, они снова получат то же сообщение - получается холостой цикл, называемый *живой блокировкой* (*livelock*), который на практике может быть даже большей проблемой, чем дэдлок, так как постоянно потребляет ресурсы ЦП.

По этой причине реакций на сообщение о возможном дэдлоке должно быть освобождение всех ресурсов нитью и повторная попытка их захвата через случайный интервал времени.

Есть ещё 2 альтернативных решения:

- Разрешить нити захватывать лишь один ресурс, что просто нереализуемо во многих ситуациях
- Проводить захват всех ресурсов (установку всех флагов) как транзакцию

Все 3 решения имеют одну серьёзную проблему - необходимость при попытке захвата ресурса обладать информацией обо всех ресурсах, которые могут понадобиться нити во время удержания блокировки.

Более того, все эти решения будут куда хуже работать при наличии трёх и более нитей, которые работают с одними и теми же ресурсами. Вплоть до того, что 2 нити могут словить дэдлок (можно описать проблему голодного философа со стр. 389)

Примитивы синхронизации с ожиданием

Примитивы взаимоисключения в виде спинлоков будут создавать проблему холостых циклов, схожую с той, что возникает при опросах событий, однако механика прерываний в данном случае проблему не решит.

Решением задачи будет синхронизация асинхронных нитей. Для этого нам необходим ещё один примитив, позволяющий нити остановиться в ожидании, пока флаг примет правильное значение. Однако тогда мы получаем проблему вечного сна: если проверка и засыпание - не атомарная операция, то флаг может поменять своё значение до засыпания, о чём процесс уже не узнает, уйдя в вечный сон. Решением стало объединение операции проверки флага и засыпания в, собственно, атомарную операцию.

Семафоры и мутексы

КРИТ

Семафор Дейкстры - целочисленная переменная, с которой ассоциирована очередь ожидающих нитей

Для крита этого по идеи достаточно. Дальше буду пояснять, как он работает

Пытаясь пройти сквозь семафор, нить вычитает из его значения единицу. Если после этого значение семафора ≥ 1 , нить проходит сквозь семафор (продолжает исполнение), если значение равно нулю, нить засыпает и становится в очередь. Если семафор закрыт, значит связанный с ним объект захвачен. Выходя из критической секции, нить увеличивает значение семафора на один, открывая его. Если в очереди семафора были нити, первая из них пробуждается и действует по тому же алгоритму. Если очередь пуста, семафор остается открытым и всё.

Значения семафора всегда неотрицательные. Двоичный семафор, также называемый мутексом, предназначен для разделяемых ресурсов, с которыми допустима работа только одной нити. Семафор общего вида может принимать любые значения. Нитям разрешено в таком случае прибавлять и отнимать отличные от единицы значения. Предназначен для случаев, когда с разделяемым ресурсом допустима работа нескольких нитей (например, несколько одинаковых принтеров)

Также в некоторых системах есть вариант неблокирующего обращения к семафору, которое в случае, когда при классическом варианте произошла бы блокировка, вернёт ошибку.

Для мутексов также есть одно дополнительное требование блокировку и снятие блокировки должна проводить одна и та же нить. С этой целью при обращении к мутексу также может передаваться информация для идентификации нити.

На странице 396 описывается задача производитель-потребитель, которая почти что напоминает трубы, но тут особого интереса не представляет. Скажу лишь, что для её решения обязательно нужен семафор общего вида (ну либо вообще что-то отличное от семафоров)

Флаги событий RSX-11 и VMS (AST)

В этих системах основным средством синхронизации являются флаги событий, которые могут возводиться (set) и очищаться (clear). Флаги делятся на:

- Локальные - используются для взаимодействия между процессом и ядром
- Глобальные - между процессами

Процесс может остановиться в ожидании взведения определённого флага, что делает их похожими на двоичные семафоры. Также процесс может связать с флагом события процедуру-обработчик AST (*asynchronous system trap*). которые во многих ситуациях очень удобны и напоминают обработчики сигналов (опасно говорить) и прерываний. Использоваться AST может, например, для исполнения асинхронных запросов на ввод-вывод:

- Программа создаёт локальный флаг и посыпает запрос
- Засыпает в ожидании этого флага либо продолжает исполнение
- Если исполнение было продолжено без засыпания, программа всё равно сможет узнать о том, что запрос был выполнен

Блокировка чтения-записи

Чтение немодифицированных данных не создаёт критической секции, однако в ходе чтения может произойти их модификация, что уже чревато. Чтобы избежать таких ситуаций, были введены 2 режима захвата:

- **Захват для чтения (read lock)** - разрешает другим нитям читать заблокированную структуру, но не разрешает её модифицировать
- **Захват для записи (write lock)** - запрещает другим нитям читать и писать в захваченную структуру (по своей логики похоже на мутекс)

Для обоих захватов существует единый примитив – **захват чтения-записи (read-write lock)**. Как и у семафоров, у него есть очередь и он также может приводить к дедлокам (по принципу задачи производитель-потребитель (стр. 396)), но при преобладании чтения над записью и малом числе нитей может быть оправдан в использовании.

Однако, при большом количестве записей и нитей, может возникнуть ситуация “голодания по чтению” (нити на чтение ждут долго), из-за чего теряется преимущество над обычными семафорами. Если же мы будем пропускать операции на чтение вперед стоящих в очереди операций на запись, то получим “голодание по записи”.

Копирование при записи

Копирование при записи стало одной из попыток решить проблему голодания по чтению. Суть в том, что при запросе на модификацию создаётся копия объекта, которая и будет модифицироваться. Запросы на чтение в это время работают со старой версией объекта. Когда модификация завершена, происходит замена адреса на новый адрес объекта. Таким образом, новые запросы на чтения будут получать доступ к уже новой согласованной версии. Таким образом, мы можем обойтись уже без блокировки на чтение.

Блокировка на запись может казаться необходимой, однако в некоторых случаях используются более сложные сценарии со слиянием всех модифицированных копий в единый объект. ЯП с таким механизмом сложно реализуемы (вообще такие представить не могу), а вот с СУБД и системах контроля версий он получил широкое распространение.

На стр. 401 идёт про обработку транзакций в СУБД. Не считаю нужным отдельно на этом останавливаться

Захват участков файлов

Для работы с разделяемыми ресурсами со внутренней структурой данных, допускающей работу по частям, допустим захват отдельных участков данных.

Например, такие блокировки в Unix могут накладываться на отдельные участки данных в двух режимах:

- **Допустимый (advisory)** - не оказывает влияния на систему ввода-вывода, а просто говорит о том, что участок заблокирован, прося его не трогать
- **Обязательный (mandatory)** - запрещает физически доступ, но требует дополнительный расходов на реализацию

Оба режима могут быть как блокировками на чтение, так и на запись

Мониторы и серверы транзакций

Про сервера здесь говорить не буду, смотрите NetApp WAFL либо инфу про транзакции со стр. 401, упомяну про мониторы

Монитор ресурса – специальная нить, через которую все остальные нити обращаются к разделяемому ресурсу. Обеспечивает инкапсуляцию и довольно гибкий контроль над методом распределения ресурса и предоставления непротиворечивого его состояния.

Вопрос 2: Файловая система NetApp WAFL

Билет 8

Вопрос 1: Как реализуется многопоточность на однопроцессорной машине. Что такое контекст процесса? Какие особенности процессора влияют на скорость переключения процессов?

РЕКОМЕНДУЮ ПРЕЗЕНТАЦИЮ ОБЯЗАТЕЛЬНО ПОСМОТРЕТЬ

Иртегов с 431

8.1. Кооперативная многозадачность

По-видимому, самой простой реализацией многозадачной системы была бы библиотека подпрограмм, которая определяет следующие процедуры.

struct Thread;

В тексте будет обсуждаться, что должна представлять собой эта структура, называемая дескриптором нити

Thread * ThreadCreate(void (*ThreadBody)(void));

Создать нить, исполняющую функцию ThreadBody.

void ThreadSwitch();

Эта функция приостанавливает текущую нить и активизирует очередную, готовую к исполнению.

void ThreadExit();

Прекращает исполнение текущей нити.

Сейчас мы не обсуждаем методов синхронизации нитей и взаимодействия между ними (для синхронизации были бы полезны также функции void

DeactivateThread(); и void ActivateThread(struct Thread *);). Нас интересует только вопрос: что же мы должны сделать, чтобы переключить нити?

Функция

ThreadSwitch называется **диспетчером** или **планировщиком** (scheduler) и ведет себя следующим образом.

1. Она передает управление на следующую активную нить.
2. Текущая нить остается активна, и через некоторое время снова получит управление.
3. При этом она получит управление так, как будто ThreadSwitch представляла собой обычную функцию и возвратила управление в точку, из которой она была вызвана.

Очевидно, что функцию ThreadSwitch нельзя реализовать на языке высокого уровня, вроде С, потому что это должна быть функция, которая не возвращает [немедленно] управления в ту точку, из которой она была вызвана. Она вызывается из одной нити, а передает управление в другую. Это требует прямых манипуляций стеком и записью активизации и обычно достигается использованием ассемблера или ассемблерных вставок.

Самым простым вариантом, казалось бы, будет простая передача управления на новую нить, например, командой безусловной передачи управления по указателю. При этом весь описатель нити (**struct Thread**) будет состоять только из адреса, на который надо передать управление. Беда только в том, что этот вариант не будет работать

Действительно, каждая из нитей исполняет программу, состоящую из вложенных вызовов процедур. Для того чтобы нить нормально продолжила исполнение, нам нужно восстановить не только адрес текущей команды, но и **стек вызовов**. Поэтому мы приходим к такой архитектуре:

- каждая нить имеет свой собственный стек вызовов;
- при создании нити выделяется область памяти под стек, и указатель на эту область помещается в дескриптор нити;
- ThreadSwitch сохраняет указатель стека (и, если таковой есть, указатель кадра) текущей нити в ее дескрипторе и восстанавливает SP из дескриптора следующей активной нити (переключение стеков необходимо реализовать ассемблерной вставкой, потому что языки высокого уровня не предоставляют средств для прямого доступа к указателю стека);
- когда функция ThreadSwitch выполняет оператор return, она автоматически возвращает управление в то место, из которого она была вызвана в этой нити, потому что адрес возврата сохраняется в стеке

(Особенность процессора мб) Если система программирования предполагает, что при вызове функции должны сохраняться определенные регистры (как, например, С-компиляторы для 8086 сохраняют при вызовах регистры SI и DI (ESI/EDI в x86)), то они также сохраняются в стеке. Поэтому предложенный нами вариант также будет автоматически сохранять и восстанавливать все необходимые регистры.

Понятно, что кроме указателей стека и стекового кадра, **struct Thread** должна содержать еще некоторые поля. Как минимум, она должна содержать указатель на следующую активную нить. Система должна хранить указатели на описатель текущей нити и на конец списка. При этом **ThreadSwitch** переставляет текущую нить в конец списка, а текущей делает следующую за ней в списке. Все вновь активизируемые нити также ставятся в конец списка. При этом список не обязан быть двунаправленным, ведь мы извлекаем элементы только из начала, а добавляем только в конец.

Часто в литературе такой список называют **очередью нитей** (thread queue) или очередью процессов.

Нить в такой системе (как, впрочем, и в более сложных системах, рассматриваемых далее в этой главе) может находиться в одном из трех состояний:

1. Исполнение.
2. Ожидание исполнения.
3. Ожидание события.

Состояние ожидания исполнения соответствует готовой к исполнению нити, которая стоит в очереди активных нитей, пока процессор занят другой нитью.

В однопроцессорной системе исполняющаяся нить может продолжать находиться в голове очереди активных, однако в многопроцессорных системах планировщик вынужден извлекать активную нить из очереди, иначе есть риск, что одну и ту же нить запланируют на нескольких процессорах.

Ожидание события означает, что нить не готова к исполнению.

Количество одновременно исполняющихся нитей, как легко догадаться, не может превосходить количества процессоров в системе; в однопроцессорной системе такая нить может быть только одна. В условиях, когда ни одна нить не готова к исполнению, система обычно выполняет специализированную задачу, так называемую холостую нить (**idle task** или **idle thread**).

В кооперативных системах такая нить должна циклически вызывать TaskSwitch() для того, чтобы как можно скорее передать управление первой из нитей, которая перейдет в состояние ожидания исполнения.

Далее здесь [вопросы про кооперативную и вытесняющую многозадачность](#).

Какие особенности процессора влияют на скорость переключения процессов?

Возможно тут хотят про беды с сохранением контекста. Иртегов с 441

Теоретически, довольно легко представить себе процессор, на котором сохранение контекста невозможно. Для этого достаточно, чтобы какой-то из регистров, влияющих на исполнение программы (возможно, но не обязательно, счетчик команд или слово состояния процессора), нельзя было явным образом сохранить или загрузить. У **суперскалярных** процессоров может возникать более сложная проблема, связанная с динамическим отображением физических регистров на логические регистры. При этом часть логических регистров имеет одновременно несколько разных значений. С практической точки зрения это означает, что значение такого логического регистра не определено.

Сам по себе процесс спекулятивного опережающего исполнения означает, что команды исполняются не в том порядке, в котором закодированы, — т. е. при этом не определено значение счетчика команд.

Такой процессор мог бы использоваться под управлением однопоточной или кооперативно многозадачной ОС, но вытесняющую многозадачную систему на нем реализовать нельзя. Ни один такой процессор не имел коммерческого успеха, во всяком случае в качестве процессора общего назначения.

Как мы видели в главе 6, большинство реальных суперскалярных процессоров при прерываниях выполняют **сериализацию**, т. е. приводят все свои логические регистры (в том числе и логический счетчик команд) к определенным значениям. Поэтому планировщик, вызываемый по прерываниям, вполне может сохранять контекст процессора в том состоянии, в котором он был на момент прерывания.

Однако в вытесняющих ОС вызовы планировщика могут также происходить по инициативе текущей нити. При некоторых типах суперскалярного исполнения это также может приводить к неопределенным значениям логических регистров, так что перед сохранением контекста планировщику необходимо специально попросить процессор о сериализации.

У современных процессоров x86 сериализация происходит при исполнении всех команд, имеющих префикс lock (префикс монопольного захвата шины), а также при исполнении команды xchq и некоторых других команд, которые работают в режиме захвата шины даже без префикса lock. Поскольку спинлоки в ядре реализуются с помощью именно таких команд, разработчик ОС может убить одним ударом двух зайцев: проверяя спинлок, защищающий семафор и/или очередь активных процессов, планировщик одновременно просит процессор сериализоваться.

Впрочем, сериализация, как мы видели в главе 6, — это дорогая операция, которая может потребовать десятки и даже сотни тактов.

Дальше написано про Intel 860 выглядит как ответ на вопрос вообще.

Как правило, оказывается неудобным сохранять контекст именно в стеке. Тогда его сохраняют в какой-то другой области памяти, чаще всего в дескрипторе процесса. Многие процессоры имеют специальные команды сохранения и загрузки контекста. Для реализации вытеснения достаточно сохранить контекст текущей нити и загрузить контекст следующей активной нити из очереди. Необходимо предоставить также и функцию переключения нитей по их собственной инициативе, аналогичную ThreadSwitch в кооперативно многозадачных системах. Впрочем, эту функцию обычно совмещают с блокировкой нити на каком-либо примитиве синхронизации или средстве гармонического взаимодействия.

Вытесняющий планировщик с разделением времени ненамного сложнее кооперативного планировщика — и тот, и другой в простейшем случае реализуются несколькими десятками строк на ассемблере. В работе приводится полный ассемблерный текст приоритетного планировщика системы VAX/VMS, занимающий одну страницу.

Впрочем, планировщики, рассчитанные на многопроцессорные машины, часто бывают несколько сложнее, т. к. они, во-первых, должны защищать очередь процессов от копий планировщика, работающих на других процессорах (обычно для этого используются спинлоки), и, во-вторых, применяют различные эвристические приемы, направленные на то, чтобы по возможности планировать процесс на одном и том же процессоре. Это позволяет оптимизировать работу процессора с кэш-памятью, а на процессорах с регистровыми окнами, таких как SPARC, может заметно сократить количество перезагрузок окон.

Дальше про контексты современных процессоров

Не особо понятно, где ответ на вопрос, но, мне кажется, можно сказать, что скорость зависит от размера кванта, по которому переключается нить, но не уверен, что это особенность процессора. Хотя, думаю, чем выше

тактовая частота процессора, тем больше операций в секунду он может выполнить, тем меньше, скорее всего, будет квант.

Ещё можно написать про задержку прерываний, это есть в примере с intel 860. Ну и от процессора зависит, какие регистры надо сохранять при переключении контекста. Чем их больше, тем медленнее будет происходить переключение.

Вопрос 2: Формирование запросов на ввод/вывод в RSX-11, VMS, OpenVMS. Какие преимущества предоставляет этот метод?

Иртегов с 467

Синхронный и асинхронный ввод/вывод в RT-11, RSX-11 и VMS

Системный вызов ввода/вывода в этих ОС называется **QIO** (Queue Input/Output [Request] — [установить в] очередь запроса ввода/вывода) и имеет две формы:

асинхронную QIO и синхронную QIOW (Queue Input/Output and Wait — установить запрос и ждать [завершения]).

С точки зрения подсистемы ввода/вывода эти вызовы ничем не отличаются, просто при запросе QIO ожидание конца запроса выполняется пользовательской программой "вручную", а при QIOW **выделение флага события** и ожидание его установки делается системными процедурами пред- и постобработки.

Поскольку запрос QIO состоит только в постановке запроса в очередь, для его реализации необходимо защитить спинлоками только сам заголовок очереди (на VAX не требуется даже этого, потому что данный процессор имеет команду установки записи в очередь, которая исполняется в режиме монопольного захвата шины). В однопроцессорных системах на время исполнения этой операции можно просто запретить прерывания.

Благодаря этому, в RT-11 запросы на ввод/вывод можно было исполнять даже из обработчиков прерываний!

Иртегов с 611

Запросы к драйверу в VMS

В операционной системе VAX/VMS драйвер получает запросы на ввод/вывод из очереди запросов. Элемент очереди называется IRP (Input[Output] Request Packet — **пакет запроса ввода/вывода**). Обработав первый запрос в очереди, драйвер начинает обработку следующего. Операции над очередью запросов выполняются специальными командами процессора VAX и являются атомарными. Если очередь пуста, основная нить драйвера завершается. При появлении новых запросов система вновь запустит ее.

IRP содержит:

- код операции (чтение, запись или код SPFUN — специальная функция, подобная ioctl в системах семейства Unix);
- адрес блока данных, которые должны быть записаны, или буфера, куда данные необходимо поместить;
- информацию, используемую при постобработке, в частности, идентификатор процесса, запросившего операцию.

В зависимости от кода операции драйвер запускает соответствующую подпрограмму. В VAX/VMS адрес подпрограммы выбирается из таблицы FDT (Function Definition Table). Подпрограмма инициирует операцию и приостанавливает процесс, давая системе возможность выполнить другие активные процессы. Затем, когда происходит прерывание, его обработчик инициирует fork-процесс, исполняющий следующие этапы этого запроса.

Завершив один запрос, fork-процесс сообщает об этом процедурам постобработки (разбудив соответствующий процесс) и, если в очереди еще что-то осталось, начинает исполнение следующего запроса.

Иртегов с 622

Синхронный и асинхронный ввод/вывод в RSX-11 и VMS

Например, в системах RSX-11 и VAX/VMS фирмы DEC для синхронизации используется флаг локального события (local event flag). Как говорилось в разд. 7.3.1, флаг события в этих системах представляет собой аналог двоичных семафоров Дейкстры, но с ним также может быть ассоциирована процедура AST. [Дальше аналогично выше](#)

Вот про флаги событий Иртегов с 397

Флаги событий в RSX-11 и VMS

Так, например, в системах RSX-11 и VMS основным средством синхронизации **являются флаги событий** (event flags). Процессы и система могут **очищать** (clear) или **взводить** (set) эти флаги. Флаги делятся на **локальные** и **глобальные**.

Локальные флаги используются для взаимодействия между процессом и ядром системы, глобальные— между процессами. Процесс может остановиться, ожидая установки определенного флага, поэтому флаги во многих ситуациях можно использовать вместо двоичных семафоров. Кроме того, процесс может связать с флагом события

процедуру-обработчик AST (Asynchronous System Trap — асинхронно [вызываемый] системный обработчик).

AST во многом напоминают сигналы или аппаратные прерывания. В частности, флаги событий используются для синхронизации пользовательской программы с асинхронным исполнением запросов на ввод-вывод. Исполняя запрос, программа задает локальный флаг события. Затем она может остановиться, ожидая этого флага, который будет введен после исполнения запроса.

При этом мы получаем псевдосинхронный ввод-вывод, напоминающий синхронные операции чтения/записи в UNIX и MS DOS. Но программа может и не останавливаться! При этом запрос будет исполняться параллельно с исполнением самой программы, и она будет оповещена о завершении операции соответствующей процедурой AST.

Преимущества

Возможность как синхронного, так и асинхронного ввода вывода.

[Про критические секции](#)

Билет 9

Вопрос 1: Что такое гармонически взаимодействующие последовательные процессы? Средства для реализации этой дисциплины в существующих системах.

[Начало темы межпроцессного взаимодействия](#). Здесь пойдёт глава 7.4. "Гармонически взаимодействующие последовательные потоки"

Взаимоисключение доступа к разделяемым данным приводит к целому ряду проблем, начиная от создания дополнительных сущностей вроде семафоров и заканчивая падением производительности при большой области исключительного доступа либо возникновением race condition'ов при попытках уменьшить эту область.

Попыткой решить эти проблемы стала концепция гармонически взаимодействующих последовательных потоков (процессов).

КРИТ

Гармонически взаимодействующие последовательные процессы – процессы, при взаимодействии между которыми глобальные переменные и разделяемая память не используются.

А теперь подробнее по пунктам (у дедов в качестве ответа на крит приведён и этот вариант):

- Каждый поток (нить) – независимый программный модуль, для которого создаётся иллюзия чисто последовательного исполнения
- У нитей нет разделяемых данных

- Все обмены данных и взаимодействие между нитями происходят при помощи специальных примитивов, которые осуществляют и обмен данных, и синхронизацию (ключевое требование)
- Синхронизация без обмена данных лишена смысла, так как нити совершенно независимы

Важно понимать, что гармоническое взаимодействие не исключает критические секции из алгоритма. Оно лишь сосредотачивает их и примитивы взаимоисключения внутри примитивов гармонического взаимодействия.

Теоретически, можно получить дэдлок, замкнув гармонически взаимодействующие нити в кольцо, но такую проблему относительно легко заметить. На практике, гармоническое взаимодействие лишено и проблемы блокировок, и проблемы голодного философа благодаря тому, что нити при таком взаимодействии имеют дело не с самим ресурсом, а с копией его состояния (*лично мне не совсем понятно, что вкладывается в эту фразу и как это соотносится, скажем, с модификацией ресурса, так что я бы последнюю фразу просто не говорил*)

В разных системах реализуются разные примитивы гармонического взаимодействия: почтовые ящики и линии в RSX-11 и VMS, трубы в Unix, randevu в Ada, линки в транспьютере. (желательно прочитать вопросы о них всех, чтобы уверенно говорить этот абзац).

Примитивы бывают двухточечные (один приёмник, один передатчик) и многоточечные (много приёмников и передатчиков), причём последние могут быть синхронными (много и тех, и других) и асинхронные (один передатчик, много приёмников и наоборот).

Примитивы могут передавать неструктурированный поток байтов (потоковые) либо структурированные сообщения.

Могут быть:

- Синхронными - передатчик ждёт, пока все переданные данные не будут приняты
 - По своей природе двухточечные
- Буферизованными - данные складываются в буфер и могут быть прочитаны позднее
 - Может быть реализован на основании двух синхронных примитивов и нити-монитора буфера
 - Зачастую двухточечные, особенно если потоковые, хотя трубы Unix теоретически могут быть многоточечными (*почему-то тут говорится о том, что асинхронные с одним приёмником, но насколько я помню из теории, это не так*)
 - Очереди сообщений часто будут синхронными многоточечными
- С негарантированной доставкой - данные игнорируются, если приёмник не был готов принять сообщение
 - Такие потоковые примитивы достаточно абсурдны, а вот структурированные достаточно широко используются в низкоуровневых интернет-протоколах

Различные типы всех этих свойств по разному комбинируются, порождая разные примитивы.

Таблица 7.1. Примитивы синхронизированной передачи данных

Примитивы	Синхронные	Буферизованные
Потоковые	Линки (транспьютер)	Трубы (Unix), сокеты (TCP/IP)
Структурированные	Randevu (Ada)	Очереди сообщений

Примитивы гармонического очень хороши и удобны для написания программ, однако в некоторых случаях создают существенные накладные расходы по памяти (у нас должно быть по буферу для хранения данных на каждую нить и третий буфер для буферизованных примитивов) и по времени работы (для медленных каналов передачи данных, особенно если нити располагаются на разных машинах), поэтому в некоторых случаях, например при работе с большим объёмом данных, оказывается более оптимальным принять возможные риски и сложности написания программы и использовать разделяемую память (зачастую с мониторным процессом)

Для общего вопроса по гармоническому взаимодействию будто бы этого уже даже и достаточно, но продолжу тут же писать про конкретные примитивы (стр. 409)

Программные каналы Unix (трубы)

Стр. 409

Основное средство взаимодействия между процессами в ОС семейства Unix. Труба представляет собой поток байтов, который имеет начало (исток) и конец (приёмник). Нить, пытающаяся прочитать данные из пустой трубы, будет задержана. Нить пытающаяся записать данные в заполненную трубу - тоже. На практике труба реализуется в виде небольшого кольцевого буфера. Можно использовать трубу в неблокирующем режиме, тогда в ситуациях, где должна была бы быть блокировка, будет возвращён особый код ошибки.

Чтение и запись осуществляются стандартными вызовами `read` и `write`, что даёт широкие возможности по переназначению ввода-вывода, заменяя дескрипторы `stdin` и `stdout`. Соединённые в цепочку трубами процессы будут называться конвейером.

Далее идёт инфа из конспекта, а не книги. Первый абзац сказать стоит, остальное - по усмотрению

`int pipe(int filedes[2])` - создаёт неименованный канал и записывает его дескрипторы в переданный массив. В стандарте UNIX первый дескриптор на чтение, а второй - на запись. В Solaris оба дескриптора двунаправленные и позволяют читать и писать перекрёстно. Используя `fork` и `dup` можно настроить взаимодействие через трубу для родственных процессов

Взаимодействие с программными каналами через разные системные вызовы:

- `open` - не нужен
- `close` - используется, когда работа с трубой закончена. Если мы закрыли дескриптор на запись, другой конец при чтении получит 'EOF', при попытке чтения. ***ЗАКРЫТЬ НЕОБХОДИМО ВСЕ КОНЦЫ ТРУБЫ****, причём самым первым делом надо закрыть концы, которые не будут использоваться в этом подпроцессе
- `read` - читает до заданного количества байт из трубы, блокируется, если труба пустая. Если читать будут сразу несколько процессов, то кому какие байты достанутся, совершенно неясно. После чтения те же байты прочитать из трубы невозможно
- `write` - пишет в трубу до заданного кол-ва байт и блокируется, если не может провести запись. Если конец на чтение уже закрыт, завершится с ошибкой 'EAGAIN', а процессу будет направлен сигнал 'SIGPIPE', который его по умолчанию аварийно завершает
- `lseek` и `mmap` - не допустимы
- `dup` - используется для перенаправления ввода-вывода в канал
- `fcntl` - используя его, можно установить чтение/запись без задержек, поменять размер буфера и, возможно, что-нибудь ещё

Снова к книге.

Для связи между неродственными процессами используются именованные трубы в System V и Unix domain sockets в BSD (второе говорить ой как опасно). Для создания можно использовать `mkfifo(char* name, mode_t flags)`. В результате будет создан специальный файл, открывая который, процесс получит доступ к одному из концов трубы.

И снова из конспекта, потому что там и системный вызов для создания другой, и доп. инфа есть:

`int mknod(const char *path, mode_t mode, dev_t dev)` - создаёт файл произвольного типа

При этом для не-'root' разрешено лишь создание FIFO-файлов (`mode = S_IFIFO | access_rights` - причём `mode_t`)

Системные вызовы для именованных труб:

- `open(2)` - блокируется, пока не будет открыт другой конец канала
- `close(2)` - когда доступ к каналу закончен
- `read(2)` - обычно блокируется, если канал пуст
- `write(2)` - обычно блокируется, если канал заполнен
- `lseek(2)` - не допустим

- `dup(2)` - используется для перенаправления ввода/вывода в канал
- `fcntl(2)` - может установить режим ввода/вывода без задержек

Для работы с несколькими трубами или другими примитивами предоставляются вызовы `select` и `poll`, которые позволяют получить список дескрипторов файлов, которые доступны для отправки или принятия данных. (тема этих системных вызовов *весьма обширна, поэтому оставлю тут ссылку на конспект*)

Почтовые ящики VMS

Стр. 413

Почтовые ящики во многом аналогичны трубам: представляют собой кольцевой буфер, доступ к которому осуществляется через те же системные вызовы, что используются для работы с внешними устройствами. (Да, чёрт побери, это всё. Больше ящики нигде не упоминаются, так что, видимо, дальше рассказывать надо будет про трубы!)

Линки в транспьютере

Стр. 413

Линки - примитив синхронизации, отчасти похожий на трубы (классно было бы тут избежать упоминания труб хотя бы в самом начале). Бывают физические и логические. Первые представляют собой интерфейс, реализованный прямо на кристалле процессора. К обоим видам линков доступ осуществляется одними и теми же командами.

Физические линки дают скорость передачи до 20 Мбит/с и могут использоваться как для связи нескольких транспьютеров, так и для связи транспьютера с внешними устройствами.

Логический линк - структура данных, расположенная в физической памяти. На программном уровне отличаются от физических только тем, что для физических адреса всегда фиксированные. Логические линки могут использоваться только для связи нитей (в терминах транспьютера они называются процессами) на одном транспьютере.

Для передачи данных по линку используется команда `out` с тремя operandами: адрес линка, адрес массива данных и его длина. Данные передаются при помощи стэка. Процесс, исполнивший такую команду, будет задержан до момента передачи данных (речь именно о получении данных на другой стороне)

Похожую семантику имеет команда `in`, которая позволяет прочесть данные из линка. Процесс будет заблокирован до момента прочтения указанного объёма данных. Приёмник и передатчик могут использовать буферы разного размера.

Команда `alt` позволяет ожидать данные из нескольких линков. В связанное с каждым линком слово будет копироваться указатель на дескриптор процесса, а уже в дескрипторе будут данные о буфере. В слове может также храниться особое значение `NotProcessP`, которое и позволяет процессам блокироваться, когда на другом конце линка нет другого процесса.

Линки никак не контролируют количество процессов, использующих их, поэтому процесс, попытавшийся записать данные в линк поверх данных другого записывающего процесса, успешно это сделает, при этом, если размер передаваемого буфера был одинаковым, второй процесс будет уверен, что он корректно передал данные. По этой причине рекомендуется использовать линки в одностороннем режиме для общения только между двумя процессами.

Физические линки в случае подключения к внешним устройствам не осуществляют копирования данных, происходит передача или приём в режиме прямого доступа к памяти (DMA)

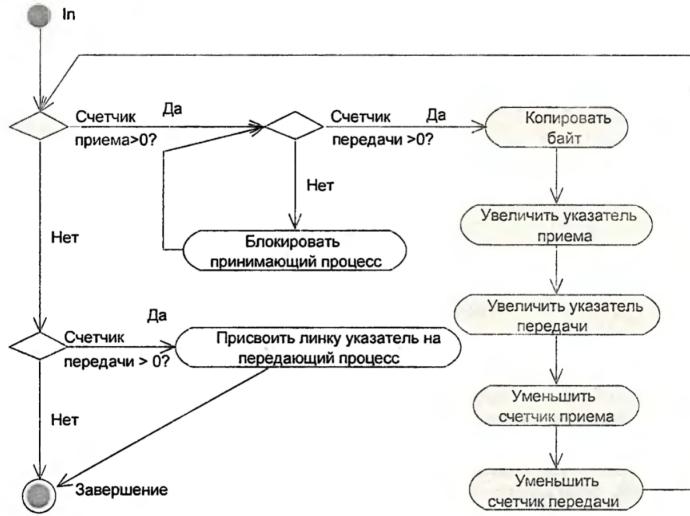


Рис. 7.9. Алгоритм работы команд `in` и `out`

Системы, управляемые событиями

Стр. 416 (архирекомендую прочесть из книги, потому что тема немного тяжкая)

Системы, управляемые событиями (событийно-ориентированные, *event-driven systems*) - архитектура многозадачных систем, в которой программа понимается как совокупность объектов, обменивающихся сообщениями о событиях и реагирующими на сообщения от других источников. При обработке сообщения объект изменяет своё состояния и, возможно, порождает новые сообщения. При такой модели взаимодействия нам не важно, исполняются ли нити параллельно (или псевдопараллельно) или и вовсе последовательно вызываются одной нитью - менеджером сообщений (по идеи, отсюда можно сделать вывод, что такая система не обязательно является многопоточной)

В первую очередь событийно-ориентированная архитектура использовалась в подсистемах ввода-вывода в ядре ОС, что позволяло одновременно обрабатывать потоки событий от нескольких устройств, не мешая друг другу. В некоторых системах событийно-ориентированная архитектура выносится на уровень системных вызовов - к моменту возврата управления прикладной программе запрашиваемая операция может ещё не завершиться, о её завершении станет известно благодаря похожим на семафоры механизмам (флаги событий RSX и VMS).

Реализация этой архитектуры на пользовательском уровне появилась в системах с многооконной графикой, среди событий в которой могли быть движения и нажатия мыши, сопровождающиеся необходимостью перерисовать окна, а также более классические события (нажатия на клавиатуру, системные таймеры, события от процессов).

Каждое сообщение о событии имело структуру данных с кодом события и некоторой дополнительной информацией, специфичной для разных событий.

Все сообщения помещаются в очередь в порядке их возникновения. Далее специальная нить - менеджер событий - просматривает очередь и передаёт сообщения в обработчик событий - особая структура данных, с которой связано несколько методов-обработчиков - вызывает необходимый метод, которые пошлют сообщения тем объектам, которые должны отреагировать на событие. Те в свою очередь, могут передать сообщения следующим объектам. В результате изменения состояния всех объектов, получивших сообщение мы, можем, например, увидеть открутое окно выбора файла для текстового редактора (событием было нажатие мыши на пункт Меню → Открыть).

В итоге вместо последовательно исполняющейся программы, вызывающей систему для некоторых действий, мы получаем набор обработчиков, вызываемых системой после определённых событий. Каждый отдельный обработчик можно легко представить в виде конечного автомата, иногда даже без переменной состояния.

Обработчики могут как синхронно вызываться менеджером событий (синхронная обработка сообщений), так и исполняться параллельно (асинхронная обработка событий).

В первый графических ОС обработка была синхронной, поэтому при долгой работе обработчиков они "зависали" (*песочные часы вместо мыши*).

Более сложный, но эффективный механизм (Presentation manager в OS/2) состоит в том, чтобы использовать также синхронный менеджер сообщений, но обработчики для сложных и долгих операций могут выделить отдельные нити, которые будут исполняться асинхронно основному процессу обработки сообщений. Если обработчик не выносит сложные операции в отдельную нить и приводит к зависанию всей очереди, менеджер может его принудительно отцепить от очереди или и вовсе завершить.

Асинхронные очереди сообщений (X Window, Win32) позволяют обрабатывать сообщения в разы эффективнее, однако даже при такой реализации зависший однопоточный обработчик сильно замедлит отрисовку.

Современные архитектуры обычно делятся на два или более слоя: пользовательский (frontend) часто использует событийно-управляемую архитектуру, тогда как более глубокие слои (backend) строятся на более традиционных архитектурах (впрочем, не всегда гармонических)

Один из примеров развития событийно-ориентированной архитектуры - это архитектура *рабочих нитей*, которая активно используется на серверах для обработки множества клиентских запросов. Суть заключается в заблаговременном создании набора нитей, которые будут осуществлять обработку событий по вышеописанному принципу, при этом каждая нить может обрабатывать сразу несколько клиентских запросов, оперируя объектами, связанными с каждой клиентской сессией. (здесь описание оставлю совсем куцое. Подробнее см. стр. 426-428)

Вопрос 2: Организация страничного обмена в VMS, OpenVMS и Windows NT

Здесь пойдёт глава 5.5 "Страницный обмен". [Начало главы](#)

Подкачка или свопинг - процесс выгрузки резко используемых областей виртуальной памяти на диск или другой ПЗУ (здесь говорится "устройство массовой памяти", но мне совершенно не нравится эта фраза)

Существует несколько эмпирических правил, которые делают целесообразным такой вид оптимизации, ведь быстрая память стоит дороже, а нужна далеко не всем программам одновременно. Среди этих правил есть 80-20: "80% транзакций работают с 20% файла" и 90-10: "90% кода исполняется 10% времени" (второе не имеет прямого отношения к свопингу и не совсем применимо тут по значениям, но суть у него похожая - мы можем 90% кода исполнять на медленном языке с более удобным синтаксисом, а 10% - на сложном языке, но быстром)

Миграцию с жёстких дисков на гибкие (или другие совсем-совсем медленные и дешёвые носители) обычно проводят вручную. Миграция из ОЗУ в ПЗУ обычно проходит автоматически из-за трудоёмкости. Миграция из кэша процессора в ОЗУ проходит только автоматически.

Поиск жертвы

Для свопинга желательно выбрать самые подходящие области памяти (жертву). Для этого есть множество стратегий. Самыми простыми из них будут:

- Выбирать объект, который не был изменён за время пребывания в быстрой памяти (в этом случае нам достаточно удалить его из памяти, а не выгружать на диск)
- Выбирать объект для свопинга случайным образом при условии, что мы обеспечили достаточно хорошее распределение для генератора псевдослучайных чисел
- Использовать стратегию FIFO и подвергать свопингу в первую очередь те объекты, которые попали в память раньше всего

Поиск жертвы в VAX/VMS и Windows NT/2000/XP

В этих системах используется модифицированная версия поиска жертвы через очереди. Страница-кандидат в жертвы забирается у процесса (в дескрипторе выставляется бит отсутствия (это уточнение можно сказать, если охереть как охота пояснять про дескрипторы и отвечать **КРИТ** по ним)), но не отдаётся сразу под другие нужды, а помещается в пул свободных страниц, состоящий из трёх очередей (этот же пул используется для дискового кэша и может занимать до половины ОЗУ):

- Очередь модифицированных страниц, ждущих записи на диск. После записи страницы переходят во вторую очередь
- Очередь немодифицированных страниц
- Очередь свободных страниц, то есть тех, которые были освобождены процессом или стали свободными после его завершения

Кандидатом в жертвы с равной вероятностью могут стать модифицированные и немодифицированные страницы. Для запросов прикладных программ и буферов дискового кэша могут быть выбраны страницы только из второй или третьей очередей

При обработке страничного отказа (**КРИТ** про исключения), система сначала будет искать страницу среди этих очередей и при удаче сразу возвращать её процессу. И только в ином случае будет проводиться загрузка страницы с диска.

Такая стратегия увеличивает количество страничных отказов, но сильно сокращает количество обращений к диску.

В VAX/VMS для каждого пользователя задаётся квота на **рабочее множество** (набор страниц, который в данный момент действительно нужен программе) запущенных программ. При превышении этой квоты ОС начинает искать жертву среди страниц задач этого пользователя. При правильной балансировке квот система будет показывать высокую работоспособность даже при малом объёме ОЗУ.

Windows пытается регулировать пул свободных страниц динамически, из-за чего в момент высокой заполненности ОЗУ наблюдается сильное падение производительности

На самом деле в том или ином виде пул свободных страниц используют все системы со страничной адресацией, но обычно он значительно меньше по объёму и отделён от дискового кэша. Поиск жертв в таком случае начинается при падении размера пула ниже определённого значения

LRU и clock-алгоритм поиска жертвы

LRU (*Least recently used*) предполагает выбирать в качестве жертвы, к которой дольше всего не было обращений, но для этого требуется, хранить в дескрипторах дополнительную информацию (счётчик обращений, время или что-то ещё), что критически понижает эффективность обращений к памяти.

Во многих системах семейства Unix используется Clock-алгоритм, который является некоторым приближением к LRU и заключается в добавлении в дескриптор всего одного *clock*-бита:

- При обращении к странице со сброшенным *clock*-битом, диспетчер памяти устанавливает его
- При поиске жертвы, если у страницы пустой *clock*-бит, она объявляется жертвой. Если бит установлен, то он сбрасывается, а поиск жертвы продолжается

В современных реализациях *clock*-алгоритма используется ещё один бит в дескрипторе - бит модификации, который устанавливается при первой записи в страницу, где этот бит был сброшен

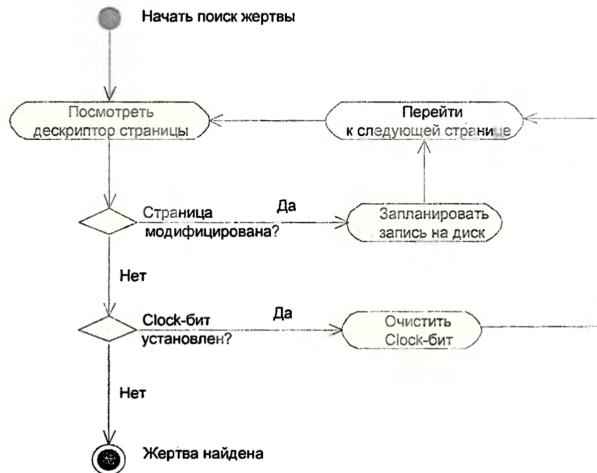


Рис. 5.21. Clock-алгоритм (блок-схема)

На практике выбор алгоритма поиска жертвы достаточно слабо влияет на производительность. Куда большую роль играет баланс между скоростью процессора, скоростью обмена с диском и потребностей в ПО. Рекомендуется подбирать количество памяти так, чтобы канал дискового обмена был загружен наполовину. Другое эмпирическое правило предлагает на каждый миллион операций в секунду иметь мегабайт ОЗУ (хотя в первом приближении оно и может применяться к ПК, но в целом достаточно древнее и мне не дало адекватных данных для ноута)

Управление своп-файлом

Вроде как эта инфа уже и ни в каком билете не нужна, но пусть будет

Для сохранения модифицированных страниц памяти, выбранных жертвами может использоваться либо особый своп-раздел на диске, либо своп-файл. При этом не подвергшиеся модификации страницы-жертвы туда не попадают. В случае стражничного отказа они будут загружены с того же места на диске, откуда были загружены в прошлый раз (например, если ОС использует абсолютную загрузку либо позиционно-независимый код (**КРИТ**)). При использовании относительной загрузки или той или иной формы сборки в момент загрузки страницы-жертвы могут либо выгружать в своп, либо каждый раз настраиваться заново

Говорят про использование `mmap` в режимах `MAP_SHARED` (изменения в памяти будут отражаться в файле) и `MAP_PRIVATE`, не буду это здесь приводить. Практику мы помним неплохо

Размер своп-файла может меняться динамически, однако ему можно задать верхнюю границу. При приближении к ней система будет выдавать предупреждения, а при достижении лимита может перестать выделять память прикладным процессам. Впрочем, в некоторых случаях память может выделяться без оглядки на заполненность своп-файла. Такая стратегия называется оптимистической алокацией или *overcommit* и широко используется в ОС семейства Unix (полезна при использовании `fork`, так как в реальности нам достаточно редко приходится реально копировать данные исходного процесса, ведь мы достаточно быстро вызываем `exec`)

Дальше идёт глава 5.7 про одноуровневую память (приравнивание по скорости ОЗУ и другой памяти), которая вот прям совсем не относится к темам билетов, но может быть интересна

Билет 10

Вопрос 1: Методы реализации виртуальной памяти. Базовая адресация, сегментная и страничная виртуальная память

[Презентация Иртегова](#)

По главе 5 "Страницная и сегментная виртуальная память" (стр. 281)

Концепция виртуальной памяти нацелена на решение целого ряда критически важных задач:

- Повышение безопасности за счёт того, что процесс может работать лишь в своей области памяти (защита процессов друг от друга и системы от процессов)
- Борьба с внешней фрагментацией за счёт возможности размещения логически непрерывные данные в несмежных областях физической памяти
- Дисковая подкачка для увеличения количества доступной памяти

За реализацию виртуальной памяти отвечает MMU (УУП - устройство управления памятью) - аппаратный модуль, стоящий на пути от процессора к ОЗУ и определённым образом обрабатывающий запросы для преобразования адресов.

Базовая адресация

Описывается в конце главы 4, стр. 273, а также в главе 3.4 на стр. 167, но сюда я взял инфу из конспекта, она мне показалось достаточно похожей

Базовая адресация - самый простой метод реализации виртуальной памяти, который заключается в передаче MMU двух значений (на самом деле базовая адресация может работать и без MMU ввиду своей простоты): адреса начала области памяти и её лимита. Если запрашиваемый адрес (фактически, сдвиг) больше лимита, то MMU выбросит исключение, в ином случае, добавит к базовому адресу сдвиг и предоставит доступ к получившейся ячейке физической памяти. Главная проблема этого метода - необходимость выделять непрерывную область памяти для процесса, а значит мы получаем фрагментацию. Шагом к решению стало выделение для процесса набора из пар база-лимит, что стало шагом на пути к концепции сегментной памяти.

Сегментная и страницчная адресация

Сегмент - область памяти, размер которой может задаваться произвольно, между сегментами могут быть "дыры" (возможно последнее предложение лучше не говорить). Страница - блок памяти фиксированного размера, занимающие пространство непрерывно. Таким образом, адрес в виртуальной памяти состоит из двух частей: селектора страницы или сегмента и смещения. Селектор позволяет получить дескриптор страницы/сегмента, который будет содержать физический адрес начала, права доступа: чтение, запись, исполнение, длину (для сегмента), флаг существования и некоторые другие параметры. (**2 КРИТА**) Благодаря данным из дескриптора и смещению мы можем получить доступ к физической памяти, ассоциированной с изначальным виртуальным адресом.

На старых машинах с малой разрядностью (PDP-11 16-bit) дескрипторы могли храниться прямо в регистрах MMU.

В машинах с разрядностью 32 бита количество дескрипторов возрастает многократно, поэтому они хранятся в таблицах трансляции, расположенных в ОЗУ. Регистр MMU в таком случае будет хранить указатель на эту таблицу. В большинстве MMU есть также кэш, называемый TLB (translation lookaside buffer), который содержит часто используемые дескрипторы. TLB организован как ассоциативная память (мапа) и состоит из 32-64 записей. Его существование позволяет сильно ускорить работу MMU в некоторых случаях, однако этот кэш достаточно глуп и в некоторых случаях (изменение дескриптора) кэш необходимо вручную сбрасывать.

В 64 разрядных машинах таблицы могут быть невероятно больших размеров, поэтому они не хранятся в ОЗУ. Вместо этого MMU сначала пытается найти дескриптор у себя в TLB и при неудаче выбрасывает исключение, которое обрабатывается особой процедурой в ядре, цель которой - найти или сочинить дескриптор. Под капотом там могут быть использованы самые различные методы от таблиц трансляции до хэш-таблиц.

Общий алгоритм доступа к памяти по виртуальному адресу будет таким (далее буду говорить о страницах, подразумевая такой же принцип для сегментов):

1. Проверить, существует ли страница по данному селектору, если нет, выбросить исключение segmentation violation
2. Попытаться найти дескриптор в TLB, если нет, получить дескриптор из таблицы

3. Проверить права доступа в дескрипторе. Выбросить исключение seg. violation, если прав недостаточно
4. Проверить, загружена ли страница по дескриптору в ОЗУ. Если нет, вызвать особое исключение, обработчик которого выгрузить страницу с диска в ОЗУ
5. Получаем физический адрес из дескриптора и (ДЛЯ СЕГМЕНТА) проверяем, не выходит ли смещение за длину сегмента
6. Произвести доступ к памяти

Виртуальная адресация даёт огромное количество преимуществ, часть из которых была описана выше, а часть относится к многопроцессным системам:

- Несколько процессов могут использовать одну и ту же область памяти для взаимодействия или как разделяемую библиотеку
- Каждому процессу может соответствовать своя таблица трансляции

Далее на страницах 288-289 говорится о проблемах виртуальной памяти для систем РВ (**КРИТ**) и о специфике выбора размера страниц и сегментов. Не считаю нужным переписывать это сюда.

Далее рассматривается вопрос взаимодействия пользовательского процесса и системы при виртуальной адресации. Думаю, он уже не совсем относится к вопросам, но Иртегов явно респектанёт, если об этом сказать (в книге огромный кусок на стр. 289-299)

Возникает вопрос, как должны работать системные вызовы с виртуальной памятью? Решения 2:

1. Хранить в диспетчере памяти несколько TLB с особыми идентификаторами для пользовательского и системного режима, а также использовать дополнительные регистры для контроля того, какой режим сейчас активен. Достаточно удобное решение, которое, однако, усложняет логику работы MMU
2. Хранить образ системы в виртуальном пространстве каждого процесса. Соответственно, TLB также будет одна, но системные участки будут защищены флагами в дескрипторах. Решение куда проще в реализации и не обладает особенно значительными недостатками, разве что несколько уменьшается адресное пространство, предоставляемое процессу

Вопрос 2: Программные каналы (трубы) в системах семейства Unix

Билет 11

Вопрос 1: Что такое абсолютный и относительный загрузчики? Структура абсолютного и перемещаемого загрузочных модулей. Что такое позиционно-независимый код?

По главе 3 "Загрузка программ"

Все системы программирования можно разделить на системы с **ранним (статическим)** связыванием и системы с **поздним (динамическим)** связыванием. Под связыванием подразумевается определение адресов именованных объектов.

При динамическом связывании адрес функции определяется в момент её вызова, из-за чего в разные моменты времени по одному и тому же имени могут вызываться и вовсе разные функции. Примеры языков с динамическим связыванием: BASIC, Java, C#, Perl

При статическом связывании программа считает, что адреса всех символьических объектов определены заранее. При этом реальная привязка ссылок может происходить на разных этапах: в момент компиляции или загрузки, но гарантировано до запуска программы. Примеры: ассемблер, Pascal, C, C++. За счёт того, что при таком связывании адреса могут непосредственно прописываться в поля команд, скорость работы зачастую возрастает. Для подмены исполняемых функций (ООПшная перегрузка) в некоторых языках со статической линковкой

используются особые структуры (`virtual` в C++, для этой же цели могут быть использованы указатели на функции в C)

Во всех языках со статическим связыванием после компиляции запускается процесс сборки или компоновки (linking), за который отвечает отдельная программа - линкер. В результате работы линкера мы получаем загружаемый модуль (executable), который содержит машинный код и, возможно, некоторую дополнительную информацию, необходимую загрузчику для связывания ссылок, которые не были связаны ранее.

Отдельно стоит выделить ленивое связывание, которое связывает символ в момент обращения к нему (как динамическое связывание), но после этого уже не изменяет ссылку (в этом разница с динамическим). Может использоваться вместо статического и с некоторыми предосторожностями вместо динамического. Пример - модули формата ELF.

Запуск задач в ОС семейства Unix

Если я не ошибаюсь, речь всё же идёт о процессах, что особенно прикольно, учитывая, как долго он распинался на стр. 157-158 о терминологии и заявил в итоге, что задачей процесс назывался лишь в старых ОС

В системах семейства Unix процессу создаются системным вызовом `fork`, который создаёт копию вызвавшего его процесса, отличающуюся только возвращаемым форком значением: 0 в новом процессе и pid нового процесса в старом. В системах со страницной или сегментной виртуальной памятью реально копироваться для нового процесса будут только изменённые страницы или сегменты (copy-on-write). Без виртуальной памяти будет необходимо полное копирование адресного пространства.

Для запуска другой программы используется системный вызов из семейства `exec`, который прекращает исполнение текущего образа процесса и загружает новый образ процесса, сохраняя pid, дескрипторы открытых файлов, uid, euid, pgid, sid

Абсолютная загрузка

Заключается в том, что мы всегда будем загружать программу с одного и того же адреса. Проще всего это реализуется в системах, обеспечивающих виртуальную адресацию и исполняющих в каждый момент только один процесс. Исполняемый таким образом загрузочный файл называется *абсолютным загрузочным модулем*. За счёт известного адреса загрузки все ссылки в нём разрешаются на этапе сборки, так как нам известно, на каких адресах будут расположены все объекты → дополнительной настройки на этапе загрузки не требуется. Формирование начального образа процесса происходит просто загрузкой модуля в память.

На странице 160-161 описывается формат абсолютных модулей .out, используемый в старых 32-битных ОС семейства Unix, который приведу совсем кратко:

- заголовок с информацией о длине участков модуля и стартовым адресом
- Области TEXT, DATA, BSS

На этапе загрузки выделяется память по очереди под TEXT (в режиме Read/Execute), DATA (RW), BSS (RW); BSS заполняется нулями, остальное, данными из модуля. В finale выделяется область памяти под стэк, в который помещаются позиционные аргументы и среда исполнения

Другой не особо примечательный момент - разделы памяти, позволяющий загружать несколько образов процессов (насколько я понимаю, речь о системах без виртуальной адресации, потому что в них такой проблемы нет). Суть в том, что выделяется набор адресов для абсолютной загрузки, эти адреса определяют идущие подряд разделы памяти. Модуль может быть загружен в один или несколько разделов, если он большой. Проблема - при абсолютной загрузке мы должны собирать по модулю для каждого раздела, что безумно неудобно

Относительная загрузка

Относительная загрузка состоит в том, что образ процесса каждый раз может размещаться начиная с разного адреса. При этом на момент компиляции и сборки этого адреса мы не знаем, а обладаем только информацией о сдвигах объектов относительно друг друга. При загрузке такого модуля мы должны настроить его новые адреса. Для этого необходимо найти адресные поля всех команд, использующих относительную загрузку, то есть такие

места, где в регистр загружается значение адреса (этую фразу говорить опасно, потому что она будто бы предполагает некоторое знание главы 2). Ощущимая сложность состоит в том, что определить, было ли в регистр загружено скалярное значение, или адрес, или часть будущего адреса, оказывается непростой задачей. Самое простое и единственное решение - содействие программиста, которое заключается в том, чтобы не использовать произвольные значения в качестве адресов, а чаще прибегать к символам, соответствующим адресам.

При обнаружении таких символов, ассемблер генерирует заготовку адреса в коде и **запись в таблице перемещений**

КРИТ:

запись в таблице перемещений хранит место ссылки на объект, являющийся ссылкой в коде или данных либо несколько таких ссылок, из которых будет сформировано итоговое смещение. (мдэм... как мне кажется, это формулировка лучше отражает суть, но звучит куда кривее формулировки прошлых курсов)

Файл, содержащий таблицу перемещений будет называться **относительным либо перемещаемым загрузочным модулем**. (Например, EXE файлы в MS-DOS).

Интересным примером комбинации двух типов загрузочных модулей будет система RT-11. В ней в адресном пространстве внизу размещаются вектора прерываний. По адресу 01000 загружается абсолютный модуль (расширение .sav, может быть загружен, что логично, только один). В конце адресного пространства располагаются драйвера и ядро ОС. Между драйверами и абсолютно загруженной программой могут быть расположены относительно загружаемые модули (расширение .rel)

Позиционно-независимый код

Позиционно-независимый код - код, в котором используется относительная адресация, при которой адрес получается путём сложения адресного поля с команды с адресом самой этой команды (то есть значения счётчика команд). Такой код можно загружать с любого адреса без перенастройки.

Такой код достаточно удобен, но сложен в написании: нельзя использовать статически инициализированные указатели, возникнут сложности со сборкой программы из нескольких модулей. Многие архитектуры процессоров вообще не поддерживают позиционно-независимый код (Intel 8080/8085, многие современные RISC-процессоры (наверное надо бы наконец прочитать, чем RISC отличается от CISC, раз уж на тех лекциях по ЦП я спал...))

Напишу здесь главу 3.6 "Оверлеи", хотя по ней билетов вроде нет

Оверлейная загрузка - способ загрузки программы, при котором она разделяется на модули (оверлеи), каждый из которых отображается в память по необходимости, при этом оверлеи могут перекрывать друг друга (отсюда и название). Позволяет отображать большое количество объектов в ограниченное адресное пространство. Замена оверлеев производится специальной программой - оверлейным менеджером, которая располагается в особой части памяти резидентном ядре вместе с некоторыми другими процедурами, которые нужны во многих оверлеях. Зачастую оверлеи применяются для самого кода, но не для данных. Оверлеи могут быть как абсолютными, так и относительными модулями, хотя структура из-за этого ощутимо усложняется

Вопрос 2: Семафоры Unix System V IPC. Наборы семафоров.

Иртегов с.873 в пдф

В 1987 году вышла версия UNIX System V Release 3, включавшая в себя асинхронные драйверы последовательных устройств (STREAMS), универсальный API для доступа к сетевым протоколам (TLI), **средства межпроцессного взаимодействия (семафоры, очереди сообщений и сегменты разделяемой памяти)**, ныне известные как SysV IPC, BSD-совместимые сокеты и ряд других "BSDизмов" [Робачевский 1999]. SVR3 в то время воспринималась как этапная ОС, однако дальнейшее развитие системы вынуждает нас отнести ее, скорее, к переходным версиям.

Подробно про семафоры - **7.3.1 Иртегов с.397 в пдф**

Виды семафоров:

- Двоичный семафор — мутекс

Наиболее простым случаем семафора является двоичный семафор. Начальное значение флаговой переменной такого семафора равно 1, и вообще она может принимать только значения 1 и 0. Двоичный семафор соответствует случаю, когда с разделяемым ресурсом в каждый момент времени может работать только одна нить.

- Семафоры общего вида — семафоры-счетчики

Семафоры общего вида могут принимать любые неотрицательные значения.

В современной литературе такие семафоры называют семафорами-счетчиками (counting semaphore). Это соответствует случаю, когда несколько нитей могут работать с объектом одновременно, или когда объект состоит из нескольких независимых, но равноценных частей — например, несколько одинаковых принтеров. При работе с такими семафорами часто разрешают процессам вычитать и добавлять к флаговой переменной значения, большие единицы. Это соответствует захвату/освобождению нескольких частей ресурса.

Подробнее про мутекс (Иртегов с. 398 в pdf)

Во многих современных книгах и операционных системах семафорами называются только семафоры общего вида, двоичные же семафоры носят более краткое и выразительное имя mutex — от MUTual Exclusion, взаимное исключение). Проследить генезис этого названия мне не удалось, но можно с уверенностью сказать, что оно вошло в широкое употребление не ранее конца 80-х годов XX века. Так, в разрабатывавшейся в середине 80-х годов OS/2 1.0, двоичные семафоры еще называются семафорами, а в Win32, разработка которой происходила в начале 90-х, уже появляется название mutex. Операции над мутексом называются захватом (acquire) (соответствует входу в критическую секцию) и освобождением (release) (соответствует выходу из нее).

В современных ОС термин mutex имеет дополнительный важный оттенок. А именно, предполагается что нить должна осуществлять операции захвата и освобождения парами: за захватом ресурса всегда должно следовать его освобождение. Недопустимы ситуации, когда одна нить захватывает ресурс, а другая его освобождает.

Благодаря этому для каждого заблокированного мутекса можно указать, какая именно нить его заблокировала. Пользуясь данной информацией, можно проверять, не приведет ли захват мутекса к мертвой блокировке и решать некоторые другие своеобразные проблемы, например проблему инверсии приоритета (см. разд. 8.2.3). Однако есть ситуации, когда требование парности операций оказывается слишком строгим для программиста.

Семафоры System V - кратко

[https://ru.wikipedia.org/wiki/Семафор_\(программирование\)#Семафоры_System_V](https://ru.wikipedia.org/wiki/Семафор_(программирование)#Семафоры_System_V)

Стандарты POSIX также определяют набор функций из стандарта X/Open System Interfaces (XSI) для межпроцессовой работы с семафорами в рамках операционной системы[60]. В отличие от обычных семафоров семафоры XSI можно увеличивать и уменьшать на произвольное число, они выделяются массивами, и их время жизни распространяется не на процессы, а на операционную систему. Таким образом, если забыть закрыть семафор XSI по завершении всех процессов приложения, то он продолжит существовать в операционной системе, что называется утечкой ресурса. В сравнении с семафорами XSI обычные семафоры POSIX намного проще в использовании, и у них может быть выше быстродействие[61].

Наборы семафоров XSI в рамках системы идентифицируются по числовому ключу типа `key_t`, однако можно создавать анонимные наборы семафоров для использования в рамках приложения, если указывать константу `IPC_PRIVATE` вместо числового ключа[62].

Функции для работы с семафорами XSI из заголовочного файла <code>sys/sem.h</code>	
Функция	Описание
<code>semget()</code> [док. 11]	Создает или получает идентификатор набора семафоров с заданным числовым ключом ^[62] .
<code>semop()</code> [док. 12]	Выполняет атомарные операции уменьшения и увеличения на заданное число счетчика семафора по его номеру из набора с заданным идентификатором, а также позволяет заблокироваться в ожидании нулевого значения счетчика семафора, если в качестве заданного числа указан 0 ^[47] .
<code>semctl()</code> [док. 13]	Позволяет управлять семафором по его номеру из набора с заданным идентификатором, в том числе получать и устанавливать текущее значение счетчика; также отвечает за уничтожение набора семафоров ^[63] .

Немного про семафоры <https://parallel.uran.ru/book/export/html/505>

Семафоры System V - подробно

(перевод; на этой же странице ниже есть описание функций `semget()`, `semop()`, `semctl()`)

<https://docs.oracle.com/cd/E19683-01/806-4125/6j7pe6bs/index.html>

Семафоры позволяют процессам запрашивать или изменять информацию о состоянии. Они часто используются для мониторинга и контроля доступности системных ресурсов, таких как сегменты совместно используемой памяти. Семафорами можно управлять как отдельными блоками, так и элементами в наборе.

Поскольку семафоры System V IPC могут находиться в большом массиве, они имеют чрезвычайно большой вес. Гораздо более легкие семафоры доступны в библиотеке потоков (см. Справочную страницу [semaphore\(3THR\)](#)). Кроме того, семафоры POSIX являются наиболее актуальной реализацией семафоров System V. (см. [Семафоры POSIX](#)). Семафоры библиотеки потоков должны использоваться с подключенной памятью (см. [Интерфейсы управления памятью](#)).

Набор семафоров состоит из управляющей структуры и массива отдельных семафоров. Набор семафоров может содержать до 25 элементов. Набор семафоров должен быть инициализирован с помощью [semget\(2\)](#). Создатель семафора может изменить своего владельца или разрешения, используя [semctl\(2\)](#). Любой процесс, имеющий разрешение, может использовать [semctl\(2\)](#) для выполнения управляющих операций.

Семафорные операции выполняются с помощью [semop\(2\)](#). Этот интерфейс принимает указатель на массив структур семафорных операций. Каждая структура в массиве содержит данные об операции, выполняемой с семафором. Любой процесс с разрешением на чтение может проверить, имеет ли семафор нулевое значение. Операции по увеличению или уменьшению размера семафора требуют разрешения на запись.

При сбое операции ни один из семафоров не изменяется. Процесс блокируется, если не установлен флаг IPC_NOWAIT, и остается заблокированным до тех пор, пока:

- Все операции с семафором могут завершиться, поэтому вызов выполняется успешно.
- Процесс получает сигнал.
- Набор семафоров удален.

Одновременно обновлять семафор может только один процесс. Одновременные запросы от разных процессов выполняются в произвольном порядке. Когда массив операций задается вызовом [semop\(2\)](#), обновления не выполняются до тех пор, пока все операции над массивом не завершатся успешно.

Если процесс, использующий исключительно семафор, завершается аварийно и ему не удается отменить операцию или освободить семафор, семафор остается заблокированным в памяти в том состоянии, в котором процесс оставил его. Чтобы предотвратить это, управляющий флаг SEM_UNDO заставляет [semop\(2\)](#) выделять структуру отмены для каждой операции семафора, которая содержит операцию, возвращающую семафор в его предыдущее состояние. Если процесс завершается, система применяет операции в структурах отмены. Это предотвращает то, чтобы прерванный процесс не оставил набор семафоров в несогласованном состоянии.

Если процессы совместно используют доступ к ресурсу, контролируемому семафором, операции над семафором не должны выполняться с включенным SEM_UNDO. Если процесс, который в данный момент контролирует ресурс, завершается аварийно, ресурс считается несовместимым. Другой процесс должен быть способен распознать это, чтобы восстановить ресурс в согласованном состоянии.

При выполнении семафорной операции с включенным SEM_UNDO у вас также должен быть включен SEM_UNDO для вызова, выполняющего операцию реверсирования. Когда процесс выполняется нормально, операция реверсирования обновляет структуру отмены дополнительным значением. Это гарантирует, что, если процесс не будет прерван, значения, примененные к структуре отмены, будут равны нулю. Когда структура отмены достигает нуля, она удаляется.

Непоследовательное использование SEM_UNDO может привести к утечкам памяти, поскольку выделенные структуры отмены могут быть не освобождены до перезагрузки системы.

Билет 12

Вопрос 1: Устойчивые к сбоям файловые системы. Методы реализации устойчивых ФС.

Точно также, как и ранее, сначала буду расписывать всё подряд про устойчивость ФС, а потом раскидаю по билетам (возможно XD)

Устойчивость файловой системы может рассматриваться в двух смыслах: как устойчивость к сбоям системы, из-за которых ФС не была корректно размонтирована (возможно конец предложения лучше не говорить, потому что точно описать, что из себя представляет "размонтирование" я пока что не могу), и устойчивость к физическим повреждениям носителя ФС.

Устойчивость к сбоям системы

К ним можно отнести:

- Извлечение носителя ФС
- Экстренная перезагрузка пользователем
- Фатальный аппаратный сбой и не фатальный аппаратный сбой, который система, тем не менее, не смогла обработать
- Программный сбой

В узком смысле устойчивость означает лишь то, что системе после сбоя при работе ФС не обязательно потребуется проводить процедуру восстановления. Более того, устойчивые ФС могут более-менее гарантировать целостность своих структур данных, но не пользовательских файлов (такую устойчивость нельзя обеспечить на уровне ФС, только в прикладных программах с большими ограничениями)

"Врождённая" устойчивость файловой системы FAT к сбоям обеспечивается тем, что обозначение блока занятым и назначение его файлу происходят за одной действие изменения таблицы. В таком случае при самом неприятном раскладе мы всего лишь рискуем потерять один блок диска, который был выделен по таблице, но в длине файла в каталоге длина меньше.

Однако устойчивость FAT (и любой другой ФС) сильно уменьшается при использовании отложенной записи, а она сейчас применяется повсеместно

Если же мы отдельно храним информацию о свободных блоках и блоках, выделенных для файлов, то могут возникнуть ситуации, когда мы либо только обозначили блок занятым, но не отдали его файлу, что не так страшно, либо только выделили блок файлу, но не обозначили его занятым. Эта ситуация уже куда опаснее и легко приводит смешиванию данных разных файлов.

Самый простой механизм восстановления ФС - использование dirty-флага, который при размонтировании ФС зануляется, а при монтировании либо первой модификации поднимается. Если система смонтировала ФС и в этот момент обнаружила dirty-флаг, она заключает, что необходимо провести восстановление ФС, в которое входят:

- Каждая запись в каталогах должна соответствовать формату и быть осмысленной (помеченная свободной запись не должна ссылаться на данные файла или инод)
- Обнаружения общих для файлов блоков. Чаще всего в такой ситуации восстановщик спрашивает у пользователя, к какому файлу относятся данные этого блока
- Проверка соответствия длины файла и количества его блоков. Если блоков больше указанной длины, лишние блоки освобождаются, если блоков меньше длины, то уменьшается длина
- Все блоки без входящих ссылок должны быть помечены как свободные, хотя в некоторых случаях среди таких потерянных блоков ищутся файловые записи, а найденные таким образом файлы помещаются в особый каталог для дальнейшей проверки пользователем

В Unix-системах могут возникнуть несколько inode-специфичных ошибок:

- Несоответствие счётчика ссылок их реальному количеству. Решается простой корректировкой счётчика
- Ноль ссылок, но инод не помечен, как свободный, значит это инод-сирота. Такие иноды помещаются в каталог lost+found
- Есть ссылки, но инод помечен как свободный, в таком случае ссылки на инод удаляются (то есть удаляются файлы). Также у нас есть серьёзные основания полагать, что был повреждён каталог, в котором были ссылки

на иноду

В UFS есть свой собственный dirty-флаг для каждого цилиндра, что позволяет проводить проверку целостности только для малой части данных на диске, для которой это действительно нужно.

Журналирование и журнальные ФС

Лекция 17 45 минута

Презентация

Идея журналов пришла из СУБД, где во многих сферах критически важна была устойчивость.

Основным решением стали транзакции - группа операций модификации разделяемой структуры данных, которая происходит атомарно (неделимо), не прерываясь никакими другими операциями с той же структурой данных.

Каждая транзакция осуществляется в 3 этапа:

- Запись желаемой транзакции в специальный файл (*журнал регистрации намерений*)
- Если запись была успешной, транзакция начинает выполняться
- Если транзакция была успешно завершена, система помечает её в журнале как успешную

Транзакция записи данных будет помечена как завершённая только после физической записи последнего блока на диск. В один момент чаще всего будет много незавершённых транзакций. Запись в журнал не может быть отложенной.

Общий принцип восстановления БД с журналами такой:

- Если запись в журнале некорректная, значит произошёл сбой на этапе записи в журнал. Запись можно отбросить
- Если все записи помечены как завершённые, значит сбой произошёл между транзакциями и беспокоиться не о чём
- Если обнаружена начатая, но невыполненная транзакция, значит сбой произошёл в процессе её выполнения. В этом случае в журнале достаточно информации либо для того, чтобы откатить изменения, либо для того, чтобы корректно их завершить

Возникает вопрос, что считать транзакцией для ФС? Тут есть 2 варианта: изменение метаданных и структур, связанных с самой ФС, либо всех данных внутри ФС.

Чаще всего выбирается первый вариант, хотя он уже и не будет гарантировать целостность пользовательских данных. Впрочем, даже целостность данных ФС он не может гарантировать стопроцентно, так как журнальные ФС в ходе проверки и восстановления полагаются на корректность журнала, не учитывая возможные ошибки в самой реализации ФС.

Из-за этого большое распространение получили совместимые ФС, которые поддерживают возможность журналирования, но могут использоваться и как традиционные. В традиционных проверка производится над всеми системными данными и сопровождается устранением ошибок сбоев и рассогласований из-за ошибок программирования.

JFS как пример журнальной ФС (многие моменты справедливы и для NTFS)

ФС может располагаться сразу на нескольких дисках, которые могут свободно добавляться и убираться без переразметки. Диски объединяются в агрегаты. За это отвечает LVM (Logical volume manager). На агрегате располагаются системные структуры данных и наборы файлов. Агрегат делится на блоки по степеням двойки от 512 до 4096 байт (значение по умолчанию)

В агрегате содержатся:

- Резерв 32 КБ
- Первичный суперблок агрегата на 4 КБ
- Вторичный суперблок - копия первичного

- Первичную и вторичную таблицу инодов
- Карту инодов агрегата
- Инод 0 зарезервирован, 1 содержит размещение кусков таблицы инодов (изначально известно лишь расположение первого куска таблицы)
- Инод 2 - размещение карты свободных блоков
- Инод 3 - размещение журнала транзакций
- Инод 4 - размещение дефектных блоков
- Иноды 5-15 резерв
- >16 - описание файловых наборов (при этом на данный момент в агрегате может быть всего один набор, поэтому иноды больше 16-го не используются)

Дальнейшая часть агрегата содержит файловые наборы и делится на логические группы, которые по количеству блоков кратны степеням двойки.

Описание файлового набора представляет собой таблицу инодов этого набора, которая размещается в экстентах, расположенных в B+-дереве. Из этого следует, что таблица динамическая. Иноды файлового набора:

- Нулевой зарезирован
- Первый хранит информацию о наборе
- Второй - каталог
- Третий - ACL - список управления доступом
- Последующие иноды хранят информацию об обычных файлах

JFS допускает сжатие динамической таблицы инодов как с конца, так и в середине, однако номера инодов будут сохраняться незиленными

В иноде содержатся базовые атрибуты файла по стандарту POSIX (права доступа, юзер, группа и т.д.), расширенные атрибуты ОС и ссылка на корень B+-дерева экстентов файла либо непосредственно данные специфических файлов (информация о символьской ссылке и т.п.)

Каждый экстент имеет свой описатель, который состоит из флага, сдвига экстента в логических блоках от начала файла, адрес первого блока экстента и длину экстента в логических блоках. Хотя экстенты и организованы в B+-дерево, на практике его корень часто является единственным листом, так как весь файл помещается в 2-3 экстента (а их в узле может быть 8)

Каталоги в JFS в B+-дереве содержат короткие префиксы и ссылки на полные имена файлов. Ввиду этой сложной структуры многократное удаление файлов приводит к частой ребалансировке дерева

Журнал может харниться на другом томе, так и здесь же. Транзакции записываются в журнал в следующих случаях:

- Создание/удаление/переименование файла
- Создание жёсткой ссылки
- Создание/удаление каталога
- Создание инодов особых типов (трубы и т.п.)
- Изменение ACL и расширенных атрибутов, приведшее к изменению размера экстента
- Запись в файл, приведшая к добавлению экстентов и/или ребалансировки деревьев
- Отбрасывание части данных файла без изменения размера

Создание сегмента отката может позволить журналировать в том числе и пользовательские данные, однако это очень сильно замедляет работу ФС, а для некоторых файлов сегмент отката необходим слишком большой.

В некоторых современных системах эту проблему решили при помощи *checkpoint* или *snapshot* - виртуальное устройство только для чтения, содержащее состояние файловой системы, включая и пользовательские данные, на определённый момент времени.

Устойчивость к сбоям диска

К сбоям диска относят различные физические повреждения дисков (дорожек на компакт-дисках и дискетах, соприкосновение головки с дорожкой в жёстком диске)

В FAT плохие блоки отмечаются особым символом (`0xFFB` или `0xFFFF`), если плохой блок появляется в таблице или в корневом каталоге, диск считается непригодным для использования

В более сложных ФС резервируется пул блоков для горячей замены (hotfixing) - ФС хранит список плохих блоков и ставит им в соответствие блоки из этого пула. В некоторых контроллерах дисков горячая замена уже реализована и производится втайне от ЦП. Таблица замены может быть как статической, так и динамической, причём ввиду специфики использования, статическая таблица вполне приемлема, так как ускорит работу ФС, а при слишком большом количестве плохих блоков есть большие основания избавиться от диска вовсе.

ФС с копированием при записи (NetApp WAFL)

Главный принцип ФС с копированием при записи - не затирание прошлых данных новыми, а запись новых данных в новом месте, причём это распространяется на все структуры, кроме самой корневой (суперблока). Внесение изменений в блок файла повлечёт за собой такую цепочку: копирование нового блока файла → копирование иноды файла с новыми данными в блоках → копирование изменённой таблицы инодов → ... → копирование старой версии и изменение суперблока.

Такой подход сход с концепцией Copy-on-Write и называется *Write Anywhere*. Он позволяет куда более естественно и удобно пользоваться концепцией журналов и транзакций, а также снапшотов.

WAFL - файловая система с концепцией копирования при записи, разработанная компанией NetApp для особых устройств - аппаратных файловых серверов.

В лог транзакций в ФС WAFL записываются запросы к серверам, что даёт по некоторым операциям значительный выигрыш в объёме журнала.

WAFL использует многоуровневую древовидную структуру, в которой все большие объекты метаданных имеют свои иноды, могут динамически расширяться и занимать произвольное место на диске. Суперблок называется корневым инодом. Структура инодов похожа на UFS с той лишь разницей, что уровень косвенности чётко привязан к размеру файлов и фиксирован. В иноде может быть 16 указателей непосредственно на блоки файла.

За счёт такой древовидной структуры принцип write-anywhere с проходом от точки изменения до корня легко реализуется.

Каждые несколько секунд WAFL создаёт снимки ФС, называемые точками согласования. Все транзакции, исполненные на момент создания снимка, помечаются завершёнными. Таким образом, при восстановлении ФС достаточно взять последнюю точку согласования и заново отработать все неисполненные транзакции.

Кроме того, в WAFL настраивается расписание снимков с более долгим сроком жизни, которые также могут быть использованы для восстановления

Вопрос 2: Диспетчер задач в транспьютере.

Иртегов с 449

КРИТ ПРО КОНТЕКСТ ПРОЦЕССА

Дело в том, что транспьютер не имеет диспетчера памяти и у него вообще очень мало регистров. В худшем случае при переключении процессов (в транспьютере, как и в старых ОС, нити называются процессами) должно сохраняться 7 32-разрядных регистров. В лучшем случае сохраняются только 2 регистра — счетчик команд и статусный регистр. Кроме того, перенастраивается регистр wptr, который выполняет по совместительству функции указателя стека, базового регистра сегмента статических данных процесса и указателя на дескриптор процесса.

Транспьютер имеет три арифметических регистра, образующих регистровый стек. При этом обычное переключение процессов может происходить только, когда этот стек пуст. Такая ситуация возникает довольно часто; например, этот стек обязан быть пустым при вызовах процедур и даже при условных и безусловных переходах, поэтому циклическая программа не может не иметь точек, которых она может быть прервана. Упомянутые в предыдущем разделе команды обращения к линкам также исполняются при пустом регистровом стеке.

Поэтому оказывается достаточно перезагрузить три управляющих регистра, и мы передадим управление следующему активному процессу.

Операция переключения процессов, а также установка процессов в очередь при их активизации полностью реализованы на микропрограммном уровне.

Деактивизация процесса происходит только по его инициативе, когда он начинает ожидать сигнала от таймера или готовности линка. При этом процесс исполняет специальную команду, которая устанавливает его в очередь ожидающих соответствующего события, и загружает **контекст очередного активного процесса**. Когда приходит сигнал таймера или данные по линку, то также вызывается микропрограмма, которая устанавливает активизированный процесс в конец очереди активных.

У транспьютера также существует микропрограммно реализованный **режим разделения времени**, когда по сигналам внутреннего таймера активные процессы циклически переставляются внутри очереди. Такие переключения, как уже говорилось, могут происходить, только когда регистровый стек текущего процесса пуст, но подобные ситуации возникают довольно часто.

Кроме обычных процессов в системе существуют так называемые **высокоприоритетные процессы**. Если такой процесс получает управление в результате внешнего события, то текущий низкоприоритетный процесс будет прерван независимо от того, пуст его регистровый стек или нет. Для того чтобы при этом не разрушить прерванный процесс, его стек и весь остальной контекст записываются в быструю память, расположенную на кристалле процессора. Это и есть тот самый худший случай, о котором говорилось ранее. Весь цикл переключения занимает 640 нс по сравнению с десятками и порой сотнями микросекунд у традиционных процессоров

Благодаря такой организации транспьютер не имеет равных себе по времени реакции на внешнее событие. На первый взгляд, микропрограммная реализация такой довольно сложной конструкции, как планировщик, снижает гибкость системы. В действительности, в современных системах планировщики имеют довольно стандартную структуру, и реализация, выполненная в транспьютере, очень близка к этому стандарту, известному как **микроядро** (microkernel)

Иртегов с 468

Микроядро транспьютера

Другим примером классического микроядра является транспьютер. Микропрограммно реализованное микроядро транспьютера содержит планировщик с двумя уровнями приоритета и средства для передачи данных по линкам.

Иртегов с 465

С понятием **ядра** — комплекса программ, исполняющихся в привилегированном (системном) режиме процессора, — мы уже сталкивались в разд. 4.5 и главе 5.

Микроядерные системы реализуют вытесняющую многозадачность не только между пользовательскими процессами, но и между нитями ядра.

Классическая реализация микроядра, QNX, состоит из вытесняющего планировщика и примитивов гармонического межпоточного взаимодействия, средств для обмена сообщениями send и receive.

Дальше странничная тирада от Иртегова по поводу додиков которые неправильно используют термин микроядро

Дальше Иртегов

Действительно, как говорилось ранее, в чистом микроядре взаимодействия происходят асинхронно, а в чистом монолитном ядре — синхронно. Если некоторые из взаимодействий происходят **асинхронно** (что неизбежно,

например, в многопроцессорной машине), то мы можем сказать, что система частично микроядерная. Если же некоторые из взаимодействий обязательно синхронны, мы, наверное, вынуждены будем признать, что наша система частично монолитная, как бы странно это ни звучало.

Танненбаум:

Следует почитать с 78 где начинается монолитное ядро

с 91

Замысел, положенный в основу конструкции **микроядра**, направлен на достижение высокой надежности за счет разбиения операционной системы на небольшие, вполне определенные модули. Только один из них —

микроядро — запускается в режиме ядра, а все остальные запускаются в виде относительно слабо наделенных полномочиями

обычных пользовательских процессов. В частности, если запустить каждый драйвер устройства и файловую систему как отдельные пользовательские процессы, то ошибка в одном из них может вызвать отказ соответствующего компонента, но не сможет вызвать сбой всей системы. Таким образом, ошибка в драйвере звукового устройства приведет к искажению или пропаданию звука, но не вызовет зависания компьютера.

В отличие от этого в монолитной системе, где все драйверы находятся в ядре, некорректный драйвер звукового устройства может запросто сослаться на неверный адрес памяти и привести систему к немедленной вынужденной остановке.

с 790

Преимущество системы с микроядром перед монолитной системой заключается в том, что систему с микроядром легко понять и поддерживать (благодаря ее высокой модульности). Кроме того, перемещение кода из ядра в пользовательский режим обеспечивает системе высокую надежность, так как сбой работающего в режиме пользователя процесса не способен нанести такой ущерб, какой может нанести сбой компонента в режиме ядра.

Основной недостаток такой системы состоит в несколько меньшей производительности, связанной с дополнительными переключениями из режима пользователя в режим ядра.

Билет 13

Вопрос 1: Сборка в момент загрузки. Преимущества и недостатки этого метода. Чем отличаются DLL Win32 и разделяемые библиотеки ELF

Начало про загрузчики смотри тут. Здесь будет инфа, начиная с 3.7 "Сборка программ"

Сборка программ

Процесс сборки программы и получения загружаемых модулей может быть несколько различен в разных ОС, но из-за того, что все системы используют +- одни и те же языки программирования, общий принцип сборки тоже будет похожим.

Первым делом код программ обрабатывается компилятором, который создаёт объектные модули, которые далее обрабатываются линкером (редактором связей), который и формирует загружаемый модуль.

Объектные модули в некотором смысле можно сравнить с перемещаемыми загружаемыми модулями, а процесс сборки с загрузкой нескольких программ, так как нам также необходимо разрешить ссылки в объектниках, связывающие их между собой. Для этих целей в каждом объектном модуле существует 2 таблицы: одна содержит объекты из других модулей, на которые ссылается текущий модуль, а другая - объекты текущего модуля, на которые могут ссылаться другие. Обычно с объектом ассоциировано имя, называемое **глобальным символом**.

Для каждой внешней ссылки мы должны уметь определить, является ли она абсолютной, относительной или и вовсе комбинируется из нескольких адресов. Для каждого доступного извне объекта мы должны указать, будет ли его символ абсолютным, относительным либо равен другому символу со смещением.

Таким образом, объектных модуль будет содержать:

- Таблицу перемещений своих внутренних объектов
- Таблица ссылок на объекты других модулей (список импорта)
- Таблица объектов этого модуля, доступных извне (список экспорта)
- Служебную и отладочную информацию
- Собственно код и данные модуля, разбитые на именованные секции. При сборке именованные секции всех объектников сливаются воедино

А дальше со стр. 179 на кой-то чёрт идёт чуть ли не полная структура данных объектника ELF. Читайте её сами, если жизнь не дорога

Объектные библиотеки

Библиотеки объектных модулей предназначены для поддержания порядка в программах, состоящих из большого количества объектных модулей. Состоит такая библиотека из заголовка, за которым последовательно располагаются сами объектные модули. В заголовок входит список всех объектных модулей с их смещением и список всех глобальных символов библиотеки с указанием того, к какому модулю относится каждый символ.

Обычные объектные модули линкер собирает при указании в командной строке даже если в остальном коде нет ни одной ссылки на глобальные символы этого модуля, а вот из библиотеки линкер загрузит только те модули, на глобальные символы которых есть ссылки., таким образом из библиотеки будут взяты только нужные модули. В системах семейства Unix объектные библиотеки называются архивными библиотеками

Сборка в момент загрузки

Отсюда идёт материал, относящий уже непосредственно к вопросу 13.1

В объектных модулях содержится достаточно информации для того, чтобы производить их сборку не заранее, а уже в процессе загрузки. Применяться такой вариант может в тех случаях, когда несколько программ используют одни и те же библиотеки. Мы можем настроить все программы на работу с одной копией такой библиотеки.

Такой вариант сборки будет занимать время при запуске программы, зато сильно упростит и ускорит процесс разработки и внесения изменений в код. В добавок, мы получаем экономию памяти.

Чаще всего для сборки в момент загрузки используются не сами объектные модули, а *разделяемые библиотеки*, которые обладают двумя отличиями от классически архивных библиотек:

- Из такой библиотеки нельзя извлечь отдельные модули, все ссылки в ней разрешены
- В такой библиотеке нет общего списка глобальных символов. При загрузке разделяемой библиотеке необходимо указывать, какие глобальные символы из неё нас интересуют

Далее со стр. 190 приводятся примеры разделяемых библиотек на архитектурах N9000 и AS/400. Мне кажется, куда целесообразнее далее будет сосредоточиться на Win32 и ELF, чем на них

Динамические библиотеки

В Windows и OS/2 при сборке используется комбинация из предварительной сборки программы и сборки в момент загрузки. Исполняемый модуль собирается из объектных модулей, но при этом содержит неразрешённые ссылки на динамически связываемые библиотеки (DLL)

Кроме общих для разделяемых библиотек особенностей, у DLL есть ещё одна: исполняемый модуль или библиотека могут загружать другие библиотеки по своему усмотрению уже после своей собственной загрузки (то есть в момент исполнения). При этом загрузчик также предоставляет возможность просмотреть список глобальных символов библиотеки и получить на него указатель, обратившись по имени.

В DEF-файле должен содержаться список символов, экспортруемых библиотекой, кроме того, в нём могут быть указаны функции инициализации и терминации библиотеки, а также необходимость использовать отдельный

сегмент данных библиотеки вместо общего. При этом современные компиляторы позволяют обойтись и без DEF-файла.

DLL очень удобны для создания отдельно загружаемых программных модулей, однако наиболее естественно реализуются в системах, использующих единое адресное пространство, при котором есть риск одним процессом данных другого. Решение же этой проблемы в виде виртуальной памяти сильно усложняет разделение кода. Второй недостаток - конфликт версий...

Но о нём и других особенностях DLL и ELF можно прочитать по ссылке из заголовка вопроса 13.1

Вопрос 2: Динамическое выделение памяти в ОС семейства Unix и стандарте POSIX

Здесь смотри секцию про динамическое выделение памяти выше

[Дополнительно имеет смысл проверить, в чём в данном случае особенности стандарта POSIX](#)

Билет 14

Вопрос 1: Драйвер устройства. Функции драйвера в ОС семейства Unix.

По главе 10 "Драйверы внешних устройств" (стр.... хз какая по книге - нумерация сломалась. В моей ПДФке - 565)

Драйвер - специализированный программный модуль, управляющий внешним устройством. Обеспечивают единый интерфейс доступа к устройствам, не зависящий от особенностей аппаратуры.

Драйвер может не относиться ни к какому физическому устройству, а быть связанным с псевдоустройством - объектом ОС, который ведёт себя аналогично устройствам ввода-вывода.

Примерами таких устройств могут быть трубы, /dev/null, а также объекты в виртуальной ФС /proc, дающие возможность работать со многими параметрами системы: адресным пространством процессов, данными и статистикой ядра и т.п.

К другим важным функциям драйверов относятся безопасность (доступ к аппаратуре прикладные процессы могут получать только через драйвер) и взаимоисключение доступа (то есть его синхронизация) в системах с вытесняющей многозадачностью

Крайне желательно как минимум обеспечение обратной совместимости ОС со старыми драйверами, потому что иначе новые версии ОС просто не смогут работать с определёнными типами внешних устройств.

Далее описываются примеры обратной совместимости в OS/2, а затем концепция ОС-универсальных драйверов, которая на практике сложна реализуема

Функции драйверов

Для начала работы драйвера используются следующие функции:

- `_init` - вызывается при загрузке модуля. Резервирует необходимые ресурсы и инициализировать глобальные переменные
- `probe` - проверить наличие устройства в системе
- `attach` - создание копии драйвера, управляющей конкретным устройством. Напоминает создание объектов классов в ООП. При вызове читается конфигурационный файл устройства, инициализируется блок переменных состояния, регистрируется обработчик прерывания и, наконец, инициализируется и регистрируется само устройство. Регистрация как доступного для пользовательских команд происходит за счёт создания мажорной записи (довольно часто таких записей создаётся несколько и действия при обращении к каждой из них будут немного отличаться, но все они будут относиться к одному драйверу)
 - В современных системах поддерживается отложенная инициализация - `attach` вызывается только при первом обращении пользовательской программы к драйверу

- `detach` - деинициализировать экземпляр драйвера, если это возможно (то есть если все выполняемые драйвером операции над устройством могут быть нормально завершены). Освобождаются занятые ресурсы системы, уничтожается минорная запись и, возможно, проводятся некоторые действия над устройством
- `_fini` - освобождение всех ресурсов модуля, не привязанных к конкретному устройству. Вызывается перед выгрузкой модуля

В ОС семейства Unix драйвера делятся на 2 типа: блочные и символьные.

Для символьных устройств определяются почти все те же операции, что и для файлов, хотя некоторые могут отсутствовать (`read`, для предназначенных только для вывода, `write` для только ввода, `lseek`, если перемещение по блокам данных устройства (в случае символьных устройств обеспечивалось для очень малого множества устройств, например, для лентопротяжных), `mmap` также использовался достаточно ограниченно (зачастую реализовывалась для тех же устройств, для которых была реализована `lseek`)

Устройства-генераторы событий - класс устройств, при работе с которыми важно не только или даже не столько поступающая структура данных, сколько время её поступления. При наступлении события программа должна будет отреагировать на него определённым образом. Может использоваться, например, для синхронизации видео и звука в играх. Реализуется за счёт блокирования всистемного вызова на чтения либо `select/poll`, либо ещё более сложных механизмов, называемых коллбэками - к событию привязывается указатель на функцию, которая будет вызвана при наступлении этого события (и *снова событийно-ориентированная архитектура*).

Для хранения очередей сообщений от устройств и к ним, а также других более сложных механизмов в Unix SVR3 появились потоковые драйвера (STREAMS), которые с точки зрения приложений воспринимались как символьные устройства, но под капотом были реализованы в разы сложнее и комплекснее. ([подробнее читай тут](#))

Далее упоминается классификация драйверов в OS/2 и Windows, но к вопросу относится слабо и уже слишком обширная, так что см. стр. 576 ПДФ.

В главе 10.2 рассказывается о том, что многие драйверы вообще не предназначены для взаимодействия с пользовательскими программами, а используются другими драйверами для удобной комбинации разных интерфейсов. Таким образом, мы получаем многоуровневые драйверы (отлично механизм работы разбирается на примере STREAMS по ссылке выше)

В 10.3 говорится о безопасности драйверов. Основная проблема - классические драйверы располагаются в ядре, так как должны работать с внешними устройствами, а потому имеют максимальный уровень доступа и могут нанести огромный вред системе. Решением стало разделение драйверов на две части: минимально возможная часть, отвечающая за общение с устройством, остаётся в системном режиме, а вся остальная логика драйвера будет исполняться в пользовательском режиме. Однако, такой метод сложно реализуем для некоторых типов устройств (в первую очередь блочных) и ощутимо понижает скорость работы драйвера

Архитектура драйвера

Основной процесс работы драйвера состоит из следующих этапов:

- Анализ запроса от процесса
- Передача команды устройству (и возможно каких-то дополнительных подготовительных команд)
- Ожидание прерывания от устройства по завершении обработки переданной команды (и для каждой дополнительной команды логика будет такая же)
- Анализ результата и формирование ответа для процесса и/или устройства

Таким образом, драйвер чаще всего делится на 2 нити: основную и обрабатывающую прерывание. Основная нить может быть самостоятельной либо той, что отправила запрос драйверу. Крайне желательно, чтобы нить прерывания выполняла возможный минимум команд. Последнее ограничение решается за счёт *fork*-процессов - высокоприоритетных нитей, создаваемых обработчиками прерываний для реализации последующей логики. С таким механизмом мы можем реализовать функцию драйвера как конечный автомат, что очень удобно (если у вас *непреодолимое желание*, то на стр. 599 ПДФ можно прочитать о подглаве о конечных автоматах. У меня такого желания нет).

Итог: драйвер архитектурно представим в виде конечного автомата и содержит набор нитей, среди которых есть основная нить, нить обработчика прерываний и, возможно, несколько fork-процессов. Все нити гарантируют взаимоисключение.

Запрос к драйверу

Обработка запроса обычно делится на 3 фазы: предобработка, исполнение и постобработка. Предобработка отвечает за проверку прав доступа, преобразование данных, подготовку драйвера и устройства и, возможно, блокирование устройства. Постобработка может включать обработку ошибок и преобразование данных (ну и какие-то доп. операции над драйвером и устройством)

Синхронный ввод-вывод

Синхронный ввод-вывод заключается в простом косвенном вызове (через системные вызовы) функций драйвера из пользовательского процесса. Основная нить драйвера будет исполняться в режиме ядра, но в контексте процесса, за счёт чего может, например, копировать данные из его буфера в системный, к которому будет иметь доступ уже исполняющийся в собственном контексте обработчик прерываний.

Проблема такого подхода очевидна - возникновение критической секции. Обычно проблема решается установкой семафора с очередью запросов.

Асинхронный ввод-вывод

Отличительной особенностью этого подхода будет то, что формирование очереди запросов на функцию предобработки запроса. Реализуется такой метод для блочных и потоковых устройств в Unix, а в некоторых системах является и вовсе единственным возможным.

В драйверах вместо стандартных `read`, `write`, `ioctl` и т.п. используется общая точка входа `strategy`, которая принимает структуру запроса и помещает её в очередь запросов к устройству. Отдельный вопрос, как поступать с этой структурой, ведь `strategy` часто вызывается не из основной нити драйвера, а из обработчика прерываний либо форка, которые не имеют доступа к пользовательской памяти. Одной из решений, копирование в ядро при предобработке и обратно при постобработке. Другое - передавать в структуре указатель на пользовательское пространство. (Оба варианта звучат так себе, честно говоря, и у каждого есть плюсы и минусы)

Скипаю 10.6 "Сервисы ядра, доступные драйверам" (хотя инфа там есть интересная) и продолжаю про асинхронный ввод-вывод по 10.7

Асинхронное чтение тоже имеет смысл. Например, мы можем отправить запрос на чтение большого объёма данных, заниматься чем-то полезным и лишь затем проверить, записались ли данные в буфер и, если нет, заснуть.

При отложенной записи нам необходимо знать, когда она завершится, чтобы иметь возможность вновь работать с буфером, используемым при записи. Впрочем, если система копирует данные из пользовательского буфера и далее сама организует буфера для отложенной записи, эта проблема снимается. Системы с отложенной записью не всегда могут мгновенно отреагировать на ошибку в устройстве. При ошибке они устанавливают флаг ошибки в блоке данных устройства, который проверяется на этапе предобработки, из-за этого некоторые ошибочно исполнившиеся запросы могут восприниматься пользовательским процессом, как исполнившиеся нормально.

Вопрос про технические детали реализации асинхронного ввода-вывода в стандарте POSIX

В современных системах приоритетным считается синхронный ввод-вывод. Во многом это вызвано влиянием Unix с достаточно удобным, приятным и понятным интерфейсом работы с драйверами. Сейчас такая модель включена в стандарт POSIX

Впрочем, в современных Юниксах есть и возможность использовать асинхронный ввод-вывод, включив соответствующий режим в системном вызове `fcntl`

Далее идут главы 10.8 "Дисковый кэш" и 10.9 "Спулинг", которые интересны и небольшие, поэтому тут их приводить не буду, но если выпадет этот вопрос, будет иметь смысл чекнуть

Вопрос 2: Файловая система FAT.

Билет 15

Вопрос 1: Динамическое выделение памяти. Методы борьбы с фрагментацией. Основные алгоритмы выделения памяти

И снова тут будут с подзаголовками собраны ответы на несколько вопросов по главе "Управление оперативной памятью"

Выделение памяти для систем с открытой памятью

Учитывая механизм виртуальной памяти для процессов, многие решения и стратегии для открытой памяти, будут применимы и к виртуальной памяти процесса

Динамическая память программы размещается после конца её кода, статических данных и стэка и не должна конфликтовать с идущим далее кодом системы (либо не входить в red zone, если речь о системах с виртуальной памятью). За контроль выделения динамической памяти отвечают выделяющие её функции, например `malloc`. Первым делом она проверяет, нет ли подходящего участка памяти в пуле (куче) уже выделенных и, если такого нет, запрашивает у системы расширение динамической памяти. Конец динамической памяти хранится в переменной `brk_addr`, при помощи вызова `sbrk` он может быть сдвинут,

Отличительной особенностью динамического выделения памяти является необходимость дать возможность отказываться от выделенных участков памяти, чтобы они могли быть в дальнейшем повторно использованы. Из-за этого мы не можем просто хранить конец занятой области памяти, а должны знать обо всех участках динамически выделенной памяти. При этом мы получаем проблему внешней фрагментации. Самое простое решение - выделять блоки памяти, кратные блокам фиксированной длины, причём чем блок больше, тем меньше внешняя фрагментация, однако в этом случае мы сталкиваемся уже с проблемой внешней фрагментации. Согласно подсчётом в среднем будет теряться около половины каждого запрашиваемого блока. Алгоритмов выделения, которые бы эффективно боролись с обоими типами фрагментации, не существует. (хотя само собой, дефрагментация решит проблему, но операция это долгая и затратная)

Зачастую все свободные блоки памяти объединяются в двусвязный список (выбирается именно двусвязный для удобства извлечения из него блоков). Для поиска блоков в списке могут использоваться стратегии `first fit`, `best fit` и `worst fit`.

`Worst fit` оправдана только в случае работы с отсортированным по убыванию списком, но вставка освободившихся блоков в такую структуру будет $O(n)$. Другой вариант - использовать `maxHeap` (сортирующую кучу), однако вставка туда свободных блоков также занимает время больше константы - $O(\log n)$. В связи с этим `worst fit` при выделении памяти используется редко.

`Best fit` достаточно неплоха когда мы работаем с блоками фиксированных размеров. В ином случае эта стратегия быстро приведёт к внешней фрагментации.

Поиск блока методом `first fit` в среднем будет быстрее, чем `best fit`, однако если каждый раз начинать поиск с начала списка, мы рискуем со временем накопить в начале много маленьких блоков и увеличить время работы выбора блока. Решением стало закольцовывание списка с продолжением поиска нового блока с того места, на котором поиск завершился в прошлый раз.

Алгоритм парных меток

Следующей важной задачей при работе с динамической памятью является объединение смежных свободных блоков в один блок. Решается она алгоритмом парных меток:

- Каждый блок в начале и в конце содержит свою длину в байтах. Если блок занят, это помечается особым значением (например, отрицательной длиной этого блока)
- В момент освобождения блока мы легко можем проверить, свободен ли блок перед ним и после и провести объединение с ними, сложив значения длин блоков. При этом, если мы присоединяем блок к блоку перед ним, нам даже нет необходимости изменять список блоков.

На данный момент используются более сложные версии алгоритма парных меток: блоки памяти могут содержать больше информации и организовываться в иные структуры данных (сортированный по адресам список из свободных и занятых блоков), но суть остаётся та же - обеспечивать доступ к смежным блокам за константное время

Подводя итог, одной из самых выигрышных комбинаций будет создание закольцованного списка блоков с использование стратегии поиска first fit и алгоритма парных меток

КРИТ На стр. 230 есть описание `malloc` / `free` в GNU LibC, тут приводить его не буду. А краткая версия мне тут нравится из критов прошлых курсов

ПРИМЕЧАНИЕ:

В билете 13 от нас хотят динамическое выделение памяти в ОС семейства Unix и стандарте POSIX... С одной стороны, можно наверное рассказать про вот эти вот стратегии. С другой, будто бы тут от нас хотят чего-то более конкретного. Возможно, подойдёт как раз эта инфа про malloc/free в GNU, но... GNU is not Unix, так сказать. В общем, тут я доподлинно не уверен, откуда стоит брать инфу

Алгоритм близнецов

First fit со списками хоть и работает неплохо, но не даёт нам точного времени на поиск блока памяти, что недопустимо для систем реального времени (**КРИТ**). Решением будет создание списков с блоками фиксированных размеров, однако в таком случае не самой простой задачей будет отрезание незадействованной части блока и помещение её в другой список блоков (соответственно, при этом стратегии мы получаем внутреннюю фрагментацию, но куда ж деваться?)

В алгоритме близнецов блоки выделяются фиксированного по степеням двойки от 512 байт размера. Вся представляется в виде бинарного дерева, в котором размер блоков, соответствующих потомкам, равен половинам размера родителя. Благодаря такой организации мы можем за логарифмическое время выделять блоки, кратные степеням двойки минимального подходящего размера (best fit) и рекурсивно объединять освободившихся потомков также за логарифмическое время

ПРИМЕЧАНИЕ:

В билете 2 спрашиваются ограничения этих алгоритмов. В процессе переписывания я их тут особо явно не указывал, поэтому либо надо пробежаться ещё раз по тексту в книге, либо подумать

Слабовые аллокаторы

Slab - плита, лист, пластина. Используется, если нам нужно много блоков одинакового размера (особенно, если они не кратны степеням двойки). Весь пул делится на слабы, в каждом из которых содержатся фрагменты фиксированного размера. Сначала пытаемся выделить фрагмент из частично занятого слаба, при неудаче - из свободного. В ином случае можем запросить новый slab у системы.

Хз, что про них можно ещё добавить. Вставлю записи из конспекта и отмечу, что на странице 250 есть лютая душнота про использование slabов в ядре Linux

В ядре создаётся объект кэша, в котором указывается, какого размера блоки мы хотим выделять. Затем берётся большой кусок памяти и режется на фрагменты указанного нами размера. Кэши связываются списком. В кэше хранятся указатели на занятые слабы, частично свободные и полностью свободные.

Вопрос 2: Флаги событий RSX и VMS. Что такое AST?

Вопрос выглядит ну слишком халявным, поэтому наверняка к нему будет много допов про семафоры и прочие штуки (всё есть по той же ссылке)

Билет 16

Вопрос 1: Мертвая и живая блокировки. Способы их предотвращения. Преимущества и недостатки каждого из методов

Вопрос 2: Разделяемые библиотеки формата ELF

Здесь буду писать про разделяемые библиотеки на основании главы 5.4. Есть основания полагать, что тут может быть также важна инфа из главы 3

Разделяемые библиотеки стали одним из значимых преимуществ виртуальной страничной и сегментной адресации (при базовой их использование невозможно вовсе), однако и тут были определённые проблемы. В первую очередь, библиотека должны так или иначе собираться. Сборка будет происходить в момент загрузки, а для разрешения адресов в библиотеке, её будет необходимо переместить, из-за чего библиотека уже не будет разделяемой (наверное звучит несколько криво, но это то, как я понял абзац в начале главы)

Первым решением в старых Юниксах было выделение для библиотек абсолютных адресов в заранее оговоренных с разработчиками ОС местах, что оказалось крайне неудобно.

Далее последовала идея отображать часть единого адресного пространства на область в памяти, куда будут загружаться все библиотеки

Проблемы обоих подходов:

- Неэффективное использование адресного пространства - из всего списка библиотек конкретному процессу зачастую нужны лишь немногие
- Конфликт имён и, как следствие, трудности с использованием в разных процессах одной и той же библиотеки разных версий

О разделяемых библиотеках Win32/Win64 (PE)

В Windows 95 32-битные DLL отображались в третий гигабайт адресного пространства и обладали обоими недостатками, описанными выше.

В Windows 2000 и OS/2 было предложено решение для использования разных версий одной библиотеки - использование полного путевого имени библиотеки при её загрузке. Однако, проблему неэффективного использования памяти это не решало

Все системы семейства win32 используют модули загрузки формата PE (Portable executable), который является расширением формата COFF - формат PE допускает таблицы перемещений в загружаемых модулях, а не только в объектных.

Каждая библиотека обязана иметь свой загрузочный адрес, который базово располагается в начале второго гигабайта, но при наличии нескольких библиотек, может быть изменён без пересборки всей библиотеки. В конце второго гигабайта содержатся системные библиотеки.

Если библиотека загружается по своему адресу, то она может быть отображена в виртуальную память процесса без дополнительной настройки (из этого пункта следует, что теперь мы не отображаем во все процессы все библиотеки). Изменения над библиотекой запрещены. При необходимости она вновь подкачивается из файла.

Если библиотека не может быть загружена по своему адресу, ей подбирается другой подходящий, при этом библиотека уже не может быть разделённой - её содержимое копируется в память процесса. Страницы памяти с библиотекой считаются изменёнными и могут сохраняться в swap-файл, а также подкачиваться из него.

При таком подходе программы, работающие с конфликтующими по адресам загрузки библиотеками, будут работать нормально, но потребление памяти сильно возрастёт

Наконец, разделяемые библиотеки формата ELF

Для рассказывания ответа на этот вопрос критически важно знать, что такое позиционно-независимый код (КРИТ. Не, ну вы поняли? Поняли?!... Мне надо поспать...) (это снова третья глава... скоро-скоро будет и она)

Исполняемые модули формата ELF бывают статическими и динамическими. Статические модули полностью самодостаточны, так как не используют разделяемых объектов. Динамические модули содержат разделяемые

объекты и неразрешённые ссылки.

И те, и другие являются абсолютными (снова глава 3). Первым делом при создании образа процесса система статически отображает модуль в адресное пространство. Для статических модулей приключения на этом заканчиваются.

Для динамических модулей далее в контексте процесса ([КРИТ](#)) запускается интерпретатор или редактор связей времени исполнения, который осуществляет подгрузку разделяемых объектов и связывание их с модулем и между собой. Разделяемые объекты располагаются в памяти процесса как получится и могут даже менять своё расположение от запуска к запуску. (далше идёт фраза о том, что не позиционно-независимый код либо модуля либо не исполнится, либо будет работать медленно из-за большого количества перемещений текстового сегмента, но, как мне кажется, говорить это опасно)

Разделяемый объект делится на 2 части:

- Создаваемую в каждом образе процесса:
 - Сегмента данных, в которых включены статически инициализированные указатели и ссылки на процедуры других модулей
 - спользуемые в коде модуля ссылки собираются в 2 таблицы: GOT (global offset table) и PLT (Procedure linkage table)
- Разделяемую между процессами часть, в которую включён непосредственно код объекта

По мере разрешения ссылок объекта, интерпретатор будет заполнять его PLT, причём из разделяемого модуля будут загружены только те функции, которые были реально вызваны из модуля, что даёт некоторый прирост в производительности.

Сегмент данных разделяемого объекта напоминает приватных сегмент DLL из Win32 (в данном случае DLL и разделяемая библиотека могут считаться синонимами), при этом аналога глобальному сегменту разделяемых данных у ELF нет. Разделяемый объект может создать собственный сегмент разделяемой памяти, но в нём будет невозможно создать статически инициализированные данные и гарантировать его расположения на одних и тех же адресах в разных процессах.

Таким образом, ELF обеспечивает менее глубокое разделение данных, чем DLL в win32, однако делается куда проще и создаёт меньше проблем.

Разделяемые объекты идентифицируются по короткому либо полному имени. Короткое имя будетискаться в переменной среды ([может спросить, что это](#)) [LD_LIBRARY_PATH](#), в заголовке модуля [RPATH](#) и в каталогах по умолчанию из файла [var/ld/ld.config](#). Такой набор источников даёт куда больше свободы и удобства в сравнении с win32

Если охота умереть, можно ещё глянуть структуру объектника ELF на стр. 179

Билет 17

1. Разделяемая память. Преимущества и недостатки по сравнению с другими методами межпроцессного взаимодействия.

Разделяемая память - это какая-то область памяти одновременно используемая несколькими процессами (четкого определения я не нашел)

Также называется общей памятью - [Современные ОС - Танненбаум](#) - Глава 2.3 стр.147

Разделяемая память ([англ.](#) Shared memory) является самым быстрым средством обмена данными между [процессами](#)[1].

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[2].

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[3]. Таким образом, два разных процесса могут иметь разные адреса одной и той же ячейки подключенной разделяемой памяти.

Из википедии: https://ru.wikipedia.org/wiki/Разделяемая_память#cite_note-0-3

Не нашел ничего лучше, потому что у Иртегова и Танненбаума описаны алгоритмы работы именно над разделяемой памятью. Принципиально других методов взаимодействия я не нашел. Возможно, имеется в виду, что чаще всего при работе с общей памятью используются семафоры, а как альтернатива могут быть: очереди сообщений, почтовые ящики и даже сокеты.

Все эти способы описаны - Современные ОС - Танненбаум - Глава 2.3 (вся) стр.147

и в - Введение в операционные системы - Иртегов - Глава 7 стр.380-385 (и другой материал в той же главе)

2. Механизм setuid в ОС семейства Unix.

Общее про setuid

В некоторых случаях у разработчиков приложений или ОС возникает потребность или, во всяком случае, желание разрешить контролируемую смену идентичности для исполнения одной операции или группы тесно связанных операций над данными. Требование контролируемости означает, что нельзя разрешать пользователю свободное выполнение таких операций: например, в системе документооборота пользователю может быть разрешено поставить под документом подпись, что он с ним ознакомился, но при этом не дано права изменять содержание документа или удалять ранее поставленную подпись.

Если предоставляемый системой список прав на документ либо позволяет редактирование документа целиком, либо опять-таки целиком запрещает

его редактирование, то мы сталкиваемся с неразрешимой проблемой.

Одним из решений было бы хранение пароля для каждого из пользователей

в отдельном файле, но это во многих отношениях неудобно. Другое решение может состоять в использовании модели "клиент-сервер" с процессом-сервером, исполняющимся с привилегиями администратора, который является единственным средством доступа к паролям. Например, в Win32 весь доступ к пользовательской базе данных осуществляется через системные вызовы, т. е. функции процесса-сервера исполняет само ядро системы.

В системах семейства Unix для этой цели был предложен оригинальный механизм, известный как setuid (setting of user id — установка [эффективного] идентификатора пользователя). По легенде, идея этого механизма возникла именно при решении задачи о том, как же пользователи смогут менять свои пароли, если их хэши будут храниться в одном файле.

В Unix каждая задача имеет два пользовательских идентификатора: реальный и эффективный. Реальный идентификатор обычно совпадает с идентификатором пользователя, запустившего задание. Для проверки прав доступа к файлам

и другим объектам, однако, используется эффективный идентификатор. При запуске обычных задач реальный и эффективный идентификаторы совпадают.

Несовпадение может возникнуть при запуске программы с установленным признаком setuid . При этом эффективный идентификатор для задачи устанавливается равным идентификатору хозяина файла, содержащего его программу.

Признак setuid является атрибутом файла, хранящего загрузочный модуль программы. Только хозяин может установить этот признак, таким образом передавая другим пользователям право выполнять эту программу от своего имени. При модификации файла или при передаче его другому пользователю признак setuid автоматически сбрасывается.

Например, программа /bin/passwd, вызываемая для смены пароля, принадле-

жит пользователю root , и у нее установлен признак setuid . Любой пользователь, запустивший эту программу, получает задачу, имеющую право модификации пользовательской базы данных, — файлов /etc/passwd и /etc/shadow. Однако, прежде чем произвести модификацию, программа passwd проверяет допустимость модификации. Например, при смене пароля она требует ввести старый пароль (рис. 13.13). Сменить пароль пользователю, который его забыл, может только root .

Другим примером setuid-программы может служить программа /bin/ps (process status — состояние процессов) в старых системах. До установления стандарта на псевдофайловую систему /proc, Unix не предоставляла штатных средств для получения статистики об исполняющихся процессах.

Программа /bin/ps в таких

системах анализировала виртуальную память ядра системы, доступную как файл устройства /dev/kmem, находила в ней список процессов и выводила содержащиеся в списке данные. Естественно, только привилегированная программа может осуществлять доступ к /dev/kmem.

Механизм setuid был изобретен одним из отцов-основателей Unix, Деннисом Ритчи, и запатентован компанией AT&T в 1975 г. Через несколько месяцев после получения патента ему был дан статус public domain. Официально AT&T объяснила это тем, что плату за использование данного патента нецелесообразно делать высокой, а сбор небольшой платы с большого числа пользователей неудобен и тоже нецелесообразен.

Механизм setuid позволяет выдавать привилегии, доступные только супер пользователю, как отдельным пользователям (при этом setuid-программа должна явным образом проверять реальный идентификатор пользователя и сравнивать его с собственной базой данных), так и группам (при этом достаточно передать setuid-программу соответствующей группе и дать ей право исполнения на эту программу), таким образом, отчасти компенсируя недостаточную гибкость стандартной системы прав доступа и привилегий в системе Unix.

sudo в современных Unix-системах

В современных системах семейства Unix широкое распространение получил механизм делегирования полномочий sudo, в конечном итоге основанный на механизме setuid . Программа sudo принадлежит пользователю root и имеет установленный бит setuid , так что она исполняется с эффективными правами root .

Благодаря этому же обстоятельству, sudo может сменить идентичность и выполнить команду от имени любой другой учетной записи в системе.

Конфигурация утилиты sudo размещается в файле /etc/sudoers. В этом файле перечислены пользователи и группы пользователей, обладающие особыми правами в системе.

При этом могут использоваться как общесистемные группы, перечисленные в /etc/group, так и логические группы sudo, состав которых также описывается в /etc/sudoers.

Для каждой группы указано, какими именно правами члены этой группы обладают, в виде списка команд, которые они могут выполнять, и учетной записи, от имени которой эти команды будут запущены. Команды можно указывать вместе с параметрами; в именах команд и параметров можно использовать шаблоны.

Наивысший уровень полномочий допускает запуск командного интерпретатора и выполнение произвольных команд от имени root .

Файл /etc/sudoers обладает довольно сложным синтаксисом, который позволяет задавать сложные комбинации прав: например, пользователь может обладать разными полномочиями в разное время (так, оператору системы можно давать права только на период его дежурства) или при входе с разных терминалов (так, пользователю, зашедшему с консоли, можно дать право монтировать и размонтировать дискеты и другие сменные носители); реализация разных прав может требовать разных механизмов аутентификации.

sudo поддерживает три механизма аутентификации: NOPASSWD, PASSWD и ROOTPW. При использовании NOPASSWD sudo проверяет только свой реальный UID. В этом случае sudo можно использовать в фоновом режиме, например из запускаемых по расписанию заданий. Однако есть опасность, что злоумышленник может реализовать полномочия пользователя, сев за его оставленный без присмотра терминал. PASSWD — предпочтительный механизм, при котором sudo спрашивает пароль запустившего программу пользователя. В этом режиме sudo можно использовать только интерактивно, но гарантируется, что операция действительно запрошена именно тем пользователем, которому даны полномочия, sudo хранит историю аутентификаций и, как правило, в течении нескольких минут после успешной аутентификации не переспрашивает пароль, так что пользователь может исполнить серию команд, не вводя пароль каждый раз.

ROOTPW предполагает аутентификацию с помощью пароля root, а не собственного пароля пользователя. Честно говоря, я не понимаю, какие преимущества дает использование sudo в таком режиме по сравнению с более примитивными средствами, такими как команда su или вход в систему из-под root.

На практике этот режим используется очень редко.

Способы использования sudo разнообразны. Так, администратору БД можно дать право перегружать сервер СУБД и изменять его конфигурацию; оператору можно дать право управлять очередями печати, монтировать и размонтировать запоминающие устройства со сменными носителями, "убивать" сессии пользователей и выключать систему и т. д. В простейшем случае, sudo используется для того, чтобы дать некоторым людям полные административные права, не сообщая каждому из них пароль root — это приблизительно соответствует добавлению пользователя в группу Local Administrators в Windows NT. Даже в этом режиме sudo дает большие преимущества по сравнению с разделяемым паролем root: резко упрощается управление паролем супервизора (можно вообще запретить прямые регистрация в системе из-под root) для каждого действия, совершенного от имени root, остается информация в логах о том, кто именно из администраторов совершил это действие, так что значительно упрощается расследование инцидентов, связанных с ошибками администрирования.

Механизмы, функционально аналогичные sudo, предоставляются также многими пользовательскими графическими оболочками и— KDE, Gnome, MacOS X и др

Полезные источники

Некоторая конкретная информация есть в Главе 1 - [Системные вызовы Unix SVR 4](#)

Про setuid - [Введение в операционные системы - Иртегов](#) - Глава 13 стр.785

Для лучшего понимания - [Введение в операционные системы - Иртегов](#) - Глава 13 стр.772-785

Билет 18

Вопрос 1: Событийно-ориентированные системы. Обязательно ли такая система является многопоточной?

Из брейншторма сделали вывод, что такая система может быть и полностью однопоточной, так как менеджер сообщений спокойно может последовательно отправлять обработчикам сообщения из очереди, правда, большие сомнения вызывает производительность такой системы

Вопрос 2: Понятия инода и связи в файловых системах ОС семейства Unix.

Немного мутный вопрос, но в целом, думаю, тут имеет смысла рассказать следующее:

- Базовые понятия отсюда
- Потом рассказать про UFS и использование в ней инодов, а также про механизм ссылок (я полагаю, что именно они имеются ввиду под "связями")
- Продублирую тут вложенную ссылку из первого пункта на конспект, где можно прочесть про структуру инодов и многое другое

Билет 19

Вопрос 1: Реентерабельная программа. Техника реализации реентерабельных программ. Всегда ли это возможно? Что такое критическая секция?

Выглядит так, будто ответ лоскутно собирается из других вопросов:

- Базовые определения и некоторые примитивы взаимоисключения
- Отдельно про спинлоки как средство взаимоисключения
- Гармоническое взаимодействие

Материал получается архиагромным, поэтому из каждого пункта списка, вероятно, стоит взять лишь базовые определения (а дальше пусть щипцами остальное вытягивает, хых...)

Всегда ли возможно создание реентерабельной программы - вопрос хороший

Вопрос 2: Загружаемые модули и разделяемые библиотеки Win32/Win64 (PE).

Билет 20

Вопрос 1: Прерывания в классических процессорах (PDP-11, 8086, x86). Внешние прерывания и исключения (exceptions).

По главе 6: "Компьютер и внешние события"

Обработка внешних событий - одна из основных функций современных компьютеров. Реакций на событие является исполнение программы. Самый простой вариант реализации - условный переход по адресу обработчика события при обнаружении признака этого события. Самый простой признак - обнаружения бита в специальном регистре, который будет поднять при получении на вход высокого напряжения (используется в микроконтроллерах)

Опрос

Самый наивный вариант обнаружения наступления события - циклическая проверка признака события. Называется такой метод опросом. Хорош для простого кода и последовательности событий, следующих друг за

другом, но постоянно занимает процессор циклом, не давая использовать его для других задач. Обратная проблема - при исполнении другого кода процессор может отреагировать на событие с большой задержкой, а при установке опрашивающих проверок во многих местах кода мы будем занимать много памяти и усложним код.

Канальные процессоры и прямой доступ к памяти

Одним из решений описанной выше проблемы стало выделение отдельного процессора для опроса. Такие процессоры называются *периферийными* или *канальными*, имеют достаточно простое устройство и в современных машинах используются достаточно часто для многих периферийных устройств (видеокарта, модем, принтер и т.п.). Подключаются канальные процессоры напрямую к системной шине и могут обслуживать несколько внешних устройств.

Особенно простая версия канального процессора используется для контроля передачи данных с ПЗУ (синхронизации передачи блоков при разных скоростях системной шины и чтения с диска, чтобы не потерять слова). Отвечающие за это устройства называются *контроллерами прямого доступа к памяти (DMA)*. Для работы с виртуальной памятью у DMA обычно имеется свой диспетчер памяти. DMA может быть как отдельным у ПЗУ, так и установленным на системной плате, работающим с несколькими устройствами.

Хотя канальные процессоры и упростили реализацию механизма опроса, фактически создаваемых этой концепцией проблем они не решили: в некоторых случаях на событие требовалась реакция ЦП, машина в целом становилась дороже, ЦП теперь должен был опрашивать не признаки события, а флаги от канального процессора.

Прерывания

Альтернативой опросу стало *прерывание*, которое значительно усложняет логику обработки внешних событий, но даёт огромную гибкость. Суть метода в том, что у процессора есть входы, называемые *линиями запроса прерываний (IRQ - Interrupt ReQuest)*. При появлении сигнала по такой линии процессор дожидается завершения текущей команды и запускает обработчик прерываний:

- Сохраняем значение счётчика команд и, возможно, других регистров (в современных машинах оно сохраняется в стек)
- Передаёт управления на адрес, по которому располагается программа-обработчик прерываний.
- После завершения обработки, восстанавливаются значения регистров и счётчика команд, и исполнение программы исполняется с той же точки, где она была прервана

Благодаря такому механизму процессор даже в состоянии обработать прерывание внутри прерывания.

Адреса программ-обработчиков собраны в таблицу векторов прерываний. В микроконтроллерах в таблице отводятся вектора для каждого сигнала. В других системах бывает более сложная логика (например, запрашивающее прерывание устройство может также передавать адрес обработчика)

Классический подход к прерываниям на примере PDP-11

В PDP-11 определено 128 прерываний и исключений. Вектор прерывания состоит из двух слов: адреса обработчика и слова состояния процессора (говоря иначе, приоритет), определяемое разработчиком.

Каждый получаемый процессором сигнал прерывания имеет свой приоритет, определяемый входом сигнала (этот приоритет уже не определяется разработчиком!). Приоритет может быть значением от нуля до семи. Обработка прерывания начнётся только когда приоритет процессора будет ниже приоритета прерывания.

Начиная обработку прерывания, процессор дожидается исполнения текущей команды выставляет сигнал готовности и получает от внешнего устройства адрес вектора прерывания, через шину данных. Далее в стек сохраняется адрес команды и состояние (приоритет), затем из вектора прерываний берутся новые адрес и приоритет. После завершения обработчика адрес команды и состояние процессора до прерывания выгружаются из стека.

В x86 процессорах вектор прерываний содержит только адрес. Приоритет берётся из регистра устройства, вызвавшего прерывание.

В системах с виртуальной памятью логика обработки прерываний несколько усложняется. В PDP-11, где в MMU хранятся данные двух адресных пространств: системного и пользовательского - прерывания обрабатываются в системном пространстве. В x86 у каждого прерывания и вовсе может быть своё адресное пространство.

Обработка прерываний сильно выигрывает в сравнении с опросом, но обладает и одним значительным недостатком - время на подготовку к обработке прерывания и возврат из обработчика может оказаться на современных машинах очень значительным в сравнении с простой проверкой флага и условным переходом. Однако преимущества всё равно слишком велики. Одним из решений может быть использование канальных процессоров, но уже для обработки прерываний, а не опроса.

Исключения

Исключительные ситуации (exception) - своего рода внутренние прерывания, использующиеся для обработки особых исключительных ситуаций, возникших в ходе исполнения программы. Главной отличительной особенностью исключений является то, что оно возникает в ходе исполнения команды и приводит к её откату (отменяются все побочные эффекты, а счётчик команд не увеличивается), за счёт чего после обработки исключения вызвавшая его команда будет исполнена вновь. Особенно удобен такой механизм, например, при страничной подкачке в виртуальной памяти или при эмуляции расширенных команд.

В главе 6.5 "Многопроцессорные архитектуры" материал достаточно интересный, но ни к одному вопросу вроде как не относится

Вопрос 2: Сборщик мусора Java G1.

UPDATE: В ЛЕКЦИИ ИРТЕГОВА БЫЛО:

- Появился в Java 1.6
- Вместо «поколений» использует разбиение кучи на области («карты») одинакового размера
- Использует remembered set для отслеживания ссылок между картами
- Оценивает количество живых объектов в каждой карте
- Выбирает регионы с наименьшей стоимостью сборки (стоимость сборки оценивается через количество живых объектов и размер remembered set)
- Использует «неблокирующуюся» сборку

Преимущества

- Выполняет большую часть работы в фоновом режиме
- Оптимизирован для работы на многоядерных процессорах
- Минимизирует паузы при сборке

Недостатки

- Требует большого запаса памяти (как и все фоновые сборщики)
- Занимает процессор, что может быть существенно при малом числе процессорных ядер
- До некоторых объектов сборка может не доходить очень долго

Про этот сборщик в книге Иртегова ничего нет. Поэтому будем пользоваться официальной документацией oracle.

<https://www.oracle.com/technetwork/tutorials/tutorials-1876574.html>

Для перевода используется DeepL

СТАТЬЯ НА ХАБРЕ КОТОРАЯ МОЖЕТ БЫТЬ ПОНЯТНЕЕ

(Статья на хабре, конечно, понятнее, чем официальная документация, но я считаю, что документация намного надежнее и приоритетнее в глазах Иртегова, другими словами, статья на хабре это что-то вроде надписей на заборе)

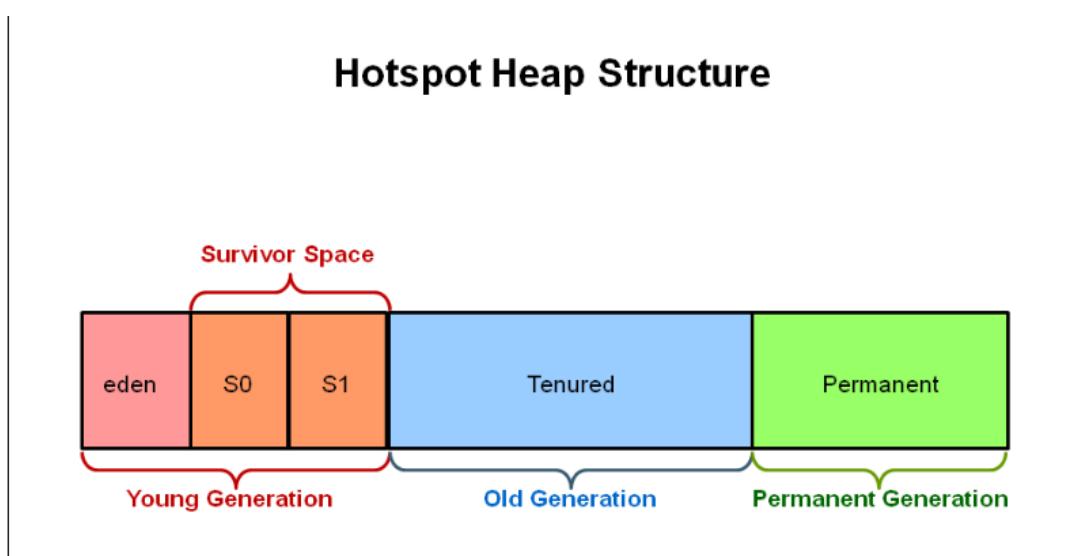
Сборщик Garbage-First (G1) – это сборщик мусора серверного типа, предназначенный для многопроцессорных машин с большой памятью. Он с высокой вероятностью достигает целей по времени остановки сборки мусора (GC), обеспечивая при этом высокую скорость.

При сравнении G1 с CMS (Concurrent mark and sweep) есть различия, которые делают G1 лучшим решением. Одно из отличий заключается в том, что G1 – это уплотняющий коллектор (compacting). G1 достаточно компактен, чтобы полностью отказаться от использования списков для распределения, и вместо этого полагается на регионы. Это значительно упрощает часть сборщика и в основном устраниет потенциальные проблемы фрагментации. Кроме того, G1 предлагает более предсказуемые паузы в сборке мусора, чем сборщик CMS, и позволяет пользователям указывать желаемые цели остановки сборки.

Обзор работы G1

Все старые сборщики мусора (последовательный, параллельный, CMS) делят кучу на три секции: **молодое поколение**, **старое поколение** и **постоянное** поколение фиксированного объема памяти.

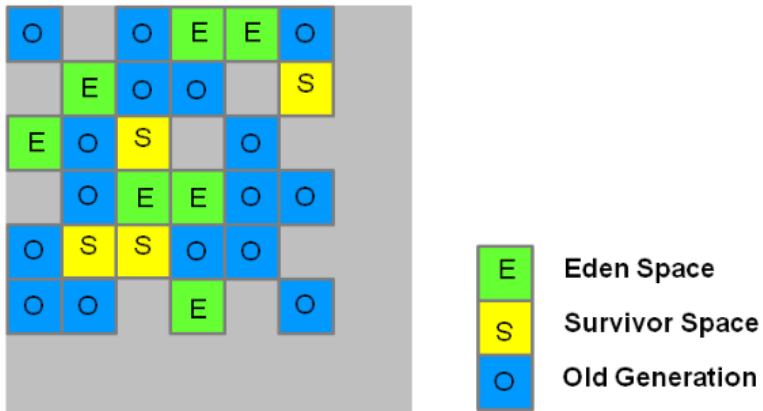
Можно читать про Hotspot сборку.



Все объекты памяти оказываются в одной из этих трех секций.

Коллектор G1 использует другой подход.

G1 Heap Allocation



Куча разбита на множество одинаковых по размеру регионов, каждый из которых представляет собой непрерывный диапазон виртуальной памяти. Определенным набором регионов присваиваются те же роли (eden - **эдем**, survivor - **пространство выживших**, old - **старые**), что и в старых коллекторах, но их размер не фиксирован. Это обеспечивает большую гибкость в использовании памяти.

При выполнении сборки мусора G1 действует аналогично сборщику CMS(mark and sweep). G1 выполняет параллельную фазу отмечания(mark) для определения актуальности объектов во всей куче. После завершения фазы отмечания G1 знает, какие регионы в основном пусты.

Он собирает мусор в этих регионах в **первую очередь**, что обычно дает большой объем свободного пространства. Именно поэтому этот метод сборки мусора называется **Garbage-First**. Как следует из названия, G1 концентрирует свою деятельность по сбору и уплотнению на тех областях кучи, которые, скорее всего, будут заполнены мусором.

G1 использует модель предсказания паузы для достижения заданного пользователем целевого времени остановки и выбирает количество областей для сбора на основе заданного целевого времени остановки.

Регионы, определенные G1 как готовые для переиспользования, собирают мусор с помощью **эвакуации**. G1 копирует объекты из одного или нескольких регионов кучи в один регион, при этом одновременно уплотняя и освобождая память. **Эвакуация** выполняется параллельно на нескольких процессорах, чтобы уменьшить время пауз и увеличить скорость. Таким образом, при каждой сборке мусора G1 непрерывно работает над уменьшением фрагментации, работая в пределах заданного пользователем времени остановки. Это выходит за рамки возможностей обоих предыдущих методов. Сборщик мусора CMS (Concurrent Mark and Sweep) не выполняет уплотнение. Сборщик мусора ParallelOld выполняет только уплотнение всей кучи, что приводит к значительным паузам.

Для понятности, добавим step by step

СТЕП БАЙ СТЕП БОЛЬШОЙ МОЖНО НАВЕРНОЕ ВЕСЬ НЕ ЧИТАТЬ НО ПРОЧИТАТЬ ВЫВОДЫ О РАЗЛИЧИЯХ МАЛОЙ И БОЛЬШОЙ СБОРОК ОБЯЗАТЕЛЬНО!

Коллектор G1 использует другой подход к распределению кучи. На следующих рисунках пошагово рассматривается система G1.

Структура кучи G1

Куча - это единая область памяти, разбитая на множество областей фиксированного размера. Размер региона выбирается JVM при запуске

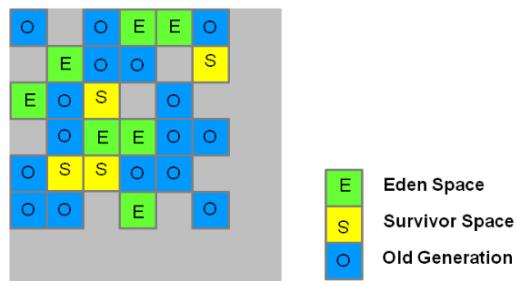
G1 Heap Structure



Распределение кучи G1

В реальности эти регионы отображаются в логические представления пространств эдема, выживших и старого поколения.

G1 Heap Allocation



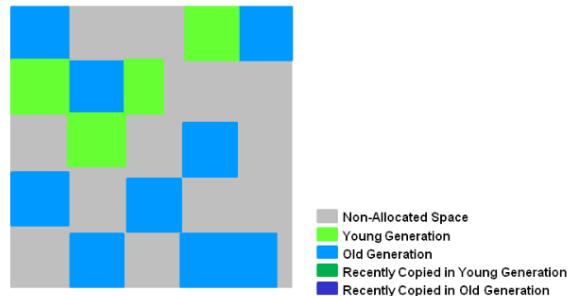
Цвета на рисунке показывают, какой регион связан с той или иной ролью. Живые объекты эвакуируются (т. е. копируются или перемещаются) из одного региона в другой. Регионы разработаны таким образом, чтобы их можно было собирать параллельно с остановкой или без остановки всех остальных потоков приложения.

Как показано на рисунке, регионы могут быть разделены на регионы эдема, выжившие и старого поколения. Кроме того, существует четвертый тип объектов, известный как регионы Humongous (огромные). Эти регионы предназначены для хранения объектов, размер которых составляет 50 % от размера стандартного региона или больше. Они хранятся в виде набора смежных регионов. Наконец, последний тип регионов – это неиспользуемые области кучи.

Молодое поколение в G1

Куча разбита примерно на 2000 областей. Минимальный размер – 1 Мб, максимальный – 32 Мб. В синих регионах находятся объекты старого поколения, а в зеленых – молодого.

Young Generation in G1

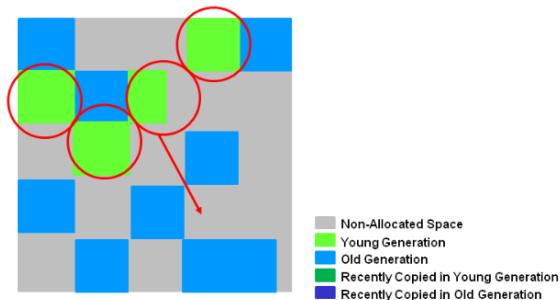


Обратите внимание, что регионы не обязаны быть смежными, как в старых сборщиках мусора.

Малая сборка в G1

Живые объекты эвакуируются (т. е. копируются или перемещаются) в одну или несколько выживших областей. Если порог старения достигнут, некоторые объекты переводятся в регионы старого поколения

A Young GC in G1



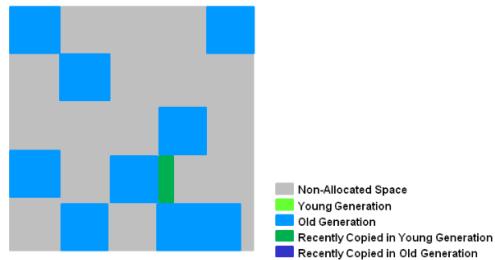
Происходит остановка мира (STW). Размер эдема и размер выжившего рассчитывается для следующей малой сборки. Для расчета размера сохраняется учетная информация. Учитываются такие вещи, как цель остановки.

Такой подход позволяет легко изменять размеры регионов, делая их больше или меньше по мере необходимости.

Конец малой сборки с G1

Живые объекты были эвакуированы в выжившие регионы или в регионы старого поколения.

End of Young GC with G1



Недавно продвинутые объекты показаны темно-синим цветом. Выжившие регионы – зеленым.

Отличия малой сборки

В целом о малой сборке в G1 можно сказать следующее:

- **Куча** – это единое пространство памяти, разбитое на области.
- Память молодого поколения состоит из набора несмежных областей. Это позволяет легко изменять ее размер при необходимости.
- Сборки мусора молодого поколения, или малые сборки, представляют собой события типа *остановки мира*. Все потоки приложения останавливаются для выполнения этой операции.
- Малая сборка выполняется параллельно с использованием нескольких потоков.
- Живые объекты копируются в новые регионы выжившего или старого поколения.

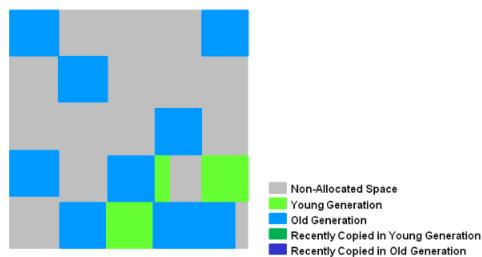
Большая сборка с помощью G1

Как и коллектор CMS, коллектор G1 предназначен для сбора объектов старого поколения с небольшой паузой. В следующей таблице описаны фазы сборки G1 для старого поколения.

Фаза начальной маркировки

Первоначальная разметка живого объекта происходит на основе сборки мусора молодого поколения.

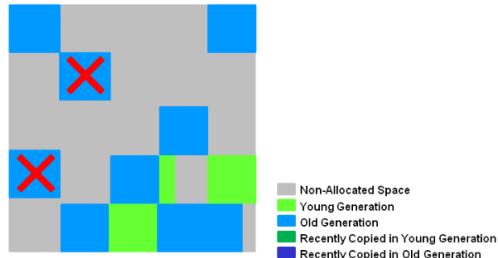
Initial Marking Phase



Фаза одновременной разметки

Если обнаружены пустые области (обозначенные символом "X"), они немедленно удаляются в фазе ремарки. Также вычисляется "учетная" информация, определяющая актуальность.

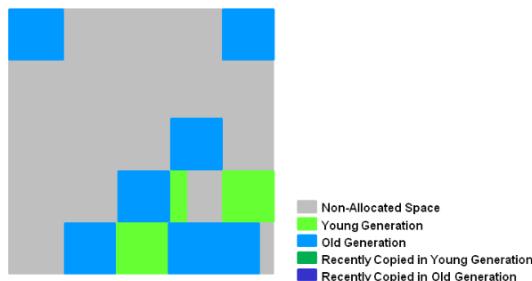
Concurrent Marking Phase



Фаза ремарки

Пустые регионы удаляются и восстанавливаются. Живучесть региона теперь рассчитывается для всех регионов.

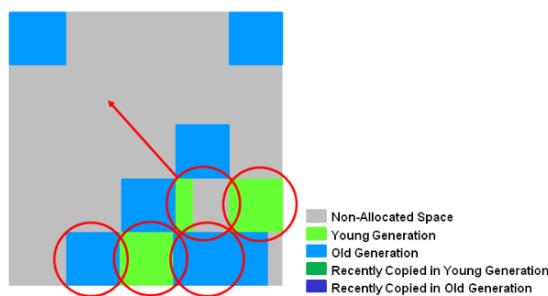
Remark Phase



Фаза копирования/очистки

G1 выбирает регионы с наименьшей "живостью", те регионы, которые можно собрать быстрее всего. Затем эти регионы собираются одновременно с малой сборкой. Таким образом, одновременно собираются и молодое, и старое поколения.

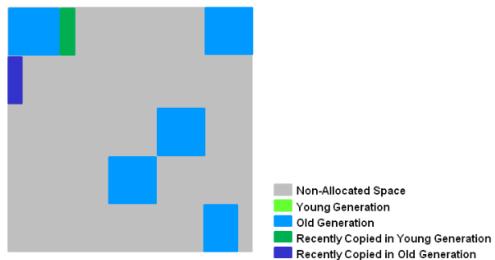
Copying/Cleanup Phase



После этапа копирования/очистки

Выбранные регионы были собраны и уплотнены в темно-синий и темно-зеленый регионы, показанные на диаграмме.

After Copying/Cleanup Phase



Отличия большой сборки

Вывод о GC старого поколения(большой сборки)

Подводя итог, можно выделить несколько ключевых моментов, связанных со сборкой мусора G1 на старом поколении(большой сборке).

- Параллельная фаза маркировки
Информация об актуальности вычисляется параллельно во время работы приложения.
Эта информация о живучести определяет, какие регионы лучше всего освободить во время паузы эвакуации.
Фаза зачистки, как в CMS, отсутствует.
- Фаза ремарки
Используется алгоритм Snapshot-at-the-Beginning (SATB), который намного быстрее, чем тот, что применялся в CMS.
Полностью пустые области восстанавливаются.
- Фаза копирования/очистки
Молодое поколение и старое поколение восстанавливаются одновременно.
Регионы старого поколения выбираются на основе их актуальности.

Билет 21

Вопрос 1: Объектный модуль. Объектная библиотека. Структуры данных, содержащиеся в объектном модуле, в общих чертах. Алгоритм работы сборщика и выбора модулей из архивной библиотеки.

Вопрос 2: Структура и принципы работы файловой системы NTFS.

Билет 22

Вопрос 1: Приоритеты процессов и нитей. Управление приоритетами для нитей реального и разделенного времени. Где используется и для чего нужно динамическое изменение приоритета?

РЕКОМЕНДУЮ ПРЕЗЕНТАЦИЮ ОБЯЗАТЕЛЬНО ПОСМОТРЕТЬ

Многопользовательские интерактивные системы, или, что то же самое,

системы разделенного времени (shared time), которые максимизируют количество событий, обрабатываемых в единицу времени или предоставляют хорошее среднее время реакции на события.

Иртегов с 450

В многозадачных системах часто возникает вопрос: в каком порядке исполнять готовые процессы? Как правило, бывает очевидно, что одни из процессов важнее других. Например, в системе может существовать три процесса, имеющих готовые к исполнению нити: процесс – сетевой файловый сервер, интерактивный процесс – текстовый редактор и процесс, занимающийся плановым резервным копированием с диска на ленту. Очевидно, что хотелось бы в первую очередь разобраться с сетевым запросом, затем – отреагировать на нажатие клавиши в текстовом редакторе, а резервное копирование может подождать сотню-другую миллисекунд. С другой стороны, мы должны защитить пользователя от ситуаций, в которых какой-то процесс вообще не получает управления, потому что система постоянно занята более приоритетными заданиями.

Самым простым и наиболее распространенным способом распределения процессов по **приоритетам** является организация нескольких очередей в соответствии с приоритетами. При этом процесс из низкоприоритетной очереди получает управление тогда и только тогда, когда все очереди с более высоким приоритетом пусты.

Легко понять, что разделение времени обеспечивает более или менее справедливый доступ к процессору для задач с одинаковым приоритетом. Современные ОС как общего назначения, так и реального времени, имеют много уровней приоритета.

В системах с пакетной обработкой, когда для задачи указывают верхнюю границу времени процессора, которое она может использовать, часто более короткие задания идут с более высоким приоритетом. Кроме того, более высокий приоритет дают задачам, которые требуют меньше памяти.

В **системах разделенного времени** часто оказывается сложно заранее определить время, в течение которого будет работать задача.

Так, если программа начала вычисления, не прерываемые никакими обращениями к внешней памяти или терминалу, мы можем предположить, что она будет заниматься такими вычислениями и дальше. Напротив, если программа сделала несколько запросов на ввод-вывод, следует ожидать, что она и дальше будет активно выдавать такие запросы. Предпочтительными для системы будут те программы, которые захватывают процессор на короткое время и быстро отдают его, переходя в состояние ожидания внешнего или внутреннего события. Таким процессам система стремится присвоить более высокий приоритет. Если программа ожидает завершения запроса на обращение к диску, то это также выгодно для системы – ведь на большинстве машин чтение с диска и запись на него происходят параллельно с работой центрального процессора.

Таким образом, система **динамически** повышает приоритет тем заданиям, которые освободили процессор в результате запроса на ввод-вывод или ожидание события и, наоборот, снижает тем заданиям, которые были сняты по истечении кванта времени. Однако приоритет не может превысить определенного значения – стартового приоритета задачи.

При этом наиболее высокий приоритет автоматически получают интерактивные задачи и программы, занятые интенсивным вводом-выводом.

На первый взгляд, не очень понятно, зачем это нужно. Впрочем, вспомним **определение системы разделенного времени**, которое мы приводили во введении, — это система, оптимизирующая среднее время реакции на запрос пользователя. В современных многопоточных системах реакция на пользовательские запросы осуществляется за счет пробуждения нитей, ждущих соответствующих событий. Таким образом, чтобы улучшить время реакции, системе выгодно повышать приоритет для тех нитей, которые чего-то ожидают.

Напротив, нити, которые снимаются по истечению кванта времени, по определению ничего не ожидают и, как следствие, не вносят вклада в наблюдаемое пользователем время реакции.

Таким образом, чтобы удовлетворить пользователя, разработчик интерактивного приложения должен обеспечить быструю — меньше кванта времени планировщика — обработку тех действий, на которые пользователь ожидает быстрой реакции, и обеспечить адекватную обратную связь для действий, которые физически невозможно реализовать за такое время. В многопоточных приложениях такие действия можно вынести в фоновый поток.

Легко понять, что работа динамического приоритизатора систем разделенного времени состоит именно в том, чтобы повысить приоритет задачам и нитям, непосредственно обслуживающим интерактивные функции за счет тех задач и нитей, завершения которых пользователь готов подождать.

Большинство старых ОС, такие, как OS/390, VAX/VMS, Unix, не делают различия между интерактивными и просто ориентированными на ввод/вывод задачами и повышают приоритет всем задачам, которые блокируются на операциях ввода/вывода и межпроцессного взаимодействия.

В Unix-системах данная проблема решается очень просто — система никогда не повышает приоритет задачи выше некоторого базового уровня. На практике, в Unix-системах даже не говорят о динамическом повышении приоритета.

Обычно работу динамической приоритизации в Unix описывают следующим образом. Приоритет процесса складывается из

статического приоритета (nice level) и **штрафа за использование процессора** (CPU penalty).

Оба эти значения представляют собой положительные целые числа; чем больше их сумма, тем ниже приоритет задачи. **Nice level** наследуется от родительского процесса или, если это необходимо, повышается запуском задачи с использованием команды shell, которая называется nice. Если это совсем уж необходимо, задача может повысить свой nice level с помощью системного вызова, который также называется nice. Простой пользователь может только повышать nice level своих задач, а суперпользователь (администратор системы) может также и понижать.

Штраф начисляется, когда система отнимает у задачи управление по исчерпанию кванта времени, и сбрасывается, когда задача сама уступает процессор.

Нужно отметить, что процесс разделенного времени может повысить свой приоритет до максимального в классе разделения времени, но никогда не сможет стать процессом реального времени. А для процессов реального времени динамическое изменение приоритетов обычно не применяется.

В Unix System V Release 4 процессы могут принадлежать к разным **классам планирования**. Класс планирования определяет **диапазон приоритетов**, **квант времени** и **политику динамической приоритизации**. В старых версиях системы поддерживалось три класса планирования: реального времени (RT), системный и разделенного времени (TS). При этом приоритет процессов RT был больше системных, а системных — больше TS. Для процессов реального времени используется фиксированная приоритизация, для TS — динамическая приоритизация с приоритетом, складывающимся из nice level и CPU penalty.

Дальше глава про честный планировщик, наверное, стоит прочитать.

Вопрос 2: Права доступа к файлам в ОС семейства Unix.

Иртегор с 780

В этих системах ACL состоит ровно из трех записей (рис. 12.16).

- Права хозяина файла (пользователя);
- Права группы;
- Права по умолчанию.

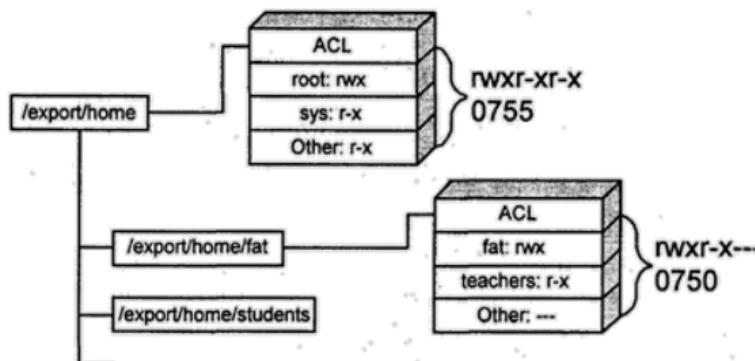


Рис. 12.16. Установление прав в системах семейства Unix

Как правило, права хозяина выше прав группы, а права группы выше прав по умолчанию, но это не является обязательным требованием и никем специально не проверяется. Пользователь может принадлежать к нескольким группам одновременно, файл всегда принадлежит только одной группе.

Бывают три права: **чтения, записи и исполнения**. Для каталога право исполнения означает право на открытие файлов в этом каталоге. Каждое из прав обозначается битом в маске прав доступа, т. е. все три группы прав представляются девятью битами или тремя восьмеричными цифрами.

Права на удаление или переименование файла не существует; вообще, в Unix не определена операция удаления файла как таковая, а существует лишь операция удаления имени **unlink**. Это связано с тем, что файл в Unix может иметь несколько имен, и собственно удаление происходит только при уничтожении последнего имени. Для удаления, изменения или создания нового имени файла достаточно иметь право записи в каталог, в котором это имя содержится.

Удаление файлов из каталога, в действительности, можно в определенных пределах контролировать: установка дополнительного бита **sticky** (маска прав содержит не девять, а двенадцать бит, setuid setgid и sticky bit") запрещает удаление из каталога чужих файлов. Обладатель права записи в такой каталог может создавать в нем файлы и удалять их, но только до тех пор, пока они принадлежат ему.

Кроме прав, перечисленных в маске, хозяину файла разрешается изменять права на файл: модифицировать маску прав и передавать файл другой группе и, если это необходимо, другому пользователю (в системах с дисковыми квотами передавать файлы обычно запрещают).

Еще один обладатель прав на файл, не указанный явно в его ACL, – это администратор системы, пользователь с идентификатором, равным 0. Этот пользователь традиционно имеет символическое имя **root**. Полномочия его по отношению к файлам, другим объектам и системе в целом правильнее описать даже не как обладание всеми правами, а как возможность делать с представленными в системе объектами что угодно, не обращая внимания на права.

В традиционных системах семейства Unix все глобальные объекты – внешние устройства и именованные программные каналы – являются файлами (точнее, имеют имена в файловой системе) и управление доступом к ним выполняется файловым механизмом. В современных версиях Unix адресные пространства исполняющихся процессов также доступны как файлы в специальной файловой (или псевдофайловой, если угодно) системе **proc**. Файлы в этой ФС могут быть использованы, например, отладчиками для доступа к коду и данным отлаживаемой программы . Управление таким доступом также осуществляется стандартным файловым механизмом.

В Unix System V появились объекты, не являющиеся файлами и идентифицируемые численными ключами доступа вместо имен, а именно средства межпроцессного взаимодействия: это семафоры, очереди сообщений и сегменты разделяемой памяти. Каждый такой объект имеет маску прав доступа, аналогичную файловой, и доступ к нему контролируется точно так же, как и к файлам.

Основное преимущество этого подхода состоит в его простоте. Фактически это наиболее простая из систем привилегий, пригодная для практического применения. Иными словами, более простые и ограниченные системы установления привилегий, по-видимому, непригодны вообще.

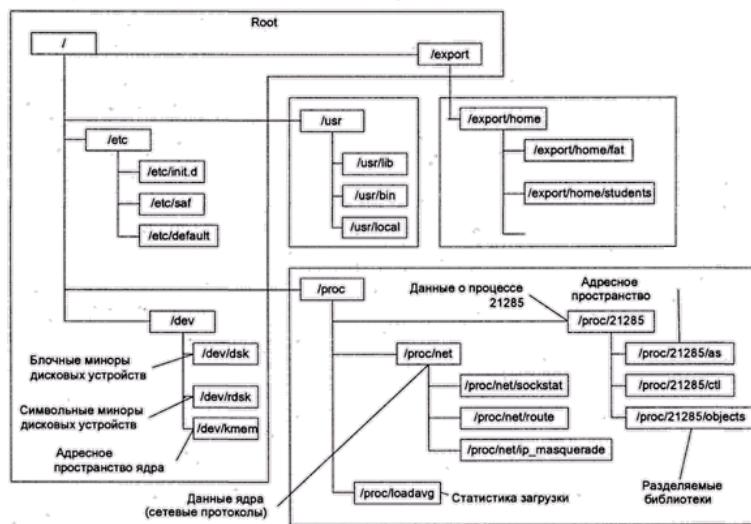


Рис. 12.17. Дерево каталогов Unix

Многолетний опыт эксплуатации систем, использующих эту модель, показывает, что она вполне адекватна подавляющему большинству реальных ситуаций. Впрочем, ряд современных файловых систем в ОС семейства Unix предоставляет произвольного вида списки для управления доступом.

Билет 23

Вопрос 1: Системы управления доступом. Полномочия и списки контроля доступа. Кольца доступа.

Системы управления доступом

Если честно, не особо понимаю, что писать в первом пункте. Ну попробуем.

Иртегов с 764

Задачу управления доступом к данным и операциям над ними разбивают на две основные подзадачи: **аутентификацию** (проверку, что пользователь системы действительно является тем, за кого себя выдает) и **авторизацию** (проверку, имеет ли тот, за кого себя выдает пользователь, право выполнять данную операцию).

Иртегов 13.2 Сессии и индентификаторы пользователя

Как правило, далеко не каждая авторизация отдельных операций сопровождается актом аутентификации. Чаще всего используется принцип сессий работы с вычислительной системой. В начале работы пользователь устанавливает соединение и "ходит" в систему. При "ходе" происходит его аутентификация.

Для того чтобы быть аутентифицированным, пользователь должен иметь **учетную запись (account)** в системной базе данных. Затем пользователь проводит сеанс работы с системой, а по завершении этого сеанса аннулирует регистрацию.

Одним из атрибутов сессии является **идентификатор пользователя (user id)** или **контекст доступа (security context)**, который используется при последующих авторизациях. Обычно такой идентификатор имеет две формы: числовой код, применяемый внутри системы, и мнемоническое символьное имя, используемое при общении с пользователем.

Далее рассказывается про сессии в Юникс. Вообще не факт что это нужно, но прочитать стоит.

Большинство современных ОС позволяют также запускать задания без входа в систему и создания сессии. Так, практически все системы разделения времени предоставляют возможность пользователям запускать задачи в

заданные моменты астрономического времени периодически, например, в час ночи в пятницу каждой недели. Каждая такая задача исполняется от имени определенного пользователя – того, кто запросил запуск задачи.

Для управления правами доступа в таких ситуациях идентификатор пользователя ассоциируется не с сессией, а с отдельными заданиями, а обычно даже с отдельными задачами. В Windows NT/2000/XP задачи, которые могут запускаться и работать без входа пользователя в систему, называются **сервисами**. По умолчанию, сервисы запускаются от имени специального псевдо-пользователя System, но в свойствах сервиса можно указать, от чьего имени он будет запускаться. Кроме того, некоторые комплектации системы допускают одновременную интерактивную работу нескольких пользователей.

Чтобы обеспечить разделение доступа во всех этих случаях, каждый процесс в системе имеет **контекст доступа (security context)**, соответствующий той или иной учетной записи.

Аутентификация

Понятно, что если права доступа выделяются на основе машинного идентификатора пользователя, то возникает отдельная проблема установления соответствия между этим машинным идентификатором и реальным человеком.

По-английски процесс входа в систему называется **login (log in)** и происходит от слова **log**, которое обозначает регистрационный журнал или процесс записи в такой журнал. В обычном английском языке такого слова нет, но в компьютерной лексике слова **login** и **logout** прижились очень прочно.

Наиболее точным переводом слова **login** является регистрация. Соответственно, процесс выхода называется **logout**.

Наиболее широкое распространение получил более простой метод идентификации, основанный на символьных паролях.

Пароль представляет собой последовательность символов. Предполагается, что пользователь запоминает ее и никому не сообщает. Этот метод хорош тем, что для его применения не нужно никакого дополнительного оборудования – только клавиатура, которая может использоваться и для других целей. Но этот метод имеет и ряд недостатков.

Использование паролей основано на следующих трех предположениях:

- пользователь может запомнить пароль;
- никто не сможет догадаться, какой пароль был выбран;
- пользователь никому не сообщит свой пароль.

Современная техника выбора паролей обеспечивает достаточно высокую для большинства практических целей безопасность.

При использовании паролей возникает отдельная проблема безопасного хранения базы данных со значениями паролей.

Для обеспечения секретности паролей обычно используют одностороннее шифрование, или **хэширование**, при котором по зашифрованному значению нельзя восстановить исходное слово. При этом программа аутентификации кодирует введенный пароль и сравнивает полученное значение (хэш) с хранящимся в базе данных.

Авторизация Иртегов с 772

Механизмы авторизации в различных ОС и прикладных системах различны, но их трудно назвать разнообразными. Два основных подхода к авторизации – это **ACL (Access Control List**, список управления доступом) или **список контроля доступа и полномочия (capability)**.

Списки контроля доступа Иртегов с 773

Список контроля доступа ассоциируется с объектом или группой объектов и представляет собой таблицу, строки которой соответствуют учетным записям пользователей, а столбцы – отдельным операциям, которые можно

осуществить над объектом. Перед выполнением операции система ищет идентификатор пользователя в таблице и проверяет, указана ли выполняемая операция в списке его прав.

В общем случае совокупность всех ACL в системе представляет собой трехмерную матрицу, строки которой соответствуют пользователям, столбцы – операциям над объектами, а слои – самим объектам. С ростом количества объектов и пользователей в системе объем этой матрицы быстро растет, поэтому, как уже говорилось, разработчики реальных систем контроля доступа предпринимают те или иные меры для более компактного представления матрицы.

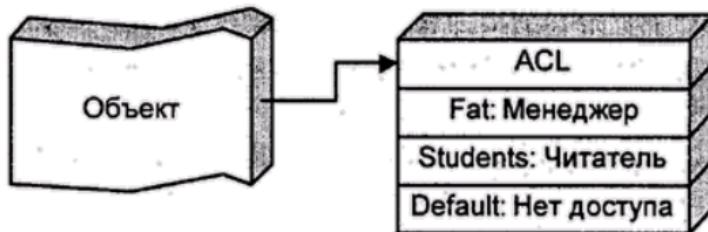


Рис. 12.9. Список контроля доступа

Есть три основных подхода, используемых для сокращения ACL.

- Использование прав по умолчанию
- Группирование пользователей и/или объектов
- Ограничение комбинаций прав, которыми пользователи и группы могут реально обладать

Основное **преимущество** этого подхода состоит в его простоте. Фактически это наиболее простая из систем привилегий, пригодная для практического применения. Иными словами, более простые и ограниченные системы установления привилегий, по-видимому, непригодны вообще.

Полномочия Иртегов с 783

Полномочие представляет собой абстрактный объект, наличие которого в контексте доступа задачи позволяет выполнять ту или иную операцию над защищаемым объектом или классом объектов, а отсутствие – соответственно, не позволяет. При реализации такой системы разработчик должен гарантировать, что пользователь не сможет самостоятельно сформировать полномочие.

Например, полномочие может быть реализовано в виде ключа шифрования или электронной подписи. Невозможность формирования таких полномочий обеспечивается непомерными вычислительными затратами, которые нужны для подбора ключа.

Типичная практически используемая архитектура управления доступом предоставляет пользователю и системному администратору управление правами в форме списков контроля доступа и содержит один или несколько простых типов полномочий, чтобы предотвратить доступ в обход этих списков. Простейшей структурой таких полномочий является разделение **пользовательского и системного** режимов работы процессора и исполнение всего не пользующегося доверием кода в пользовательском режиме.

КРИТ

Системный режим процессора является полномочием или, во всяком случае, может применяться в качестве такового: обладание им позволяет выполнять операции, недопустимые в пользовательском режиме, и этот режим не может произвольно устанавливаться. Он позволяет реализовать не только ACL, но и дополнительные полномочия: пользователь не имеет доступа в системное адресное пространство, поэтому система может рассматривать те или иные атрибуты дескриптора пользовательского процесса как полномочия.

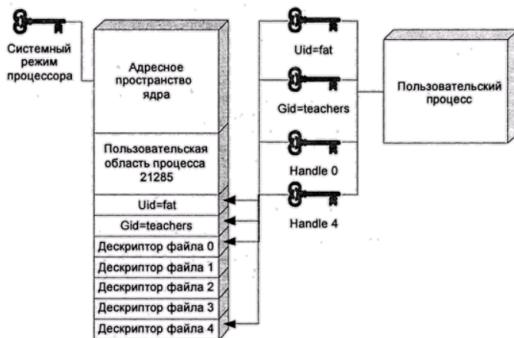


Рис. 12.18. Хранение полномочий в системном адресном пространстве

Многие ОС предоставляют и полномочия доступа к отдельным объектам – для того, чтобы не проверять ACL объекта при каждой операции. Так, в системах семейства Unix права доступа к файлам проверяются только в момент открытия. При открытии необходимо указать желаемый режим доступа к файлу: только чтение, только запись или чтение/запись. После этого пользователь получает "ручку" – индекс дескриптора открытого файла в системных таблицах.

Ручка представляет собой целое число и не имеет смысла в отрыве от соответствующего ей дескриптора, зато дескриптор является типичным полномочием: он недоступен пользовательскому коду непосредственно, потому что находится в системном адресном пространстве. Дескриптор может быть сформирован только системным вызовом `open` и допускает только те операции над файлом, которые были запрошены при открытии. Во время выполнения операций проверка прав доступа не производится (хотя, конечно, проверяется их физическая выполнимость: наличие места на устройстве и т. д.), так что если мы изменим ACL файла, это никакие не повлияет на права процессов, открывших файл до этой модификации.

Кольца доступа

КРИТ

Тут вообще в книге ничего нет.

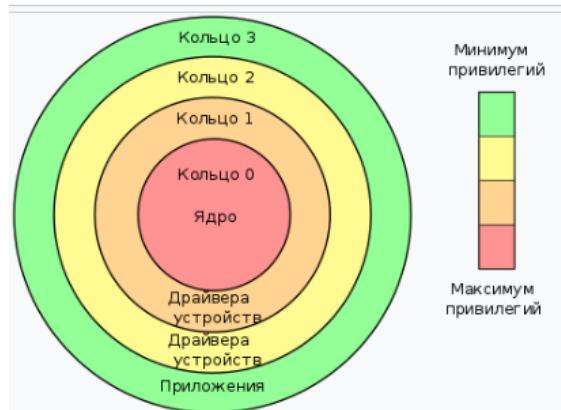
Ну надо сказать определения.

Хабр

Википедия:

Кольца защиты — архитектура информационной безопасности и функциональной отказоустойчивости, реализующая аппаратное разделение системного и пользовательского уровней привилегий.

Структуру привилегий можно изобразить в виде нескольких концентрических кругов. В этом случае системный режим ([режим супервизора](#) или нулевое кольцо, так называемое «кольцо 0»), обеспечивающий максимальный доступ к ресурсам, является внутренним кругом, тогда как режим пользователя с ограниченным доступом — внешним. Традиционно семейство микропроцессоров [x86](#) обеспечивает четыре кольца защиты.



Большинство UNIX-систем использует только 2 кольца, даже если аппаратные средства поддерживают больше режимов центрального процессора.

Дальше книга с 307 можно почитать раньше про VAX кольца с 301 и 80286 с 305

Многоуровневый доступ, основанный на концепции колец, не имеет принципиальных преимуществ по сравнению с двумя уровнями привилегий. Как и в двухуровневой системе, пользовательские модули вынуждены полностью доверять привилегированным.

С точки зрения безопасности, основной проблемой систем с кольцами защиты является неспособность таких систем защитить себя от ошибок в модулях, исполняющихся в высшем кольце защиты.

Билет 24.

Вопрос 1: Планировщики разделенного времени. Динамическое управление приоритетами в системах разделенного времени.

Ну э. Опять таки Иртегов с 431. и про приоритеты с 450

РЕКОМЕНДУЮ ПРЕЗЕНТАЦИЮ ОБЯЗАТЕЛЬНО ПОСМОТРЕТЬ.

Планировщики разделенного времени

Приоритеты

Переключение процессов по таймеру в сочетании с некоторыми другими приемами, такими как **динамическая приоритизация** (рассматривается в разд. 8.2.1), позволяет реализовать параллельную работу нескольких пользователей или пользовательских задач так, что каждый пользователь получает определенную долю процессорного времени. Со многих точек зрения каждый из этих пользователей может считать, что ему доступен свой собственный компьютер, производительность которого составляет определенную долю от реального.

Вопрос 2: Структура и особенности организации файловой системы UFS (FFS).

Билет 25

Вопрос 1: Кооперативная и вытесняющая (preemptive) многозадачность. Преимущества и недостатки обоих архитектур.

КРИТ

РЕКОМЕНДУЮ ПРЕЗЕНТАЦИЮ ОБЯЗАТЕЛЬНО ПОСМОТРЕТЬ

Если начало нужно.

Иртегов с 432

Кооперативная многозадачность, преимущества и недостатки.

Планировщик, основанный на **Threadswitch**, т. е. на принципе переключения по инициативе активной нити, используется в ряде экспериментальных и учебных систем. Этот же принцип, называемый **кооперативной многозадачностью**, реализован в библиотеках языков Simula 67 и Modula-2. MS Windows 3.x также имеют средство для организации кооперативного переключения задач – системный вызов **GetNextEvent**.

Часто кооперативные нити называют не нитями, а **сопрограммами** – ведь они вызывают друг друга, подобно подпрограммам. Единственное отличие такого вызова от вызова процедуры состоит в том, что такой вызов не иерархичен – вызванная программа может вновь передать управление исходной и оставаться при этом активной.

Основным **преимуществом** кооперативной многозадачности является простота отладки планировщика. Кроме того, снимаются все коллизии, связанные с критическими секциями и тому подобными трудностями – ведь нить может просто не отдавать никому управления, пока не будет готова к этому.

С другой стороны, кооперативная многозадачность имеет и серьезные **недостатки**.

Во-первых, необходимость включать в программу вызовы **Threadswitch** усложняет программирование вообще и перенос программ из однозадачных или иначе организованных многозадачных систем в частности.

Особенно неприятно требование регулярно вызывать **Threadswitch** для вычислительных программ. Чаще всего такие программы исполняют относительно короткий внутренний цикл, скорость работы которого определяет скорость всей программы. Для "плавной" многозадачности необходимо вызывать **Threadswitch** из тела этого цикла. Делать вызов на каждом шаге Цикла нецелесообразно, поэтому необходимо будет написать код, похожий на приведенный в примере 8.2.

Пример 8.2. Внутренний цикл программы в кооперативно многозадачной среде.

```
int counter; // Переменная-счетчик,
while(condition) {
    // Вызывать ThreadSwitch каждые rate циклов.
    counter++;
    if (counter % rate == 0) ThreadSwitch();
    ... // Собственно вычисления j
}
```

Условный оператор и вызов функции во внутреннем цикле сильно усложняют работу оптимизирующими компиляторами и приводят к разрывам конвейера команд, что может очень заметно снизить производительность. Вызов функции на каждом шаге цикла приводит к еще большим накладным расходам и, соответственно, к еще большему замедлению.

Во-вторых, злонамеренная нить может захватить управление и никому не отдавать его. Просто не вызывать **ThreadSwitch**, и все. Это может произойти не только из-за злых намерений, но и просто по ошибке.

Поэтому такая схема оказывается непригодна для многопользовательских систем и часто не очень удобна для интерактивных однопользовательских.

В-третьих, кооперативная ОС не может исполняться на симметричной многопроцессорной машине, а приложения, написанные в расчете на такую ОС, не могут воспользоваться преимуществами многопроцессорности.

Простой анализ показывает, что кооперативные многозадачные системы пригодны только для учебных проектов или тех ситуаций, когда программисту на скорую руку необходимо сотворить многозадачное **ядро**.

Вытесняющая многозадачность

Крит

Все вышесказанное подводит нас к идее вызывать **ThreadSwitch** не из пользовательской программы, а каким-то иным способом. Например, поручить вызов такой функции прерыванию от системного таймера. Тогда мы получим следующую схему.

- Каждой нити выделяется **квант времени**.
- Если нить не освободила процессор в течение своего кванта, ее снимают и переставляют в конец очереди.
При этом все готовые к исполнению нити более или менее равномерно получают управление.

Этот механизм, называемый **time slicing** или **разделение времени** практически во всех современных ОС. Общим названием для всех методов переключения нитей по инициативе системы является термин **вытесняющая (preemptive) многозадачность**. Таким образом, вытесняющая многозадачность противопоставляется кооперативной, в которой переключение происходит только по инициативе самой задачи. Разделение времени является частным случаем вытесняющей многозадачности. В системах с приоритетами, вытеснение текущей задачи происходит не только по сигналам таймера, но и в случае, когда по каким-то причинам (чаще всего из-за внешнего события) активизируется процесс, с приоритетом выше, чем у текущего.

При этом вопрос выбора кванта времени является нетривиальной проблемой. С одной стороны, чрезмерно короткий квант приведет к тому, что большую часть времени система будет заниматься переключением потоков. С другой стороны, в интерактивных системах или системах реального времени слишком большой квант приведет к недопустимо большому времени реакции.

В системе реального времени мы можем объявить нити, которым надо быстро реагировать, высокоприоритетными и на этом успокоиться. Однако нельзя так поступить с интерактивными программами в многопользовательской или потенциально многопользовательской ОС, как UNIX на настольной машине x86 или Sun.

Системы реального времени обычно имеют два класса планирования — реального и разделенного времени. **Класс планирования**, как правило, дается не отдельным нитям, а целиком процессам. Процессы реального времени не прерываются по сигналам таймера и могут быть вытеснены только активизацией более приоритетной нити реального времени. Нити реального времени высочайшего приоритета фактически работают в режиме кооперативной многозадачности. Зато нити процессов разделенного времени вытесняются и друг другом по сигналам таймера, и процессами реального времени по мере их активизации.

Вытесняющая многозадачность имеет много преимуществ, но если мы про сто будем вызывать описанный в предыдущем разделе **ThreadSwitch** по прерываниям от таймера или другого внешнего устройства, то такое переключение будет неправильно нарушать работу прерываемых нитей.

Действительно, пользовательская программа может использовать какой-то из регистров, который не сохраняется при обычных вызовах. Поэтому, например, обработчики аппаратных прерываний сохраняют в стеке все используемые ими регистры.

Полный набор регистров, которые нужно сохранить, чтобы нить не заметила переключения, называется **контекстом нити** или, в зависимости от принятой в конкретной ОС терминологии, **контекстом процесса**. К таким регистрам, как минимум, относятся все регистры общего назначения, указатель стека, счетчик команд и слово состояния процессора.

Для реализации вытеснения достаточно сохранить контекст текущей нити и загрузить контекст следующей активной нити из очереди.

Необходимо предоставить также и функцию переключения нитей по их собственной инициативе, аналогичную **ThreadSwitch** или, точнее, **DeactivateThread**.

Вытесняющий планировщик с разделением времени ненамного сложнее кооперативного планировщика.

Преимущества

Лишаемся тех недостатков, которые имеет кооперативная многозадачность.

Недостатки

Беды с сохранение контекста.

Вопрос 2: Файловая система ISO 9660 (CDFS).

Билет 26

Вопрос 1: Троянские программы и способы их внедрения. Меры по защите от троянских программ.

Иртегов с 759

Троянская программа, т. е. помещение в атакуемую систему написанного взломщиком кода. Код при этом размещается таким образом, чтобы его рано или поздно должны были запустить. Поскольку код системы обычно так или иначе защищен, внедрение такой программы часто само по себе является атакой.

Иртегов с 809

Название этого типа атак происходит от известной легенды о статуе коня, которую греки использовали для проникновения в стены города Троя во время воспетой Гомером Троянской войны. Троянские программы или, короче, "**трояны**" (trojan) представляют очень большую опасность, потому что исполняются они с **привилегиями** тех пользователей, которые имели несчастье их запустить, и имеют доступ ко всем данным этого пользователя.

Если троянскую программу удается запустить от имени системного сервиса, то автор трояна может получить **полный контроль** над системой.

Вред, который может причинить троянская программа, ограничен только фантазией ее разработчика. Из наиболее неприятных возможностей следует упомянуть полное уничтожение всех доступных данных или их анализ и пересылку результатов анализа автору или заказчику трояна. В некоторых случаях результат деятельности троянской программы выглядит как целенаправленная диверсия со стороны пользователя, что может повести расследование инцидента по ложному пути.

Троянская программа может быть реализована не только в виде самостоятельного загрузочного модуля, но и в виде разделяемой библиотеки

Многие троянские программы образуются модификацией присутствующих в системе загрузочных модулей, разделяемых библиотек и даже модулей ядра.

Такая программа не обязательно должна представлять собой бинарный код — это может быть также интерпретируемый код или последовательность команд макропроцессора.

Способы их внедрения

В зависимости от способа внедрения, трояны можно разделить на четыре основных класса.

- **Встраиваемые в систему при ее разработке.** В зависимости от намерений программиста их можно подразделить еще на три класса:
 1. "закладки", т. е. сознательно внедряемый вредоносный код;
 2. "задние двери" (backdoor) — отладочные инструментальные средства, которые позволяют разработчику получать привилегии нештатными способами;
 3. ошибки программирования.
- **Добавляемые в систему при ее распространении.** От этих типов троянских программ особенно сильно страдают бесплатные и условно-бесплатные программные пакеты, распространяемые по Интернету. Взломав

сервер, с которого распространяется дистрибутив программы, злоумышленник может внедрить в него троянский код.

- **Внедряемые в уже работающую систему путем физического доступа к этой системе**
, в том числе и путем установки в систему дополнительного оборудования.
- **Внедряемые в уже работающую систему с помощью несанкционированного удаленного доступа.**
Поскольку обычно код работающей системы так или иначе защищается от модификации, для того чтобы что-то подобное могло произойти, в системе уже должны присутствовать какие-то проблемы с безопасностью.
- **Внедряемые в уже работающую систему с согласия пользователя** (часто при этом пользователя целенаправленно вводят в заблуждение, так что он не совсем ясно представляет себе, на что именно он согласился).

Далее идет классификация по тому что они делают но это юзлесс.

ВПЕРЕДИ МИЛЛИАРД ИНФЫ, НЕ ФАКТ ЧТО ВСЁ НУЖНО, ЛУЧШЕ ПРОЧИТАТЬ ПО ДИАГОНАЛИ, НО КОНЦОВКУ ТОЧНО НАДО!!!.

Описание и защита

Иртегов с 813

Тут про закладку Томсона не факт что нужно

Закладки

Умышленно внедряемые при разработке "закладки" или логические бомбы (logical bomb) представляют большую опасность. Как уже отмечалось, надежных технических средств защиты от них не существует. Тем не менее у крупнотиражных продуктов вопросы защиты от закладок так или иначе разрешаются, главным образом за счет репутации автора или издателя программы.

Дальше примеры опасности закладок и почему во внутреннем проекте все плохо

Задние двери

При разработке и тестировании системы разработчики часто нуждаются в нештатных средствах доступа к ней. Особенно это справедливо для систем, реализующих сложные и теоретически "непробиваемые" средства защиты от несанкционированного доступа.

Чаще всего такие средства реализуются в виде жестко закодированных учетных записей, которые не содержатся в штатной БД этих записей, но позволяют разработчику аутентифицироваться в системе.

Анализ исходных текстов относительно эффективен для обнаружения закладок такого типа, главным образом потому, что их обычно никак специально не прячут.

ПОТОМ ПРИМЕРЫ

Ошибки программирования

Ошибки программирования представляют собой одну из самых важных проблем для современной отрасли информационных технологий. До 90% стоимости разработки программ и 100% стоимости их поддержки приходится на поиск, исправление или обход ошибок. При всем этом рассуждения о полном искоренении ошибок из крупных программных комплексов следует признать утопическими и даже беспредметными.

Можно сделать вывод, что разработчикам, в общем-то, не нужно целенаправленно внедрять в программные комплексы никакого вредоносного кода —такого кода в коммерческих программах и без того предостаточно.

Троянские программы, внедряемые при распространении

Наиболее громкие из историй о троянском коде в популярных программах связаны именно с раздачей дистрибутивов по сети.

Справедливости ради, необходимо отметить, что большинство современных программных продуктов, распространяемых на компакт-дисках, в том числе MS Windows и MS Office, практически непригодны для эксплуатации без регулярно скачиваемых через Интернет "заплат" и обновлений.

Троянские программы, внедряемые в уже установленную систему

В начале раздела мы видели основную классификацию способов внедрения враждебного кода в уже установленную систему. Внедрение путем физического доступа к компьютеру мы, как уже договаривались, не будем подробно обсуждать. Впрочем, важно понимать, что в современных условиях всего не скольких секунд доступа к оставленному без присмотра компьютеру может оказаться достаточно, чтобы скачать из сети или скопировать с удаленного носителя троянскую программу довольно большого объема.

В современных условиях гораздо более распространены и, соответственно, гораздо более опасны ситуации, когда троянские программы внедряются в систему без физического доступа злоумышленника к ней. Два основных метода такого внедрения:

- запуск зараженной программы самим пользователем;
- удаленный доступ, например через срывы буфера или ошибки, допускающие внедрение скрипта, в сетевых сервисах или в программах, которые работают с внешними данными.

При этом, разумеется, первый метод обычно требует использования тех или иных социально-инженерных средств. Впрочем, у большинства пользователей не очень хорошо срабатывает интуиция, и они недооценивают опасность, сопряженную с запуском посторонних программ на своем компьютере. Поэтому зачастую не требуется такой уж сложной инженерии, чтобы уговорить пользователя открыть исполняемый файл, пришедший по почте с комментарием "прикольная картинка" или согласиться на установку контрола ActiveX, "необходимого" для просмотра порносайта.

Вирусы и аналогичные троянские программы

Наиболее важным примером троянов, внедряющихся в установленную систему посредством запуска пользователем, являются компьютерные вирусы.

Классические вирусы были очень распространены в первой половине 90-х годов.

Они распространялись, главным образом, с играми и другими программами — как нелицензионными, так и бесплатными и условно-бесплатными.

Дальше тут про вирусы

Самые простые вирусы активизировались только в момент запуска зараженной программы и прекращали свою работу, передав ей управление. Более сложные — так называемые **резидентные** (resident) — вирусы внедряли себя в ядро ДОС и могли активизироваться при каждом системном вызове.

Резидентные вирусы, в частности, могут перехватывать обращения к зараженным файлам и, если обращение происходит с целью чтения файла, а него загрузки, удалять из создаваемого в памяти образа файла все следы своего присутствия. В частности, благодаря этому, при антивирусном сканировании зараженной машины вирус не может быть обнаружен. Такие вирусы называются **stealth-вирусами**.

Stealth-вирусы впервые продемонстрировали широким массам компьютерных специалистов один печальный факт, который должен осознавать любой борец с троянскими программами: **если троян написан грамотно и исполняется с достаточно высокими привилегиями, то средствами зараженной системы его обнаружить невозможно**. Обычно этот факт формулируют иначе: **невозможно достоверно гарантировать "чистоту" системы средствами самой этой системы**.

Активный контент

Особенно большую опасность в качестве средства внедрения троянских программ представляет так называемый "активный контент", т. е. совмещение данных и кода, или, точнее, внедрение в данные различных "активных элементов", так или иначе реагирующих на действия пользователя при просмотре и/или модификации документа.

Черви и боты

Черви (worm) представляют собой троянский код, способный размножаться без участия пользователей системы. Заражению червями подвержены лишь многопоточные ОС с более или менее развитыми сетевыми сервисами. Заразив систему, червь запускает себя в качестве фонового процесса и начинает атаку других систем, **используя фиксированный набор известных проблем в их системах безопасности, чаще всего срывы буфера в сетевых сервисах.**

Боты аналогичны червям по способам размножения: они поражают уязвимые системы через срывы буфера и другие аналогичные ошибки. В отличие от червя, боты не автономны: заразив систему, они регистрируются на некотором центральном сервере и начинают принимать команды от этого сервера.

Владелец ботнета может затем управлять своими ботами с помощью относительно простого командного языка.

ЗАЩИТА

Черви, вирусы, боты и неразмножающиеся крупнотиражные троянские программы, такие как SPYWARE и ADWARE, представляют собой наиболее важную проблему для современного Интернета. Основным средством борьбы с ними являются антивирусные пакеты и их специализированные версии для борьбы с ADWARE/SPYWARE; впрочем, очевидно, что если ваша сеть подвержена вирусным атакам, а особенно если единственной защитой вашей сети от них служит **антивирусный пакет, способный только находить известные вирусы**, она будет столь же уязвима и для троянских программ, специально разработанных с целью атаки на данные вашей организации.

При этих условиях вашу систему безопасности следует признать абсолютно непригодной.

Ранее приводилась одна важная причина, по которой антивирус не может быть эффективен даже против крупнотиражных троянских программ: он не может защитить от сложных программ, внедряющихся в ядро системы и использующих STEALTH -технологии.

Кроме того, важно понимать, что производители антивирусных пакетов обычно на шаг, а иногда и больше, отстают от разработчиков вирусов, так что обновление баз антивируса часто появляется лишь после пика пандемии.

Поэтому, хотя установкой антивирусных фильтров, как и любым другим возможным эшелоном защиты, не следует пренебрегать, рассчитывать на них как на основное средство защиты ни в коем случае не следует. Основным средством защиты должна быть комплексная система безопасности, включающая в себя:

- продуманную структуру сети с расстановленными в ключевых местах фильтрующими маршрутизаторами и другими фильтрами
- доступ к разделяемым ресурсам, построенный по принципу минимально необходимых привилегий
- адекватные средства мониторинга сети и узлов;
- продуманные и надежные схемы восстановления узлов сети после аварий
- своевременную установку "заплат";
- и ряд других мероприятий

Дальше рекомендации на странице 837

Некоторые из приводимых рекомендаций уже упоминались в тексте, но здесь мы постараемся собрать их воедино.

Пользователь не должен иметь доступа к данным более того, что требуется для исполнения им служебных обязанностей. Это позволяет минимизировать вред от исполняемых пользователями троянских программ и прямых проникновений в систему от имени этого пользователя, а также вред, который пользователь может причинить сам, как сознательно, так и по ошибке. Кроме того, работа с ограниченным подмножеством данных удобнее для пользователя и иногда приводит к повышению производительности.

Везде, где это не приводит к чрезмерным накладным расходам, **следует применять шифрованные протоколы передачи данных.** Данные, хранящиеся на компьютерах за пределами здания компании, а особенно на домашних

и переносных компьютерах, следует шифровать в обязательном порядке.

Защита данных практически не имеет смысла без защиты самой системы и прикладного программного обеспечения: если злоумышленник имеет возможность модифицировать код системы или прикладной программы, он может встроить в него троянские подпрограммы, осуществляющие несанкционированный доступ к данным.

Доступ к коду приложений и системы для модификации должен предоставляться только техническому персоналу, занимающемуся поддержкой и установкой обновлений этих приложений. Это позволяет защититься не только от запускаемых пользователями троянских программ (особенно вирусных), но и от ошибочных действий пользователей.

Доступ к конфигурации ОС и прикладных программ также должен требовать высоких привилегий.

Когда это возможно, **следует защищать каталоги данных от исполнения** — это также может помочь защитить пользователей от некоторых способов внедрения троянских программ.

Каждый активный серверный или сервисный процесс, исполняемый в системе, может содержать ошибки и, таким образом, является потенциальной точкой атаки. Следовательно, **все сервисы, которые не нужны непосредственно для работы системы или реально используемых прикладных программ, необходимо остановить.** Это может также привести к некоторому повышению производительности.

Во всех случаях, когда это возможно, **следует делать операционную среду гетерогенной** — различные ОС и приложения имеют различные (и, как правило, непересекающиеся) наборы проблем с безопасностью, поэтому сложность взлома гетерогенной среды резко возрастает. Особенно это справедливо для автоматических и полуавтоматических взломщиков, таких как черви и боты.

Необходимо продумать схему мониторинга сети и узлов сети. Особенное внимание следует уделить схемам хранения лог-файлов ОС и приложений. Если это возможно, следует размещать логи не на той машине, которая их ведет, а на другой, что чрезвычайно затруднит подчистку логов после взлома. Необходимо также продумать схему ротации логов, т. е. уничтожения устаревших данных.

Системный администратор должен **следить за выходами обновлений и заплат, которые исправляют ошибки безопасности** — что, впрочем, не следует понимать как рекомендацию немедленно устанавливать на промышленно эксплуатируемые серверы самые последние "заплаты".

Необходимо также отметить, что интервал времени между выпуском публичного патча и его установкой большинством пользователей представляет собой самое "злачное" время для разработчиков malware.

При принятии решения об эксплуатации той или иной программной системы — операционной, прикладной или среды разработки — **необходимо ознакомиться с политикой поддержки**, которую предоставляет ее поставщик.

Хотя антивирусные пакеты и не являются адекватным средством защиты вашей сети от троянских программ, их применением не следует пренебрегать, особенно в средах, где активно используются приложения и ОС фирмы Microsoft. Лучше иметь хоть какую -то защиту, чем вообще никакой. В качестве дополнительного эшелона **защиты антивирусный пакет** также может оказаться полезен.

Вопрос 2: Асинхронный ввод-вывод в стандарте POSIX.

Реализация POSIX 1b

- aio_read, aio_write, aio_suspend
- Используют структуру aiocb
- Размещены в библиотеке librt.so
 - Компилировать с ключом -lrt
- Поддерживаются в Linux

Методичка Иртегова:

ПРО СИГНАЛЫ.

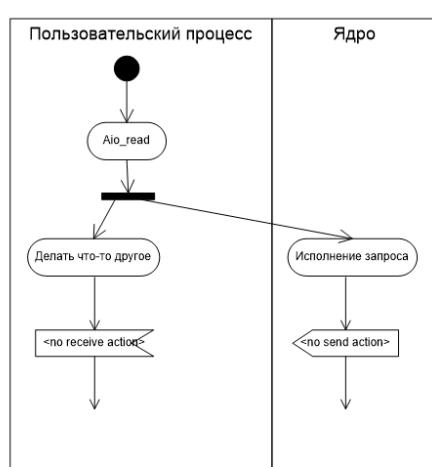
Немного теоретической части про асинхронный ввод-вывод

ТАБЛИЦА СИГНАЛОВ В ВИКИПЕДИИ

Асинхронный ввод/вывод от сайта manpages

В этом режиме системные вызовы ввода/вывода возвращают управление сразу **после формирования запроса к драйверу устройства**, как правило, даже до того, как данные будут скопированы в системный буфер.

Формирование запроса состоит в установке **записи** (IRP, Input/Output Request Packet, пакет запроса ввода вывода) в **очередь**. Для этого надо лишь ненадолго захватить mutex, защищающий «хвост» очереди, поэтому проблема инверсии приоритета легко преодолима. Для того, чтобы выяснить, закончился ли вызов, и если закончился, то чем именно, и можно ли использовать память, в которой хранились данные, предоставляется специальный API.



Библиотека POSIX AIO предусматривает два способа оповещения программы о завершении запроса, синхронный и асинхронный. Сначала рассмотрим синхронный способ.

Проверка статуса асинхронного запроса

Функция **aio_return(3AIO)** возвращает статус запроса. Если запрос уже завершился и завершился успешно, она возвращает размер прочитанных или записанных данных в байтах.

Как и у традиционного `read(2)`, в случае конца файла **aio_return(3AIO)** возвращает 0 байт. Если запрос завершился ошибкой или еще не завершился, возвращается -1 и устанавливается `errno`. Если запрос еще не завершился, код ошибки равен `EINPROGRESS`. Функция **aio_return(3AIO)** **разрушающая**; если ее вызвать для завершенного запроса, то она уничтожит системный объект, хранящий информацию о статусе запроса. Многократный вызов **aio_return(3AIO)** по поводу одного и того же запроса, таким образом, невозможен.

Функция **aio_error(3AIO)** возвращает код ошибки, связанной с запросом. При успешном завершении запроса возвращается 0, при ошибке – код ошибки, для незавершенных запросов – `EINPROGRESS`.

Функция **aio_suspend(3AIO)** блокирует нить до завершения одного из указанных ей запросов асинхронного ввода/вывода либо на указанный интервал времени. Эта функция имеет три параметра:

- `const struct aiocb *const list[]` – **массив указателей** на описатели запросов.
- `int nent` – **количество элементов в массиве list**.
- `const struct timespec *timeout` – **тайм-аут с точностью до наносекунд** (в действительности, с точностью до разрешения системного таймера).

Функция возвращает 0, если хотя бы одна из операций, перечисленных в списке, завершилась. Если функция завершилась с ошибкой, она возвращает -1 и устанавливает `errno`. Если функция завершилась по тайм-ауту, она также возвращает -1 и `errno==EINPROGRESS`.

```

Асинхронный ввод/вывод с синхронной проверкой статуса запроса.

Код сокращен, из него исключены открытие сокета и обработка ошибок.
const char req[]="GET / HTTP/1.0\r\n\r\n\r\n";
int main() {
    int s;
    static struct aiocb readrq;
    static const struct aiocb *readrv[2]={&readrq, NULL};
    /* Открыть сокет [...] */
    memset(&readrq, 0, sizeof readrq);
    readrq.aio_fildes=s;
    readrq.aio_buf=buf;
    readrq.aio_nbytes=sizeof buf;
    if (aio_read(&readrq)) {
        /* ... */
    }
    write(s, req, (sizeof req)-1);

    while(1) {
        aio_suspend(readrv, 1, NULL);
        size=aio_return(&readrq);
        if (size>0) {
            write(1, buf, size);
            aio_read(&readrq);
        } else if (size==0) {
            break;
        } else if (errno!=EINPROGRESS) {
            perror("reading from socket");
        }
    }
}

```

Асинхронное оповещение о завершении операции

Асинхронное оповещение приложения о завершении операций состоит в генерации сигнала при завершении операции. Чтобы это сделать, необходимо внести соответствующие настройки в поле **aio_sigevent** описателя запроса.

Поле **aio_sigevent** имеет тип **struct sigevent**. Эта структура определена в `<signal.h>` и содержит следующие поля:

- **int sigev_notify** – **режим нотификации**. Допустимые значения – **SIGEV_NONE** (**не посылать подтверждения**), **SIGEV_SIGNAL** (**генерировать сигнал при завершении запроса**) и **SIGEV_THREAD** (**при завершении запроса запускать указанную функцию в отдельной нити**).
- **int sigev_signo** – **номер сигнала, который будет сгенерирован** при использовании **SIGEV_SIGNAL**.
- **union sigval sigev_value** – **параметр, который будет передан обработчику сигнала или функции обработки**. При использовании для асинхронного ввода/вывода это обычно указатель на запрос.
- **void (*sigev_notify_function)(union sigval)** – **функция, которая будет вызвана при использовании SIGEV_THREAD**.
- **pthread_attr_t *sigev_notify_attributes** – **атрибуты нити, в которой будет запущена sigev_notify_function** при использовании **SIGEV_THREAD**.

Далеко не все реализации libaio поддерживают оповещение **SIGEV_THREAD**. Некоторые Unix-системы используют вместо него нестандартное оповещение **SIGEV_CALLBACK**. Далее в этой лекции мы будем обсуждать только оповещение сигналом.

В качестве номера сигнала некоторые приложения используют **SIGIO** или **SIGPOLL** (в Unix SVR4 это один и тот же сигнал). Часто используют также **SIGUSR1** или **SIGUSR2**; это удобно потому, что гарантирует, что аналогичный сигнал не возникнет по другой причине.

В приложениях реального времени используются также номера сигналов в диапазоне от **SIGRTMIN** до **SIGRTMAX**. Некоторые реализации выделяют для этой цели специальный номер сигнала **SIGAIO** или **SIGASYNCIO**.

Разумеется, перед тем, как выполнять асинхронные запросы с оповещением сигналом, следует установить обработчик этого сигнала. Для оповещения необходимо использовать сигналы, обрабатываемые в режиме **SA_SIGINFO**. Установить такой обработчик при помощи системных вызовов **signal(2)** и **sigset(2)** невозможно, необходимо использовать **sigaction(2)**. Установка обработчиков при помощи **sigaction** рассматривается в приложении 2 к этой лекции.

Обработка сигнала в режиме **SA_SIGINFO** имеет два свойства, каждое из которых полезно для наших целей. Во первых, обработчики таких сигналов имеют три параметра, в отличие от единственного параметра у традиционных обработчиков. Первый параметр имеет тип **int** и соответствует номеру сигнала, второй параметр

имеет тип `siginfo_t *`, генерируется системой и содержит ряд интересных полей. Нас в данном случае больше всего интересует поле этой структуры `si_value`. Как раз в этом поле нам передается значение `aio_sigevent.sigev_value`, которое мы создали при настройке структуры `aiocb`.]

Sigaction русский ман и sigaction posix

Приложение 2 sigaction :

Системный вызов `sigaction(2)` рассматривается в разделе «Сигналы». Он предоставляет возможность регистрации обработчиков сигналов с дополнительными параметрами. Для этого необходимо использовать поле `sa_sigaction`. В качестве параметра, `sigaction(2)` получает структуру `sigaction` со следующими полями:

- `void (*sa_handler)(int);` – **Адрес традиционного обработчика сигнала**, `SIG_IGN` или `SIG_DFL`
- `void (*sa_sigaction)(int, siginfo_t *, void *)` – **Адрес обработчика сигнала в режиме SA_SIGINFO**.
- `sigset_t sa_mask` – **Маска сигналов, которые должны быть заблокированы**, когда вызывается функция обработки сигнала.
- `int sa_flags` – **Флаги, управляющие доставкой сигнала.**

Если аргумент `act` ненулевой, он указывает на структуру, определяющую новые действия, которые должны быть предприняты при получении сигнала `sig`. Если аргумент `oact` ненулевой, он указывает на структуру, где сохраняются ранее установленные действия для этого сигнала.

Поле `sa_flags` в структуре `sigaction` **формируется побитовым ИЛИ** следующих значений:

- `SA_ONSTACK` – **Используется для обработки сигналов на альтернативном сигнальном стеке.**
- `SA_RESETHAND` – Во время исполнения функции обработки **сбрасывает реакцию на сигнал к SIG_DFL**; обрабатываемый сигнал при этом не блокируется.
- `SA_NODEFER` – Во время обработки сигнала **сигнал не блокируется**.
- `SA_RESTART` – **Системные вызовы, которые будут прерваны исполнением функции обработки, автоматически перезапускаются.**
- `SA_SIGINFO` – **Указывает на необходимость использовать значение sa_sigaction в качестве функции-обработчика сигнала.** Также используется для доступа к подробной информации о процессе, исполняющем сигнальный обработчик, такой как причина возникновения сигнала и контекст процесса в момент доставки сигнала.

Билет 27

Вопрос 1: Сборка мусора. Основные стратегии сборки мусора, их преимущества и недостатки

Читать Иртегов с.252 глава 4.3 Сборка мусора.

Презентация Иртегова

Иртегов 6 лекция

Явное освобождение динамически выделенной памяти может привести к двум проблемам:

- **Утечка памяти** – ошибка, возникающая, когда программист по каким-то причинам не освобождает выделенную память.
- **Висячие ссылки** – ошибка, возникающая, когда мы уничтожаем объект, на который где-то в другом месте сохранена ссылка или когда мы сохраняем ссылки на объект, который уже уничтожен. (что собственно одно и то же).

Такие ошибки могут привести к потери памяти и разрушению данных соответственно, подробнее можно прочитать в книге.

Чтобы не допустить такие ошибки некоторые системы программирования используют специальный метод освобождения динамической памяти, называемой *сборкой мусора*(*garbage collection*).

Этот метод состоит в том, что ненужные блоки памяти не освобождаются явным образом. Вместо этого используется некоторый более или менее изощрённый алгоритм, следящий за тем, какие блоки ещё нужны, а какие - уже нет.

Это приводит к значительному снижению стоимости разработки: аналогичные программы на java(язык со сборкой) и c/c++(без сборки) разрабатываются в несколько раз быстрее именно на java.

Все методы сборки мусора так или иначе сводятся к поддержанию базы данных о том, какие объекты на кого ссылаются.

Основные стратегии сборки мусора:

(Иртегов 4.3.1 с. 255) Подсчет ссылок

Самый простой метод отличать используемые блоки от ненужных -- считать, что блок, на который есть ссылка, нужен, а блок, на который ни одной ссылки не осталось -- не нужен. Для этого к каждому блоку присоединяют дескриптор, в котором подсчитывают количество ссылок на него. Каждая передача указателя на этот блок приводит к увеличению счетчика ссылок на 1, каждое уничтожение объекта, содержащего указатель -- к уменьшению.

Когда при уничтожении ссылки счетчик оказывается равен нулю, блок памяти удаляется.

Нередко этот способ освобождения памяти называют не сборкой мусора; для него есть специальное название - подсчёт ссылок.

Преимущества

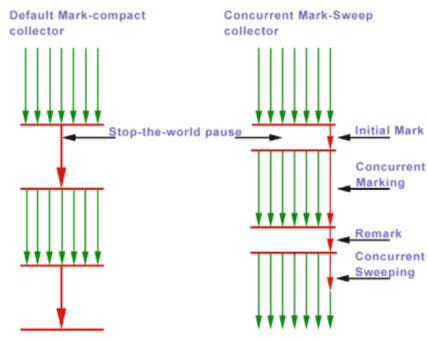
При использовании этого метода удаление объектов и, если это необходимо, вызов деструкторов происходит точно в момент удаления последней ссылки на такой объект, поэтому данный метод используется не только памятью, но и другими дорогостоящими ресурсами.

Тут в книге можно почитать про примеры использования такого метода.(Не уверен, что это нужно, но можно)

Недостатки

Впрочем, при таком подходе возникает специфическая проблема. Если у нас есть циклический список, на который нет ни одной ссылки извне, то все объекты в нем будут считаться используемыми, хотя они и являются мусором. Если мы по тем или иным причинам уверены, что кольца не возникают или возникают очень редко, метод подсчета ссылок вполне приемлем; если же мы используем графы произвольного вида, необходим более умный алгоритм.

«Неблокирующийся» сборщик мусора



(Иртегов 4.3.2 с. 256) Просмотр ссылок (mark and sweep)

Наиболее распространённой альтернативой подсчёту ссылок является периодический просмотр всех ссылок, которые мы считаем "существующими". В англоязычной литературе такой алгоритм называют *mark and sweep*(пометить и стереть).

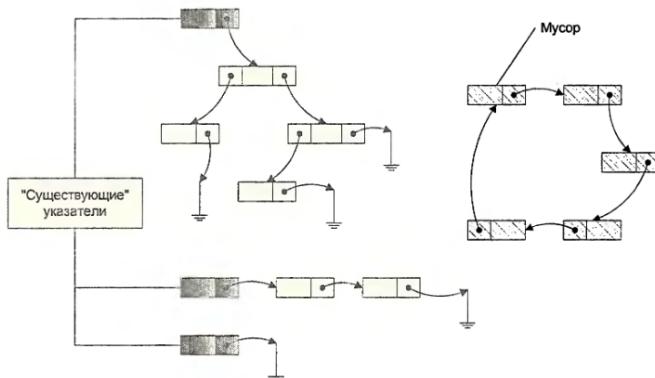


Рис. 4.14. Сборка мусора просмотром ссылок

Обычно просмотр начинается с именованных переменных и параметров функций. Если некоторые из указуемых объектов сами по себе могут содержать ссылки, мы вынуждены осуществлять просмотр рекурсивно. Проведя эту рекурсию до конца, мы можем быть уверены, что то и только то, что мы просмотрели, является нужными данными, и с чистой совестью можем объявить всё остальное мусором.

Недостатки

Много памяти потребляет.

Эта стратегия решает проблему кольцевых списков (преимущество), но требует остановки всей деятельности, которая может сопровождаться созданием или удалением ссылок.(Один из недостатков в том числе).

Сборка мусора просмотром ссылок является очень дорогой операцией, так как требует **остановки всех действий**, связанных с передачей указателей(присваивания, передача параметров процедурам и т.д.). По той же причине эту операцию нельзя запускать отдельной низкоприоритетной нитью. Если сборка делается в многопоточной программе, на время этой операции необходимо остановить все нити.

Реальные системы, использующие сборку мусора, вызывают сборщик через определённые интервалы времени либо при достижении некоторого порога занятой памяти, что произойдёт раньше.

Но все эти методы не позволяют программисту гарантировать, что объект будет уничтожен к определённому моменту времени. В результате потребности таких систем в оперативной памяти очень велики, и на большинстве

задач в несколько раз превосходят требования эквивалентных программ, разработанных на основе явного удаления объектов или подсчёты ссылок.

Кроме того, если объект занимает не только память, но и какие-то дорогостоящие внешние ресурсы - элементы графического пользовательского интерфейса, открытые файлы, сетевые соединения - эти ресурсы также будут освобождаться только по мере проходов сборщика мусора, то есть непонятно когда.

Таким образом, хотя неявное удаление объектов резко снижает стоимость разработки программ, оно столько же резко повышает стоимость их исполнения.

Дальше в книге про бизнес написано.

Важным недостатком сборщиков мусора с просмотром ссылок является тот факт, что на время работы сборщика необходимо останавливать работу системы(**STOP THE WORLD**) или, точнее, всю деятельность, которая может приводить к созданию и уничтожению ссылок на объекты.

Поэтому такая сборка не может осуществляться фоновой нитью. Кроме того, время работы сборщика растёт в линейной зависимости от количества объектов. В больших программах оно может составлять несколько десятков процентов от общего времени исполнения программы.

Тут пример про джаву и с++.

(Иртегов 4.3.3 с. 259) Генерационная сборка мусора.

Генерационная сборка мусора требует перемещения объектов по памяти, поэтому он несовместим с языками, использующими указатели, такими как Паскаль и с/с++.

Для работы такого сборщика система должна реализовать "ручки"(handle)- промежуточные объекты, через которые проходят все обращения к указанному объекту.

В простейшей форме генерационного сборщика мусора объектный пул разбит на две части одинакового размера. Все вновь созданные объекты создаются в одной из частей, называемой "эдемом". Вторая часть пула в это время не используется.

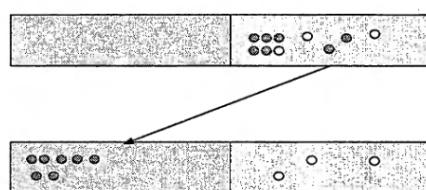


Рис. 4.15. Копирующий сборщик мусора

При заполнении эдема система останавливается и начинается сборка мусора. Сборщик просматривает все объекты и — рекурсивно — ссылки из них на другие объекты, однако вместо простой отметки объекта он копирует каждый из обнаруженных объектов в неиспользуемую часть пула. Когда просмотр завершён, бывший эдем вместе со всем его содержимым объявляется мусором. Неиспользованная половина пула объявляется новым эдемом и работа системы продолжается. Такой сборщик также называется **копирующим**.

Преимущества

Важным преимуществом такого сборщика является тот факт, что сборка мусора оказывается совмещена с дефрагментацией пула. Поэтому можно значительно ускорить работу аллокатора - вместо просмотра списка свободных блоков памяти аллокатор может просто запоминать границу между свободной и занятой частями эдема и выделять память под новые объекты, сдвигая эту границу.

Недостатки

Впрочем, основной недостаток сборщиков мусора - необходимость останавливать систему на время сборку - такой подход не устраниет.

Идея более сложных версий генерационных сборщиков основана на гипотезе, что срок жизни большинства объектов невелик и вероятность уничтожения вновь созданного объекта существенно выше, чем вероятность

уничтожения давно существовавших объектов. Кроме того, с высокой вероятностью вновь создаваемые объекты будут ссылаться на старые объекты и друг на друга, но старые объекты вряд ли будут ссылаться на короткоживущие.

Вместо двух поколений объектов пул разбивается на несколько поколений разного размера с разными ожидаемыми сроками жизни. Сборка мусора состоит в переносе объектов из младших поколений в старшие. Сборка мусора, которая охватывает только "эдем" и младшее поколение, называется "малой" (minor). Заполнение пула старшего поколения инициирует "большую" (major) сборку мусора, которая охватывает весь пул.

При таком подходе серьезную проблему представляют ссылки из объектов старших поколений на объекты младших поколений. Гипотеза, что такие ссылки редки, весьма убедительна и подтверждается практикой, но практика же показывает, что такие ссылки иногда все же возникают.

Чтобы решить эту проблему, система отслеживает все передачи ссылок между объектами и запоминает все ссылки из объектов старшего поколения на младшие в специальном списке, который так и называется — *remembered set* (запомненное множество; общепринятого русскоязычного перевода этого названия нет). Таким образом, при малой сборке мусора необходимо просмотреть ссылки из именованных переменных, *remembered set* и — рекурсивно — из объектов младшего поколения, на которые эти ссылки указывают.

Попадание по ссылке на объект старшего поколения означает прекращение рекурсии.

Этот подход обеспечивает приемлемую производительность только при небольшом объеме *remembered set*. Статистика показывает, что на практике этот объем действительно оказывается небольшим и, при удачном подборе параметров сборщика, иногда вообще нулевым.

Ориентация на среднюю производительность делает генерационную сборку мусора **неприемлемой для задач реального времени**; впрочем, все варианты сборки мусора с просмотром не обеспечивают гарантированного поведения и потому для задач реального времени непригодны.

Дополнительное **преимущество** генерационного сборщика состоит в том, что часто используемые объекты каждого из поколений накапливаются в начале пула своего поколения. Таким образом, все часто используемые объекты собираются в нескольких относительно небольших областях памяти — благодаря этому значительно повышается эффективность работы кэшей центральных процессоров и страничной виртуальной памяти.

Репликационный или инкрементальный сборщик мусора(Иртегов с. 264).

Относительно успешная попытка решить проблему фоновой сборки мусора — это **репликационный** или **инкрементальный** сборщик мусора. В действительности, это вариант **генерационного** сборщика мусора. Основное отличие состоит в том, что при работе (как при просмотре, так и при дефрагментации) репликационный сборщик блокирует не систему в целом, а только нити, которые пытаются модифицировать уже просмотренные объекты. При этом нити, которые только читают значения полей объектов или модифицируют еще не просмотренные объекты, могут продолжать исполнение.

Поскольку объекты — особенно долгоживущие — читаются гораздо чаще, чем модифицируются, это может резко сократить время блокировки и значительно повысить наблюдаемую производительность, хотя, конечно же, полностью исключить блокировку не получается. Наибольший выигрыш при этом достигается при больших сборках, которые затрагивают преимущественно или исключительно долгоживущие объекты и вносят наибольший вклад в наблюдаемое время работы сборщика.

Билет 28

Вопрос 1: Ввод-вывод в режиме опроса и по прерываниям. Преимущества и недостатки.

Иртегов с 350

Допустим, мы хотим, чтобы наша программа обрабатывала внешние события, если мы его уже получили, то всё хорошо, а если нет?

Наивное решение состоит в том, что нам следует циклически опрашивать признак события. Это решение хорошо не только концептуальной простотой, но и тем, что если цикл опроса короток, время реакции будет очень маленьким. Поэтому такой метод нередко используют для обработки последовательностей событий, следующих друг за другом с небольшим интервалом. Однако это решение, называемое *опросом (polling)*, имеет и большой недостаток: загрузив процессор опросом, мы не можем занять его чем бы то ни было другим.

Этот недостаток можно переформулировать иначе: если процессор занят чем-то другим, он может узнать о событии, только завершив текущую деятельность. Если событие действительно важное, впрочем, мы можем расставить команды проверки его признака по всему коду программы, но для сложных программ, обрабатывающих много различных событий, это решение вряд ли можно считать практическим.

С точки зрения встраиваемых приложений, режим опроса имеет еще один существенный недостаток: опрашивающий процессор нельзя выключить, в то же время, выключенный процессор потребляет гораздо меньше энергии и не создает электромагнитных помех, поэтому при разработке программ для таких приложений считается хорошим тоном выключать (переводить в режим ожидания) процессор всегда, когда это возможно. В этом случае, конечно, необходимо предусмотреть какие-либо средства для вывода процессора из этого состояния при возникновении интересующего нас события.

Одно из решений состоит в том, чтобы завести отдельный процессор и поручить ему всю работу по опросу. Процессор, занимающийся только организацией ввода-вывода, называют *периферийным или канальным (channel)*. Периферийные процессоры находят широкое применение в современных вычислительных системах. Так, типичный современный персональный компьютер, кроме центрального процессора, обычно имеет и специализированный видеопроцессор, так называемый графический ускоритель.

Так, при работе с контроллерами дисков, лент и других устройств массовой памяти возникает задача копирования отдельных байтов из контроллера в память и обратно. Передача одного блока состоит из 128 операций передачи слова, идущих друг за другом с небольшими интервалами. Темп передачи данных определяется скоростью вращения диска или движения ленты. Этот темп обычно ниже скорости системной шины, поэтому передача данных должна включать в себя опрос признака готовности контроллера принять или предоставить следующее слово. Интервал между словами обычно измеряется несколькими циклами шины. Нередко бывает и так, что частоты шины и контроллера не кратны, поэтому последовательные слова надо передавать через различное число циклов.

Дополнительная сложность состоит в том, что, не предоставив вовремя следующее слово для записи, мы испортим весь процесс.

Аналогично, не успев прочитать очередное слово, мы потеряем его и вынуждены будем отматывать ленту назад или ждать следующего оборота диска.

Видно, что это именно та ситуация, которую мы ранее описывали как показание к использованию режима опроса: поток следующих друг за другом с небольшим интервалом событий, каждое из которых нельзя потерять, а нужно обязательно обработать.

Дальше тут про контроллер прямого доступа к памяти.

Понятно, впрочем, что использование канальных процессоров и контроллеров ПДП повышает стоимость системы и не решает проблемы радикально — теперь вместо флагов, непосредственно сигнализирующих о внешних событиях, центральный процессор вынужден опрашивать флаги, выставляемые канальным процессором. В зависимости от характера событий и требуемой обработки это решение может оказаться и совсем неприемлемым; например, если на каждое событие требуется немедленная реакция именно центрального процессора.

К счастью, еще с 60-х годов практически все процессоры — как центральные, так и канальные, используют стратегию работы с событиями, во многих отношениях гораздо более совершенную, чем опрос.

Прерывания

Альтернатива опросу, применяемая практически во всех современных процессорах, называется **прерываниями** (interrupt), и состоит в **значительном усложнении логики обработки команд процессором**.

Процессор имеет один или несколько входов, называемых **сигналами** или **линиями запроса прерывания** (IRQ , Interrupt ReQuest). При появлении сигнала на одном из входов процессор дожидается завершения исполнения текущей команды и вместо перехода к исполнению следующей команды инициирует обработку прерывания.

Обработка состоит в сохранении счетчика команд и, возможно, некоторых других регистров и в передаче управления на адрес, определяемый типом прерывания. По этому адресу размещается программа, **обработчик прерывания** (interrupt handler), которая и осуществляет реакцию на соответствующее прерыванию событие.

Перед завершением обработчик восстанавливает регистры, и исполнение основной программы возобновляется с той точки, где она была прервана. В современных процессорах регистры обычно сохраняются в стеке; благодаря этому обработчик прерывания также может оказаться прерван.

Как правило, адреса программ, соответствующих различным прерываниям, собраны в таблицу, называемую таблицей векторов прерываний, размещаемую в определенном месте адресного пространства. У микроконтроллеров каждому возможному сигналу прерывания обычно соответствует свой вектор . Процессоры общего назначения часто используют более сложную схему, в которой устройство, запрашивающее прерывание, передает процессору номер прерывания или сразу адрес обработчика.

Прерывания лишены недостатков, которые мы указали ранее для обработки событий с помощью опроса: ожидая события, процессор может заниматься какой-либо другой полезной работой, а когда событие произойдет, он приступит к обработке, не дожидаясь полного завершения этой работы.

Однако этот механизм имеет и собственные недостатки. В частности, обработка прерывания сопряжена с гораздо большими накладными расходами, чем проверка флага и условный переход в режиме ожидания.

У оптимизированных для обработки событий микроконтроллеров разница невелика или даже может быть в пользу механизма прерываний.

Однако у процессоров общего назначения, которые при обработке прерывания вынуждены сохранять несколько регистров и осуществлять относительно сложный диалог с вызвавшим прерывание устройством, задержка между установкой сигнала прерывания и исполнением первой команды его обработчика — этот интервал и называется **задержкой прерывания** (interrupt latency) — составляет десятки тактов.

Современные суперскалярные процессоры при обработке прерываний вынуждены сбрасывать очередь предварительной выборки команд и по крайней мере часть кэшей команд и данных, поэтому у них накладные расходы еще больше.

Может надо про селект и полл(select poll)

Вопрос 2: Уровни RAID.

Иртегов с 555

Женщины замечательно умеют хранить секреты.
Они делают это сообща.

Еще один прием оптимизации производительности дисковых накопителей — это объединение нескольких физических дисков в один большой логический диск. При таком объединении некоторые диски могут передавать данные, в то время как другие позиционируют блок головок или ждут подхода нужного сектора к головке.

Дисковые массивы выгодны не только с точки зрения производительности, но и повышают единичную емкость запоминающего устройства.

Следует учесть, что объединение дисков приводит к резкому снижению наработки массива на отказ: вероятности независимых событий складываются, поэтому

вероятность отказа любого из дисков массива равна сумме вероятностей отказа одиночного диска. Для компенсации или устранения этого недостатка данные в дисковых массивах обычно хранятся с избыточностью.

Общее название всех технологий объединения дисков — RAID (Redundant Array of Inexpensive Disks — избыточный массив недорогих дисков).

Этот термин был предложен в работе [Gibson/Kat2 уPatterson 1988], в которой проведен анализ различных технологий создания дисковых массивов с точки зрения их производительности и надежности и было рассмотрено пять возможных способов размещения избыточных данных. Предложенная авторами статьи нумерация этих технологий без ссылки на источник используется в самых разнообразных публикациях в форме RAID уровня X.

Под RAID **уровня 0** практически единогласно понимают простой **стриппинг** (stripping — дословно, разделение на полосы). Стриппинг, строго говоря, не является методом создания избыточных массивов, потому что он не предполагает избыточности: емкость результирующего логического диска равна сумме объемов физических дисков (рис. 9.40).

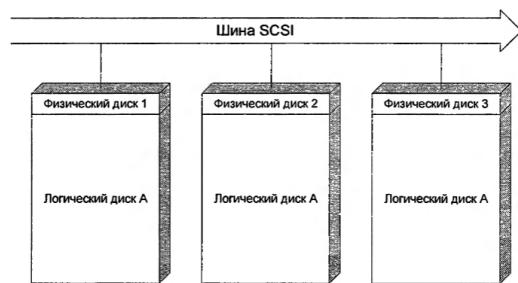


Рис. 9.40. RAID 0 (стриппинг)

RAID **уровня 1** известен также как зеркалирование (mirroring). При зеркалировании на каждый из дисков записывается полная копия данных (рис. 9.41).

Обычно в зеркальном режиме используется не более двух дисков. Зеркалирование обеспечивает некоторое падение производительности (обычно все-таки менее, чем двукратное), но гарантирует работоспособность системы при отказе любого из дисков. Оба эти приема широко применяются на практике, обычно с небольшим количеством дисков, не более двух, реже, трех. Они не требуют значительных вычислительных ресурсов и потому обычно реализуются программно, драйвером диска или файловой системы.

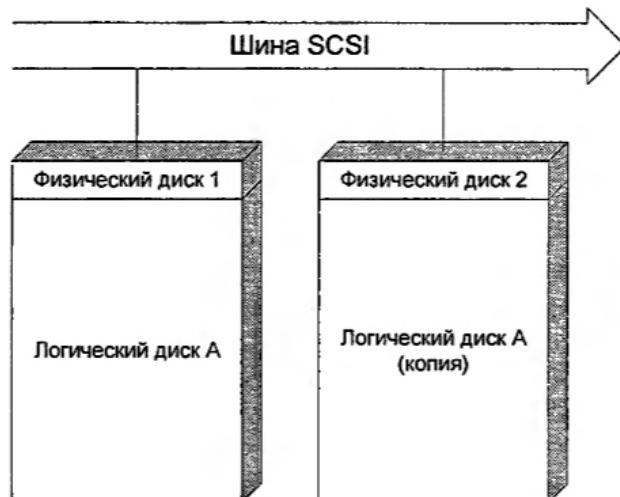


Рис. 9.41. RAID 1 (зеркалирование)

RAID **уровня 2** предлагает снабжение блоков данных на дисках кодом Хэмминга (см. разд. 1.7 с 71) и размещение этого кода на отдельном диске массива.

Эта технология имеет смысл, если контроллеры дисков не хранят код восстановления ошибок в собственных заголовках секторов. Поскольку современные диски практически без исключения используют восстанавливающее кодирование и динамическое переназначение дефектных секторов, RAID 2 ныне практически не применяется.

RAID **уровня 3 и 4** похожи во многих отношениях. Каждый из этих методов требует не менее трех дисков и подсчета контрольной суммы таким способом, чтобы данные могли быть однозначно восстановлены при полной потере одного из дисков. В RAID уровня 3 контрольные суммы подсчитываются **для каждого байта** записываемых данных, а в RAID уровня 4 — **для групп блоков**. Для хранения контрольных сумм отводится отдельный диск.

Эти методики обеспечивают более низкую (хотя и достаточную для большинства практических целей) избыточность, чем зеркаливание, но, как и стриппинг, увеличивают единичную емкость диска и повышают производительность. Для отказа всей системы требуется полный отказ более чем одного диска; многие конструктивы дисковых массивов уровня более 2 предусматривают "**горячую**" (без выключения системы) замену дисков, что допускает теоретически неограниченную наработку на останов, недостижимую при использовании одиночных дисков.

RAID 3 требует изменения формата секторов диска и соответствующей переделки контроллера и не может применяться с серийными дисковыми приводами. **Основным недостатком** RAID 4 является необходимость обращения к диску с контрольной суммой при каждой модификации одного из блоков группы.

При записи большого количества логически последовательных блоков это не проблема, но может превратиться в проблему при случайно распределенных записях одиночных блоков.

Этого недостатка лишен **RAID уровня 5**, в котором контрольные суммы распределены по всем дискам массива (рис. 9.42). RAID 5 находит широкое применение в серверах уровня рабочей группы или подразделения. При программной реализации, впрочем, подсчет контрольных сумм требует значительной доли вычислительной мощности Ц П У, поэтому широкое распространение получили "аппаратные" контроллеры RAID 5, имеющие собственный процессор и, как правило, довольно большой объем собственной памяти. Это не мешает некоторым ОС, в частности WINDOWS NT/2000/XP, реализовать RAID 5 программно на уровне файловой системы.

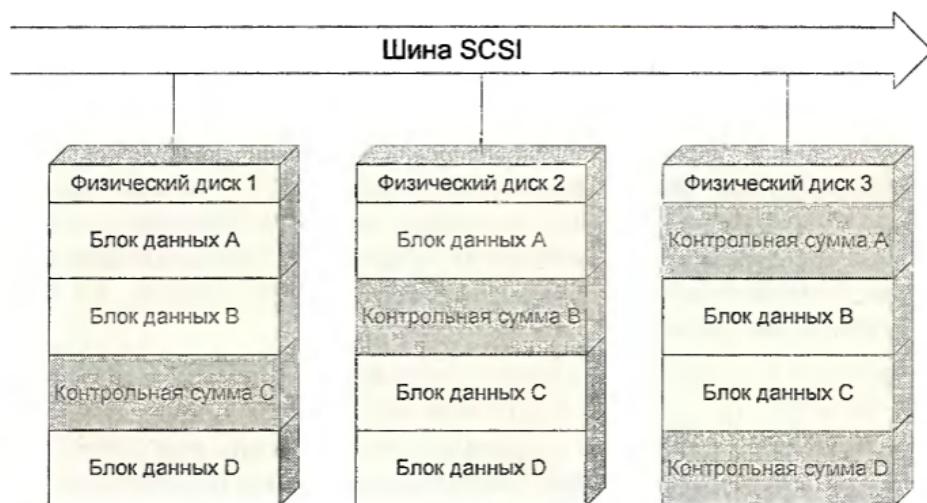


Рис. 9.42. RAID уровня 5

Дальше кстати ИРтегов кончается но есть ещё уровни <https://ru.wikipedia.org/wiki/RAID>

[Лучше про RAID 6 тоже рассказать](#)

[Другая статья](#)

Билет 29

Вопрос 1: Спинлоки и их применение. Их преимущества и недостатки по сравнению с другими средствами взаимоисключения.

Хорошее описание из билетов другого потока

Иртегов с 383

На практике, для решения проблемы работы с флаговыми и другими скалярными переменными в многопроцессорных конфигурациях большинство современных процессоров предоставляют аппаратные **примитивы взаимоисключения**: средства, позволяющие процессору монопольно захватить шину и выполнить несколько операций обращения к памяти.

Реализации этих примитивов различны у разных процессоров. Например, у x86 это команда **xchg** (eXChange, обменять), которая обменивает значения регистра и второго операнда, который обычно размещается в ОЗУ. Эта команда всегда исполняется в режиме **монопольного доступа** кшине, т. е. процессор аппаратно обеспечивает атомарность ее исполнения — в регистре всегда будет находиться именно то значение, которое было в памяти к моменту начала операции, а другие устройства — в том числе и другие процессоры — смогут получить доступ к соответствующей ячейке ОЗУ только тогда, когда операция будет уже завершена.

Самый простой способ использования этой команды для реализации флага взаимоисключения показан в примере 7.4. Видно, что он предполагает ожидание освобождения ресурса в холостом цикле, поэтому в современных ОС такой примитив называют **спинлоком** (**Spinlock**, дословно — вращающийся запор).

Спинлоки обычно используются в ядрах ОС для разделения доступа к ресурсам между нитями ядра, исполняющимися на разных процессорах.

Иртегов с 442

У современных процессоров x86 **сериализация** происходит при исполнении всех команд, имеющих префикс **lock** (префикс монопольного захвата шины), а также при исполнении команды **xchg** и некоторых других команд, которые работают в режиме захвата шины даже без префикса **lock**. Поскольку **спинлоки** в ядре реализуются с помощью именно таких команд, разработчик ОС может убить одним ударом двух зайцев: проверяя спинлок, защищающий семафор и/или очередь активных процессов, планировщик одновременно просит процессор сериализоваться.

Иртегов кончился

Лекция 8:

У нас есть функция CAS (Compare and set) которая проверяет что флаг 1, а когда какая-то нить заходит в КС, она меняет флаг на 1, и так мы понимаем что в КС кто-то есть.

То есть пока в КС кто-то сидит, наша нить ждёт в спинлоке(холостом цикле)

Уходящая нить меняет флаг на ноль и наша нить заходит.

Parallelism.ptx:

Атомарное исключение

- Так или иначе, все процессоры, пригодные для многопроцессорной/многоядерной работы, имеют средства для реализации атомарной функции Compare_And_Set (CAS)
- Взаимное исключение на основе CAS:
`while(CAS(flag, 1)) {}`
- Это называется spinlock
- Спинлоки используются в ядрах ОС
- На спинлоках можно реализовать более сложные атомарные операции

Более сложные примитивы синхронизации

- Спинлок – это холостой цикл, а холостой цикл – это плохо
- Спинлоки пригодны только для защиты коротких критических секций
- Но они позволяют объединять операции в атомарные блоки!
- Давайте объединим в атомарный блок проверку флаговой переменной и засыпание (блокировку нити)?
- Большинство примитивов синхронизации так и сделаны: семафоры, мутексы, блокировки чтения-записи, критические секции Win32, синхронизированные блоки Java, да тыщи их

Проблемы программирования на блокировках

- Мертвые блокировки (deadlock)
- Голодание
- Главная проблема:
границы критических секций может определить только программист
 - Целостность данных – неформализуемое понятие
- Люди ошибаются, а ошибки определения границ секций не ловятся тестированием

Их преимущества и недостатки по сравнению с другими средствами взаимоисключения.

Преимущества

Можно реализовать на всех процессорах.

Простота в реализации.

Позволяют объединять операции в атомарные блоки.

Недостатки

Проблема в том что пока наша нить сидит в спинлоке, процессор греется и ничего другого не выполняет, поэтому решение херня.

Чтобы найти и избавиться от дедлока нужно знать нить которую делала захват, а на спинлоках мы этого не знаем.

Собственно, это проблема касается всех средств взаимоисключения, на которых мы ничего не знаем о нити, которая сделала захват.

Вопрос 2:Структура файловой системы RT-11.