

## 0. ВСТУПЛЕНИЕ

Данный доклад представляет собой реализацию некоей идеи, так сказать - воплощение в жизнь, с целью посмотреть насколько идея работоспособна в принципе. Поэтому и разговор будет в большинстве своем общий, без каких-то особенных частных моментов. Я попробую поделиться историей пути, пути от постановки задачи до ее решения в каком-то виде. Причем история будет от имени человека, совершенно не знакомого до этого с, подобного рода, задачами.

Надеюсь будет интересно.

## I. BBOS

### 1) *Краткая история*

(Здесь — откуда все началось) в первое время этот набор скриптов вручную имплементировался в уже установленную OpenBSD, что существенно увеличивало время приведения системы к требуемому виду. В связи с этим была разработана система GENBOX, позволяющая при помощи нескольких конфигурационных файлов установить BBOS, уже настроенную и готовую к работе. Относительно недавно операционная система обзавелась собственным инсталлятором и, кроме того, существует возможность установить BBOS по сети (необходима поддержка сетевой картой протокола PXE).

Таким образом, из-за значительного количества изменений и ужесточенном подходе к обеспечению безопасности было принято решение об отпочковании от родительского проекта. На данный момент BBOS являет собой форк OpenBSD, с модификацией незначительного количества системных файлов, однако измененная идеология (или подход) к обеспечению безопасности позволяет рассматривать получившийся программный продукт, как самостоятельную операционную систему, все-таки отличающуюся от родителя и решающую конкретные цели и задачи.

Кроме того, ОС прошла сертификацию (здесь о экспертном заключении), и может использоваться при построении комплексных систем защиты информации, что, при наличии большого репозитория совместимого программного обеспечения, выглядит достаточно привлекательно (учитывая полную совместимость с родительской системой).

### 2) *Отличия*

Основные отличия ОС BBOS от OpenBSD.

- Корневая система только в read-only
- Вынесение каталогов /etc, /tmp и при необходимости /var в оперативную память, (Реализован механизм восстановления/резервирования необходимых файлов) что позволяет значительно затруднить изменение критически важных файлов.
- Система проверки целостности критических системных файлов (работает с задаваемой периодичностью).
- Собственный механизм управлением пользователями (обычным образом созданный пользователь не сможет войти в систему). Кроме того, рабочее пространство обычного пользователя вынесено в виртуальное рабочее пространство (специальный зашифрованный файл-эмулятор дискового раздела), которое подключается только по факту входа пользователя в систему и отключается по факту выхода.
- Собственный диспетчер задач для обеспечения устойчивой работоспособности систем контроля целостности и резервирования.
- Возможность быстрого приведения системы в состояние «по-умолчанию»
- Способность стабильно работать на embedded-системах.

### 3) *Необходимость в разруливании конфликтов*

Итак.

В связи с тем, что критические каталоги вынесены в оперативную память, возникла необходимость в создании механизма резервирования легитимно измененных файлов, с возможностью последующего восстановления их для получения желаемых настроек системы. На данный момент реализация этого механизма очень простая: имеется дисковый раздел, куда сбрасываются измененные файлы. Большинство времени раздел находится в отмонтированном состоянии и подключается к системе лишь по факту обнаружения изменившихся файлов (еще раз замечу – по факту легитимно изменившихся файлов). Системой резервирования управляет dump-скрипт, а восстановления restore-скрипт. Все это позволяет обеспечить возможность достаточно прозрачно работать с системой, зажатой в такие драконовские условия. Единственное дополнительное усилие, которое нужно сделать при изменении файлов, скажем в каталоге /etc, это необходимость запуска вручную скрипта резервирования. В некоторых случаях будет необходимым запуск скрипта контроля целостности.

Почему?

Как уже упоминалось выше, параллельно с системой резервирования работает система контроля целостности критически важных файлов. Суть работы ее тоже проста: имеется список файлов, целостность которых периодически проверяется, и если обнаруживается нарушение целостности, система может либо заменить такой файл, либо вообще приостановить работу, в зависимости от важности поврежденного файла. Если системный администратор вносит совершенно законные изменения в файл, подлежащий проверке, тогда простого резервирования будет недостаточно. Необходимо будет перестроить базу контрольных сумм, с тем, чтобы при следующей проверке, система сочла файл легитимным. В этом случае производить дополнительное резервирование не нужно – скрипт системы контроля целостности все сделает сам автоматически.

И вот здесь возникает основная проблема, которую мы попытаемся решить. Хочу сразу заметить, что на данный момент – это скорее модель такого решения, чем законченный продукт, хотя модель и вполне работоспособная.

Итак. Представим себе ситуацию, когда у нас периодически работает скрипт проверки целостности, скажем раз в 15 минут. Кроме того, также периодически, мы сохраняем системные логи, при помощи скрипта резервирования, с какой-то своей периодичностью, скажем раз в 5 минут. И все это как-то с горем пополам работает. Но, что если системный администратор решит внести изменения в файл, находящийся в критическом списке. Да, он сделает изменения, вызовет скрипт контроля целостности для перестройки базы контрольных сумм и автоматического резервирования полученного файла. Но в этом случае никто не может дать гарантии, что вновь измененный файл не будет заменен скриптом контроля целостности, который запустился мгновением раньше через **cron** (система справедливо посчитает его нелегитимным). Кроме того, все скрипты подключают и отключают раздел, куда сбрасываются резервируемые файлы. И скрипт резервирования, который мы запустили для сохранения логов, может успеть отмонтировать раздел, за мгновение до того, как скрипт контроля целостности соберется сохранять туда изменившийся файл. И таких конфликтных ситуаций здесь может быть еще множество. Таким образом, мы подошли к ситуации, когда необходимо некое постороннее вмешательство в работу упомянутых систем, с тем, чтобы четко гарантировать, что в единицу времени с нужными ресурсами работает один и только один объект. Т.е. необходим диспетчер для таких процессов. В нашем случае – для скриптов всех этих систем.

## II. ПРОБЛЕМЫ, КОТОРЫЕ ДОЛЖНЫ БЫТЬ РЕШЕНЫ

### 1) *Межпроцессные взаимодействия*

Самое первое, что нужно сделать — это осуществить передачу информации от одного процесса другому. Здесь выбор небольшой (если отбросить такие экзотические способы как, скажем, взаимодействие посредством e-mail) то, все что нам реально доступно — это файлы, пайпы и сокеты. Передача информации посредством записи в файл отпала сразу — слишком уж дорого это с точки зрения системных ресурсов. Второй вариант — пайпы. Лучше, но все равно дорого, да и громоздко. Сокеты — вот, было бы идеальное решение. Однако, с учетом того, что все будет создаваться на shell'e, то в конце-концов выбор пал на пайпы типа FIFO (о недостатках такого выбора будет сказано чуть позже). Впрочем, для модели пока сгодится.

### 2) *Приоритеты*

После того, как была установлена потенциальная возможность обмена информацией между процессами, возник вопрос — а, собственно говоря, что послужит отправной точкой или, проще сказать, разрешением на запуск того или иного процесса? Что если первый процесс осуществляет резервирование, а второй - проверку целостности. Может имеет смысл осуществить сначала проверку, а потом уже запускать резервирование, будучи уверенным, что в системе все в порядке?

Т.о. возникла необходимость в планировании очередности запуска процессов.

Т.е — необходимы приоритеты.

Здесь тоже нужно понимать, как расставлять эти приоритеты, чтобы, скажем, после совершенно законной правки нужного нам файла, процесс перестройки баз контрольных сумм не оказался где-то в конце списка процессов. А если один процесс (скрипт) запускает еще какой-то свой, дочерний процесс, что тогда? Где окажется уже этот внучатый процесс?

Т.о. после анализа взаимодействия процессов между собой и влияния внешних факторов (работа пользователя) был построен следующий список приоритетов (верхний — самый приоритетный):

- Запуск из другого процесса (здесь дочерний процесс имеет наивысший приоритет. Не важно какой приоритет имел родитель. Дочерний процесс всегда помещается во главу списка.)
- Запуск процесса пользователем (здесь процессы в свою очередь подразделяются на более важные и менее важные)
- Запуск посредством *cron* (в этом случае также существует своя миниерархия процессов)

Таким образом, опираясь на данную классификацию важности задач, можно быть в большой степени уверенным, что более важный процесс пойдет в работу скорее, чем процесс, который может и подождать.

### 3) *Очередь*

Ну, и поскольку мы уже можем выстраивать последовательность запуска процессов, возникает следующая задача: как ее формировать и где хранить?

Т.е. возникает задача организации очереди.

Очередь представляет собой самый обычный массив данных, верхний (или первый) элемент которого определяет какой именно процесс будет запускаться следующим.

Очередь не статична. Т.е. некий процесс, будучи уже первым в списке, может быть подвинут дальше, если на предстартовый момент в системе возник процесс с более высоким приоритетом. Кроме того, очередь может быть вообще перезаписана, если в

работу включается дочерний процесс. Тогда текущее состояние очереди буферизируется и восстанавливается впоследствии по факту отработки дочернего процесса. Причем буфер ведет себя аналогично стеку если, скажем, дочерний процесс запускает, еще один, свой подпроцесс.

## 2) Почему *SHELL*?

Еще раз оговорю, что мы говорим о реализации *модели* такого диспетчера. Модели потому, что на данный момент все, что сделано, реализовано на шелле (ваш покорный слуга на данный момент не владеет C на уровне, позволяющем писать подобные вещи), что естественно не подходит для грамотного решения подобной задачи. Впрочем, первоначально задача стояла именно в том, чтобы разработать алгоритм создания подобного диспетчера.

Кроме того, я не являюсь профессиональным программистом, да и сфера моей деятельности в компании до сих пор не требовала столь специфических знаний. Поэтому говорить буду как человек, однажды оказавшийся перед доселе неизведанной областью, что бы было понятно почему используются те а не иные решения и подходы. Да и вообще — просто интересно, можно ли на языке командной строки решить подобную задачу.

В качестве интерпретатора был выбран KSH, который является штатным командным интерпретатором OpenBSD.

### 1) *Диспетчер*

В общем случае, для решения нашей задачи, диспетчер должен постоянно проверять состояние очереди и запускать процесс, чей идентификатор находится на вершине списка. Однако помимо этого, он осуществляет еще и формирование очереди, ее очистку и буферизацию.

### 2) *Враппер*

В отличие от диспетчера, необходимость в наличии враппера не столь выражена. Однако, если представить, что каждый скрипт (или программа), должны будут иметь процедуру связи с диспетчером — наличие некоей программной прослойки, берущей на себя эту задачу становится очевидной. Кроме того, враппер позволит подключать к диспетчеру любые скрипты, не ограничивая их круг механизмом взаимодействия с диспетчером. Кроме того, враппер производит инспекцию состояния диспетчера и предполагает различные способы запуска процесса, в зависимости от того, работает диспетчер, или, например, завис.

Помимо враппера, существует еще и механизм запуска скриптов (программ), однако он универсален и отличается лишь именем запускаемого объекта. Т.е. - это некий скрипт-запускатель, который запускает, собственно враппер и передает ему имя объекта с параметрами командной строки если таковые имеются.

### 3) *Взаимодействие обслуживающих программ*

Рассмотрим последовательно механизм работы связки диспетчер-враппер.

Итак.

Диспетчер постоянно находится в оперативной памяти и проверяет наличие запросов от враппера. Как только скрипт-запускатель вызывает враппера, тот связывается с диспетчером и передает ему тип вызова (здесь формируются приоритеты) и идентификатор процесса. При очередной проверке, диспетчер обнаруживает запрос и формирует очередь, подготавливая таким образом запуск

требуемого программного объекта. Как только очередь сформирована, диспетчер посылает вращеру разрешение на запуск объекта. Что вращер и делает. После того, как запускаемый объект закончил работу, вращер посылает диспетчеру сообщение об окончании работы и тоже заканчивает свою работу. Диспетчер же, на основании сообщения окончания работы процесса, удаляет его из очереди и переходит к проверке запросов от очередного вращера.

Вот, вкратце, идеология работы механизма устранения межпроцессных конфликтов. Подробнее о работе диспетчера и вращера будет рассказано далее.

### III. НЕМНОГО О ВНУТРЕННОСТЯХ

#### 1) *Работа связки диспетчер + вращер*

Рассмотрим поподробнее работу диспетчера. (Я немного остановлюсь на shell-специфике — может кому-то будет интересно).

Итак.

Диспетчер стартует при запуске операционной системы и находится в оперативной памяти постоянно. Во время запуска, диспетчером, создается именованный FIFO-канал (пайп), при помощи которого будет происходить межпроцессный обмен информацией. Имя этого канала фиксированно и оно же используется вращером впоследствии. Если создание пайпа прошло без проблем, диспетчер выбрасывает в background функцию мониторинга состояния пайпа, которая передает ему информацию по запросу. Это сделано из-за специфики работы с пайпами, о чем будет сказано в разделе о недостатках.

После этого диспетчер входит в состояния мониторинга и ожидает ответ от функции, которая проверяет пайп. Если в канале ничего нет и очередь пуста — диспетчер просто находится в режиме проверки.

Далее в работу вступает вращер.

Сначала запускается стартовый скрипт, который вызывает вращер и передает ему имя программного объекта с параметрами командной строки и идентификатором процесса-родителя. Последнее нужно, чтобы определить откуда был запущен скрипт: **cron**, терминал или из другого скрипта.

После этого, вращер осуществляет проверку (достаточно примитивную) работоспособности диспетчера и если она успешная создает еще один именованный FIFO-канал с именем равным идентификатору вызвавшего его процесса. После этого, вращер определяет тип запуска процесса, определяет степень его важности (в частном случае существует список программных объектов, которым принудительно присваиваются высокие уровни приоритетов, ибо они и только они могут быть запущены с таким уровнем, безотносительно от типа запуска — в основном это нужно для корректной расстановки приоритетов при работе через **cron**), после чего отправляет запрос диспетчеру на разрешение старта. И переходит в режим ожидания. (Все сообщения между вращером и диспетчером после первого запроса происходят по именованному каналу вращера).

Диспетчер, в свою очередь, послав запрос на функцию мониторинга, принимает, наконец, от нее запрос от вращера и передает тело запроса функции, осуществляющей формирование очереди. Функция возвращает диспетчеру уже сформированную очередь, с учетом приоритетов, важности и проч. Диспетчер посылает вращеру разрешение на работу после чего переходит в режим ожидания.

Далее, вращер, получив «добро» от диспетчера просто запускает программный объект и ждет окончания его работы. Как только запущенный объект отработал,

вращер посылает диспетчеру сообщение об окончании работы, удаляет созданный промежуточный пайп и завершает работу.

Диспетчер же, в свою очередь, получив ответ от вращера, удаляет текущий запрос из очереди и переключается в режим мониторинга.

## **2) Вращер и обычные скрипты**

Дальше я хочу немного поговорить о скрипте-запускателе. Зачем он нужен?

Основная задача скрипта — это передача вращеру идентификатора родительского процесса. Вообще, это особенность shell, и будучи написанным на C, подобный «запускатель» возможно был бы и не нужен.

Однако при использовании командной оболочки он играет важную роль, в случае запуска программного объекта из скрипта запущенного не вращером. В таком случае дополнительный пайп не создается, а диспетчеру передается эмулированное сообщение о запуске вложенного скрипта. Вообще это все частности и оно работает.

Однако, получается, что в сложившихся условиях, получившаяся система состоит из трех составляющих: диспетчера, вращера и «запускателя».

## **3) Вложенность скриптов (дочерние подпроцессы)**

Рассмотрим еще один любопытный случай, когда один программный объект вызывается из другого.

В этом случае вращером формируется специальное сообщение, на основании которого диспетчер не удаляет объект из очереди, а буфферизирует всю очередь, после чего очищает ее, поместив в нее новый идентификатор и параллельно записав идентификатор нового процесса в свой внутренний стек дочерних процессов. После этого продолжает работу, как обычно, только в этом случае не опрашивает функцию мониторинга пайпа. Как только от дочернего объекта поступило сообщение о завершении работы, диспетчер просматривает стек дочерних процессов, и если он пуст — возвращает очередь в первоначальное состояние. Если же был осуществлен множественный вложенный вызов, то диспетчер не восстанавливает очередь до тех пор, пока стек не опустеет. В это время он по очереди запускает все процессы, чьи идентификаторы записаны в стеке.

Механизм работы с вращером при этом остается тем же.

Конечно, описанный выше механизм очень громоздок и обладает высокой вероятностью краха (большой частью это вызвано из-за применений пайпов). Чтобы немного снизить риск «забивания» системы мертвыми процессами (если диспетчер завис — вращер не отработает и система переполнится зависшими процессами) вращером осуществляется проверка наличия в системе работающего диспетчера. И если такой обнаружен не будет, программные объекты будут обрабатываться просто по мере их вызова. Это конечно не то, чего хотелось, но таким образом мы уходим от нарушения работоспособности операционной системы.

А вот почему все так печально и как с этим бороться — об этом мы и поговорим в заключение.

## V. БЫЛО БЫ ЛУЧШЕ

### 1) *Создание диспетчера на языке С*

Понятно, что рассмотренная реализация диспетчера процессов не годится как production-решение. Естественно, что все это нужно будет переписать на С, с соответствующими изменениями, которые обеспечивает этот язык. В этом случае мы уйдем из уровня юзерленда и наш скрипт превратится в полноценный демон, что тоже будет только к лучшему.

Однако, как отработку идеи, как тестовую модель — данный диспетчер можно использовать и чему он хорошо служит.

### 2) *Уход от PIPE*

Вторым недостатком данной модели является использование пайпов. Специфика работы командной оболочки с именованными каналами привела к тому, что функцию обработки входящих запросов, пришлось выносить в background, ибо если один процесс, что-то записал в пайп, а никакой другой не читает из него, то процесс, пославший сообщение, останавливает работу до момента, пока кто-то или что-то не заберет информацию из канала. Эта особенность делает всю систему урегулирования межпроцессных интересов в значительной степени подверженной краху, что в свою очередь может негативно сказаться на работе операционной системы в целом. Кроме того, постоянный мониторинг пайпа требует значительного количества системных ресурсов (до 7% - на машине среднего уровня), что не всегда бывает допустимо.

Одним из решений данной проблемы может стать использование сокетов, вместо именованных каналов, что кроме прочего снизит и потребление системных ресурсов.

### 3) *Контроль за состоянием диспетчера*

Еще одним недостатком предложенной реализации является сильная зависимость вращающегося от работоспособности диспетчера. Здесь необходимо организовать надежный механизм проверки его рабочего состояния, и в случае отсутствия последнего, вращающийся должен будет предоставить альтернативный вариант запуска программного объекта. В таком случае будет гарантироваться как запуск процесса, так и ненарушение работы операционной системы в целом.

В идеальном варианте, желательно наличие некоего «монитора», который будет отслеживать критические ситуации и устранять их: например, перезапускать диспетчер или убирать по той или иной причине нерабочие дочерние процессы.