



# Async vs. Multithreaded

checkbox	<input checked="" type="checkbox"/>
due date	
class	OS
Status	approved

## Реально используемые архитектуры многопоточных приложений.


1. Многопроцессные приложения с автономными процессами
2. Многопроцессные приложения, взаимодействующие через трубы, сокеты и очереди System V IPC
3. Многопроцессные приложения, взаимодействующие через разделяемую память
4. Собственно многопоточные приложения
5. Событийно-ориентированные приложения
6. Гибридные архитектуры

### 1. Многопроцессные приложения с автономными процессами

Это самый простой тип многопроцессных приложений.

Для каждой пользовательской сессии или даже для каждого запроса создается свой процесс. Он обрабатывает запрос или несколько запросов и завершается.

Иногда такие процессы «взаимодействуют» при записи сообщений в лог-файл. Для разрешения возникающих при этом коллизий в Unix-системах предусмотрен флаг O\_APPEND при открытии файла. При записи в файлы, открытые с этим флагом, указатель записи всегда перемещается на конец файла. Таким образом, записи в лог, выполняемые разными процессами, никогда не смешиваются. В более современных Unix-системах для ведения логов предоставляется специальный сервис syslog(3C).

 **Примеры:** apache 1.x (сервер HTTP)

#### Преимущества:

1. **Простота разработки.** Фактически, мы запускаем много копий однопоточного приложения и они работают независимо друг от друга. Можно не использовать никаких специфически многопоточных API и средств межпроцессного взаимодействия.
2. Высокая надежность. **Аварийное завершение любого из процессов никак не затрагивает остальные процессы.**
3. Хорошая **переносимость.** Приложение будет работать на любой многозадачной ОС
4. Высокая безопасность. Разные процессы приложения могут запускаться от имени разных пользователей. Таким образом можно реализовать принцип минимальных привилегий, когда каждый из процессов имеет лишь те права, которые необходимы ему для работы. **Даже если в каком-то из процессов будет обнаружена ошибка, допускающая**

#### Недостатки:

1. Далеко не все прикладные задачи можно предоставлять таким образом. Например, эта архитектура годится для сервера, занимающегося раздачей статических HTML-страниц, но совсем **непригодна для сервера баз данных и многих серверов приложений.**
2. **Создание и уничтожение процессов – дорогая операция,** поэтому для многих задач такая архитектура неоптимальна. В Unix-системах предпринимается целый комплекс мер для того, чтобы сделать создание процесса и запуск новой программы в процессе как можно более дешевыми операциями. Однако нужно понимать, что **создание нити в рамках существующего процесса всегда будет дешевле, чем создание нового процесса.**

удаленное исполнение кода, взломщик сможет получить лишь уровень доступа, с которым исполнялся этот процесс.

## 2. Многопроцессные приложения, взаимодействующие через сокеты, трубы и очереди сообщений System V IPC

Перечисленные средства IPC (Interprocess communication) относятся к так называемым средствам **гармонического межпроцессного взаимодействия**. Они позволяют организовать взаимодействие процессов и потоков без использования разделяемой памяти. Теоретики программирования очень любят эту архитектуру, потому что она практически исключает многие варианты ошибок соревнования.

### Преимущества:

1. Относительная **простота разработки**. (По сравнению с многопоточными, где нужно думать о критических секциях и тд.)
2. Высокая надежность. **Аварийное завершение одного из процессов приводит к закрытию трубы или сокета, а в случае очередей сообщений – к тому, что сообщения перестают поступать в очередь или извлекаться из нее**. Остальные процессы приложения легко могут обнаружить эту ошибку и восстановиться после нее, возможно (но не обязательно) просто перезапустив отказавший процесс.
3. Многие такие приложения (особенно основанные на использовании сокетов) легко **переделываются для исполнения в распределенной среде**, когда разные компоненты приложения исполняются на разных машинах.
4. Хорошая **переносимость**. Приложение будет работать на большинстве многозадачных ОС, в том числе на старых Unix-системах.
5. Высокая **безопасность**. Разные процессы приложения могут запускаться от имени разных пользователей. Таким образом можно реализовать принцип минимальных привилегий, когда каждый из процессов имеет лишь те права, которые необходимы ему для работы. **Даже если в каком-то из процессов будет обнаружена ошибка, допускающая удаленное исполнение кода, взломщик сможет получить лишь уровень доступа, с которым исполнялся этот процесс**

### Недостатки:

1. Не для всех прикладных задач такую архитектуру легко разработать и реализовать.
2. Все перечисленные типы средств **IPC предполагают последовательную передачу данных**. Если необходим произвольный доступ к разделяемым данным, такая архитектура неудобна.
3. **Передача данных через трубу, сокет и очередь сообщений требует исполнения системных вызовов и двойного копирования данных – сначала из адресного пространства исходного процесса в адресное пространство ядра, затем из адресного пространства ядра в память целевого процесса**. Это дорогие операции. При передаче больших объемов данных это может превратиться в серьезную проблему.
4. В большинстве систем действуют **ограничения на общее количество труб, сокетов и средств IPC**. Так, в Solaris по умолчанию допускается не более 1024 открытых труб, сокетов и файлов на процесс (это обусловлено ограничениями системного вызова select). Архитектурное ограничение Solaris – 65536 труб, сокетов и файлов на процесс. Ограничение на общее количество сокетов TCP/IP – не более 65536 на сетевой интерфейс (обусловлено форматом заголовков TCP). **Очереди сообщений System V IPC размещаются в адресном пространстве ядра, поэтому действуют жесткие ограничения на количество очередей в системе и на объем и количество одновременно находящихся в очередях сообщений**.
5. Создание и уничтожение процесса, а также переключение между процессами – дорогие операции. Не во всех случаях такая архитектура оптимальна.

## 3. Многопроцессные приложения, взаимодействующие через разделяемую память

В качестве разделяемой памяти может использоваться разделяемая память System V IPC и отображение файлов на память. Для синхронизации доступа можно использовать семафоры System V IPC, мутексы и семафоры POSIX, при отображении файлов на память – захват участков файла.

Фактически, данная архитектура сочетает недостатки многопроцессных и собственно многопоточных приложений. Тем не менее, ряд популярных приложений, разработанных в 80е и начале 90х, до того, как в Unix были стандартизованы многопоточные API, используют эту архитектуру.



**Примеры:** Многие серверы баз данных, как коммерческие (Oracle, DB2, Lotus Domino), так и свободно распространяемые, современные версии Sendmail и некоторые другие почтовые серверы.

#### Преимущества:

1. Эффективный произвольный доступ к разделяемым данным.  
**Такая архитектура пригодна для реализации серверов баз данных.**
2. Высокая **переносимость**. Может быть перенесено на любую операционную систему, поддерживающую или эмулирующую System V IPC.
3. Относительно высокая безопасность. Разные процессы приложения могут запускаться от имени разных пользователей. Таким образом можно реализовать принцип минимальных привилегий, когда каждый из процессов имеет лишь те права, которые необходимы ему для работы. Однако разделение уровней доступа не такое жесткое, как в ранее рассмотренных архитектурах.

#### Недостатки:

1. Относительная **сложность разработки**. Ошибки при синхронизации доступа – так называемые ошибки соревнования – очень сложно обнаруживать при тестировании. Это может привести к **повышению общей стоимости разработки в 3–5 раз по сравнению с однопоточными** или более простыми многозадачными архитектурами.
2. Низкая надежность. **Аварийное завершение любого из процессов приложения может оставить (и часто оставляет) разделяемую память в несогласованном состоянии**. Это часто приводит к аварийному завершению остальных задач приложения. Некоторые приложения, например Lotus Domino, специально убивают все серверные процессы при аварийном завершении любого из них.
3. **Создание и уничтожение процесса и переключение между ними – дорогие операции**. Поэтому данная архитектура оптимальна не для всех приложений.
4. При определенных обстоятельствах, использование разделяемой памяти может приводить к эскалации привилегий. **Если в одном из процессов будет найдена ошибка, приводящая к удаленному исполнению кода, с высокой вероятностью взломщик сможет ее использовать для удаленного исполнения кода в других процессах приложения**. То есть, в худшем случае, взломщик может получить уровень доступа, соответствующий наивысшему из уровней доступа процессов приложения.
5. **Приложения, использующие разделяемую память, должны исполняться на одном физическом компьютере или, во всяком случае, на машинах, имеющих разделяемое ОЗУ**. В действительности, это ограничение можно обойти, например используя отображенные на память разделяемые файлы, но это приводит к значительным накладным расходам

## 4. Собственно многопоточные приложения

**Потоки или нити приложения исполняются в пределах одного процесса. Все адресное пространство процесса разделяется между потоками.** На первый взгляд кажется, что это позволяет организовать взаимодействие между потоками вообще без каких-либо специальных API. В действительности, это не так – если несколько потоков работает с разделяемой структурой данных или системным ресурсом, и хотя бы один из потоков модифицирует эту структуру, то в некоторые моменты времени данные будут несогласованными.

Поэтому потоки должны использовать специальные средства для организации взаимодействия. Наиболее важные средства – это примитивы взаимного исключения (мутексы и блокировки чтения-записи). Используя эти примитивы, программист может добиться того, чтобы ни один поток не обращался к разделяемым ресурсам, пока они находятся в несогласованном состоянии (это и называется взаимным исключением). POSIX thread library предоставляет и другие примитивы (например, условные переменные), позволяющие организовать более сложные, чем взаимное исключение, схемы взаимодействия между нитями.

В целом можно сказать, что многопоточные приложения имеют почти те же преимущества и недостатки, что и многопроцессные приложения, использующие разделяемую память. Однако стоимость исполнения многопоточного приложения ниже, а разработка

такого приложения в некоторых отношениях проще, чем приложения, основанного на разделяемой памяти. Поэтому в последние годы многопоточные приложения становятся все более и более популярны.

Преимущества:

- 1. Высокая производительность. На большинстве Unix-систем, **создание нити требует в десятки раз меньше процессорного времени, чем создание процесса.**
- 2. **Эффективный произвольный доступ к разделяемым данным.** В частности, такая архитектура пригодна для создания серверов баз данных.
- 3. Высокая **переносимость** и легкость переноса ПО из-под других ОС, реализующих многопоточность.

Недостатки:

- 1. **Высокая вероятность опасных ошибок.** Все данные процесса разделяются между всеми нитями. Иными словами, любая структура данных может оказаться разделяемой, даже если разработчик программы не планировал этого (для сравнения, в многопроцессных приложениях, использующих разделяемую память System V IPC, разделяются только те структуры, которые размещены в сегменте разделяемой памяти. Обычные переменные и размещаемые обычным образом динамические структуры данных свои у каждого из процессов). Ошибки при доступе к разделяемым данным – **ошибки соревнования – очень сложно обнаруживать при тестировании.**
- 2. Высокая **стоимость разработки** и отладки, обусловленная п. 1.
- 3. Низкая надежность. **Разрушение структур данных**, например в результате переполнения буфера или ошибок работы с указателями, **затрагивает все нити процесса и обычно приводит к аварийному завершению всего процесса.** Другие фатальные ошибки, например, деление на ноль в одной из нитей, также обычно приводят к аварийной остановке всех нитей процесса.
- 4. Низкая безопасность. **Все нити приложения исполняются в одном процессе, то есть от имени одного и того же пользователя и с одними и теми же правами доступа.** Невозможно реализовать принцип минимума необходимых привилегий, **процесс должен исполняться от имени пользователя, который может исполнять все операции, необходимые всем нитям приложения.**
- 5. **Создание нити – все-таки довольно дорогая операция.** Для каждой нити в обязательном порядке выделяется свой стек, который по умолчанию занимает 1 мегабайт ОЗУ на 32-битных архитектурах и 2 мегабайта на 64-битных архитектурах, и некоторые другие ресурсы. Поэтому данная архитектура оптимальна не для всех приложений.
- 6. Невозможность исполнять приложение на многомашинном вычислительном комплексе. Упомянувшиеся в предыдущем разделе приемы, такие, как отображение на память разделяемых файлов, для многопоточной программы не применимы.

5. Событийно-ориентированные архитектуры

В событийно-ориентированной архитектуре мы **отказываемся от взгляда на исполнение программы как на последовательный процесс.** Вместо этого мы **реализуем программу в виде набора функций или методов-обработчиков, которые вызываются в ответ на возникновение событий.** События могут быть как внешними по отношению к нашей программе (например, прибытие порции данных из сетевого соединения, сработка таймера или нажатие клавиши пользователем), так и внутренними. Внутренние события используются для взаимодействия между разными обработчиками.

Как правило, событийно-ориентированная архитектура предполагает наличие специального модуля, называемого диспетчером или менеджером событий. Этот модуль тем или иным образом собирает информацию о возникших событиях, организует сообщения о событиях и обработчики в очереди в соответствии с теми или иными правилами и вызывает обработчики.



Главная сложность при разработке событийно-ориентированного приложения – это гарантировать, что все обработчики событий будут завершаться достаточно быстро. **Как правило это означает, что обработчики не должны вызывать блокирующихся системных вызовов.** Если удастся соблюсти это требование, то событийно-ориентированная архитектура позволяет получить многие преимущества многозадачности и многопоточности без использования собственно многопоточности. В этом смысле можно даже сказать, что событийно-ориентированная архитектура – это альтернатива многозадачным и многопоточным приложениям.

Событийно-ориентированные архитектуры применяются во множестве различных случаев. Таким образом реализуются программы с графическим пользовательским интерфейсом, сетевые серверы и некоторые клиентские сетевые программы, приложения реального времени, игры и др.



**Примеры:** Ядра большинства современных операционных систем, таких, как Windows, Linux и BSD Unix, также имеют событийно-ориентированную архитектуру.

#### Преимущества:

1. **Высокая производительность.** Грамотно разработанное событийно-ориентированное приложение может одновременно обрабатывать множество событий в рамках одного потока и одного процесса. **Некоторые событийно-ориентированные приложения обрабатывают сотни и тысячи сетевых соединений в одном потоке.**

#### Недостатки:

1. Не для всех приложений эта архитектура подходит. Так, **если обработка события требует длительных вычислений или ее невозможно реализовать без использования блокирующихся системных вызовов, это может затормозить обработку других событий.**
2. Разработка событийно-ориентированного приложения требует высокой квалификации разработчиков. На практике это может привести, и обычно приводит, к высокой стоимости разработки.
3. Код, рассчитанный на другую архитектуру (например, использующий блокирующие системные вызовы), невозможно переиспользовать в событийно-ориентированном приложении. Напротив, многие из приемов кодирования, необходимых в событийно-ориентированных приложениях, бесполезны и даже вредны в других архитектурах.
4. Низкая надежность. **Фатальная ошибка при обработке любого события приводит к аварийному завершению всего процесса.**
5. Низкая безопасность. Поскольку все события обрабатываются одним процессом, то **все обработчики работают от имени одного и того же пользователя.** Невозможно реализовать принцип минимально необходимых привилегий.

## 6. Гибридные архитектуры

Гибридные архитектуры отличаются большим разнообразием. Они позволяют сочетать преимущества, характерные для различных типов простых архитектур. Так, разработчик интерактивного приложения может реализовать пользовательский интерфейс в рамках событийно-ориентированной архитектуры, а для сложных вычислений (например, для переразбиения текста на страницы) запускать фоновые нити.

Однако важно понимать, что очень часто гибридные архитектуры сочетают не только преимущества, но и недостатки используемых базовых архитектур. Поэтому создание приложения с гибридной архитектурой предъявляет высокие требования к квалификации архитектора приложения и большинства разработчиков.

В конце лекции 9 мы рассматривали архитектуру «рабочих нитей» (worker threads), сочетающую многопоточность с событийной ориентацией. Ряд приложений, в том числе Apache 2.0, используют такую архитектуру.

## О вычислительных программах

Есть основных подхода к реализации параллельных вычислительных программ – это параллельные программы с разделяемой памятью и программы, обменивающиеся сообщениями.

Программы с разделяемой памятью удобны, когда нити алгоритма должны часто обмениваться большими объемами данных и/или осуществлять произвольный доступ к разделяемым данным большого объема. Программы, обменивающиеся сообщениями, удобны, когда объем разделяемых данных невелик, а эти данные изменяются редко и в предсказуемых местах.

Для задач, которые нуждаются в редком обмене сообщениями небольшого объема, интересны вычислительные сети типа GRID. Такие сети представляют собой обычные персональные компьютеры, соединенные обычной офисной локальной сетью или даже через Интернет. Эти сети часто допускают произвольное подключение и отключение машин. В этом случае, GRID-система позволяет использовать персональные компьютеры во время простоя. Например, в научно-исследовательском учреждении во время рабочего дня компьютеры используются сотрудниками, а ночью или даже во время обеденного перерыва они подключаются к GRID и занимаются вычислениями. Разумеется, это возможно только если алгоритм допускает разбиение на совершенно автономные подзадачи, не нуждающиеся во взаимодействии друг с другом. Примерами таких задач являются уже упоминавшиеся взлом шифров методом грубой силы и вычисление интегралов методом Монте-Карло, а также поиск внеземных цивилизаций (Seti@Home), поиск генов последовательностей, многие задачи имитационного моделирования.

## Подробнее о событийных архитектурах

Идею, лежащую в основе событийно-ориентированной архитектуры, можно описать следующим образом. программа рассматривается как набор объектов, реагирующих на события. События могут быть как внешними по отношению к системе – например, нажатие пользователем левой кнопки мыши, приход очередной порции данных из сетевого соединения, сработкой датчика в системе реального времени – так и внутренними. Внутренние события объекты-обработчики используют для коммуникации друг с другом.

Желая обрабатывать события определенного типа, программист регистрирует объект-обработчик. **При возникновении события, система вызывает метод объекта, ассоциированный с этим событием. Метод обрабатывает событие (возможно, генерируя при этом события, предназначенные другим объектам) и завершается.** Если методы всех обработчиков выполняют свои операции быстро и без применения блокирующих системных вызовов, можно обеспечить высокую скорость реакции на события даже в рамках однопоточной программы. Если события поступают быстрее, чем обработчики успевают их обрабатывать, система выстраивает их в очередь.

В типичной системе с графическим пользовательским интерфейсом обработка событий происходит в несколько этапов. Рассмотрим простой пример окна с кнопкой «ОК». Окна во всех системах графического интерфейса представляют собой обработчики событий. При добавлении кнопки в окно регистрируется еще один обработчик событий, связанный с кнопкой. Все, что делает этот обработчик при нажатии мышью на кнопку – генерирует командное событие, предназначенное окну. Далее, если наше окно представляет собой не главное окно приложения, а модальный диалог (например, диалог открытия файла), обработка командного события от кнопки «ОК» может свестись к отправке командного события главному окну – в случае диалога открытия файла это командное событие должно содержать команду «открыть файл» с указанным именем файла.

Основные идеи, лежащие в основе современных графических интерфейсов, были разработаны во второй половине 1970х в исследовательской лаборатории Xerox PARC.

Однако сама по себе событийно ориентированная архитектура была придумана еще в 60е годы XX столетия и использовалась в ядрах операционных систем, главным образом при реализации подсистемы ввода-вывода. **Драйвер устройства для типичной многозадачной ОС представляет собой набор функций («точек входа») и блок переменных состояния устройства. Указатель на этот блок переменных состояния передается каждой из функций драйвера в качестве параметра. Благодаря этому, драйвер может обслуживать несколько однотипных устройств – ему для этого достаточно создать и зарегистрировать несколько блоков переменных состояния.**

Фактически, драйвер представляет собой нечто очень похожее на объект в объектно-ориентированном программировании, однако в большинстве ОС драйверы до сих пор разрабатываются на не-объектно-ориентированных языках, чаще всего на С или даже на ассемблере. Наиболее известное исключение представляет ОС Apple Darwin (ядро MacOS X), в которой драйверы разрабатываются на специализированном подмножестве C++.

Драйвер обслуживает несколько потоков событий. **Типичный драйвер физического устройства обслуживает два потока событий – запросы от пользовательских программ и прерывания от устройства.** В простейшем случае обслуживание обоих типов событий состоит в том, что у драйвера вызываются соответствующие функции: при **формировании запроса на запись пользовательская программа (при посредстве диспетчера системных вызовов) зовет функцию write** драйвера, а **при обработке прерывания ядро системы (точнее, диспетчер прерывания) вызывает функцию interrupt того же самого драйвера.** Защита от одновременного вызова этих функций возлагается на драйвер и часто состоит в том, что драйвер запрещает прерывания на время работы критических

секций своей функции write. Такая архитектура используется драйверами символьных устройств в старых (так называемых «монолитных») ядрах Unix-систем.

При работе с блочными устройствами, а в более современных Unix-системах также с символьными устройствами **STREAMS**, используется более совершенная архитектура, когда запросы пользовательских программ ставятся в **очередь к драйверу**. Каждый запрос снабжается кодом запроса (чтение, запись, ioctl). В некоторых ОС, например в **DEC RSX-11** и **VMS**, такая архитектура была единственно допустимой архитектурой для драйверов.

Многие драйверы вынуждены иметь дело с множественными потоками событий. Так, например, **драйвер сетевого протокола IP** в машине с несколькими сетевыми интерфейсами должен обслуживать **потоки событий от драйверов всех этих интерфейсов**, **потоки событий от драйверов протоколов транспортного уровня**, а также, возможно, **потоки событий от приложений, работающих с сетью напрямую**, через сокет типа RAW.

Драйвер, допускающий одновременную работу с устройством нескольких пользовательских процессов, должен обеспечивать обслуживание нескольких потоков пользовательских запросов.

С формальной точки зрения обработчик событий удобнее всего описывать в виде конечного автомата. Конечный автомат описывается в виде набора допустимых состояний и набора допустимых переходов между ними, т.е. представляет собой ориентированный граф. Этот граф может описываться как матрицей инцидентности (таблицей состояний), так и картинкой, состоящей из коробочек (состояний) и стрелочек (переходов), например, диаграммой состояний UML. В некоторых случаях применяют также диаграмму, в которой переходы («активности») изображаются в виде коробочек, а состояния – в виде стрелочек, например диаграмма активностей UML (см. рис. 1). В действительности эти два типа диаграмм – два разных способа описания одного и того же.

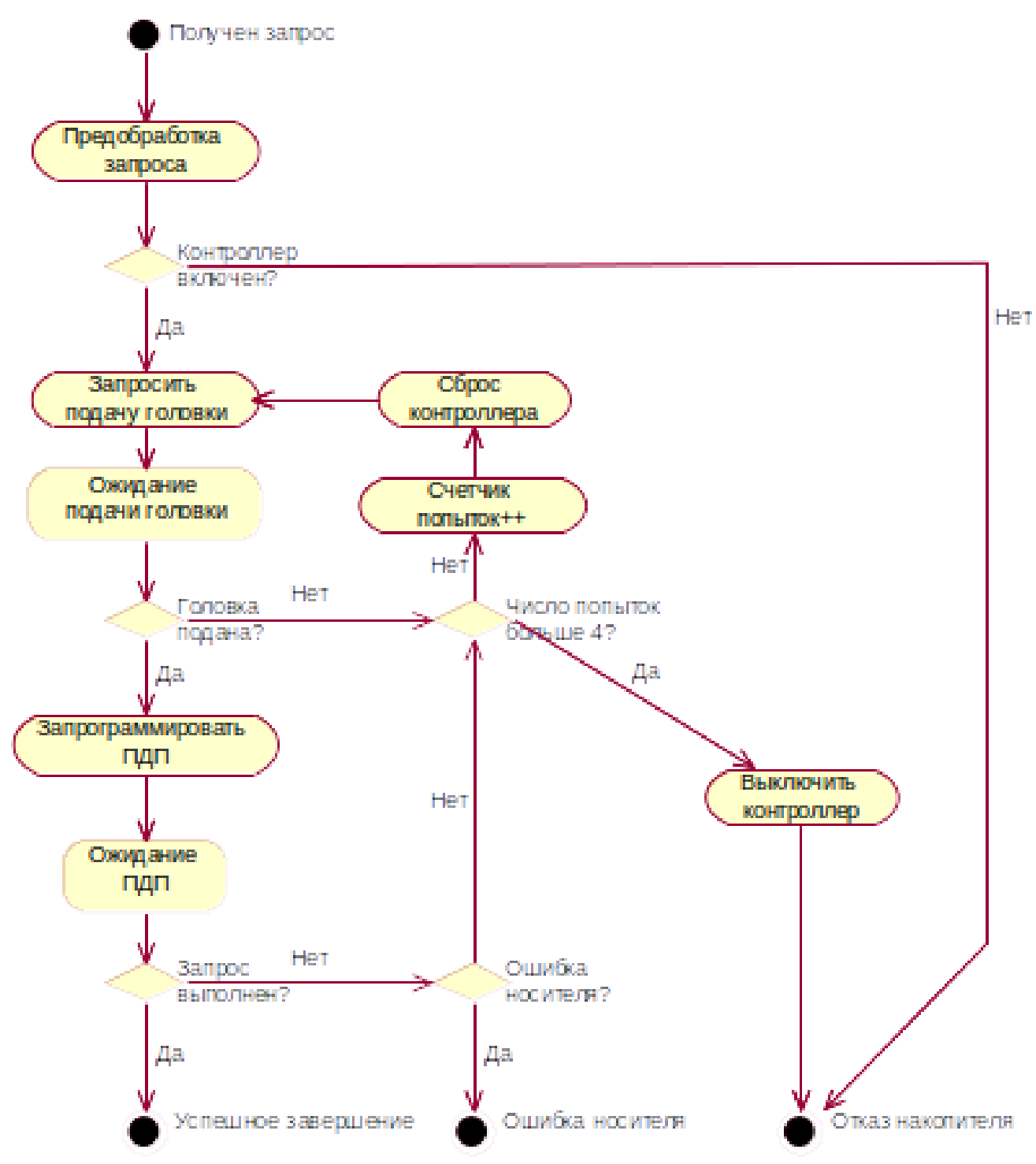


Рис 1. Диаграмма активностей для обработки одного запроса драйвером привода для гибких магнитных дисков (для упрощения диаграммы опущено включение мотора и ожидание разгона диска до рабочей скорости).

При реализации конечного автомата, состояние кодируется одной из переменных в блоке переменных состояния автомата, а переходы инициируются источниками событий. Если источник событий пытается инициировать недопустимый переход (послать событие, которое обработчик сейчас не может или не должен обрабатывать), возможны разные стратегии поведения. Соответствующее событие может просто игнорироваться (на графе состояний это изображают кольцевой стрелкой, выходящей из состояния и возвращающейся в то же самое состояние), порождать встречное сообщение об ошибке или ставиться в очередь. В некоторых случаях обработчик может блокировать нежелательные для него источники событий – например, в графическом интерфейсе контроллер диалогового окна может заблокировать кнопку ОК, пока пользователь не введет в поля диалога все необходимые данные, а драйвер может заблокировать прерывания от устройства, установив соответствующую комбинацию бит в регистрах устройства.

## Finally, Сравнение обработки событий и многопоточных приложений

В многопоточном приложении для каждой нити (потока) создается иллюзия последовательного исполнения. При переключении управления на другую нить система сохраняет контекст нити (в первую очередь стек и счетчик команд, но также и другие регистры процессора), а при возврате управления нашей нити – восстанавливает все эти регистры. Нить может хранить свое состояние в различных областях памяти – в регистрах, в стеке, в thread-specific data. Если нити необходимо обработать внешнее (по отношению к ней) событие, она блокируется в ожидании этого события, исполняя блокирующийся системный вызов или библиотечную функцию. После выхода из блокировки ее исполнение продолжается с той же точки, в которой было остановлено.

**В событийно-ориентированной архитектуре, для методов обработчика событий не создается иллюзии последовательного исполнения.** Если методу обработчика надо перейти к обработке следующего события, он просто завершается. Когда следующее событие возникнет, система его позовет. **Таким образом, обработчик событий не имеет собственного контекста.** Он должен хранить все свое состояние в блоке переменных состояния.

Таким образом, главное отличие между традиционными многопоточными и событийно-ориентированными приложениями можно описать так: многопоточное приложение блокируется на примитивах взаимодействия с источниками событий с сохранением контекста нити. Событийно-ориентированное приложение блокируется на примитивах взаимодействия с источниками событий с уничтожением контекста нити. На практике, разумеется, контекст нити обычно не уничтожается, а переиспользуется для обработки других событий другими обработчиками.

Как правило, переиспользование контекста достигается за счет того, что нить, которая вызывает методы обработчиков событий, исполняет цикл менеджера или диспетчера событий. Менеджер событий опрашивает источники событий, при отсутствии событий – блокируется, а при появлении событий – диспетчеризует события обработчикам.

Из такого описания очевидно, что **событийно-ориентированная архитектура гораздо дешевле при исполнении, чем классическая многопоточная при обработке того же потока событий. Ей требуется гораздо меньше ресурсов – памяти для хранения контекстов и процессорного времени для сохранения и восстановления этих контекстов – чем классической многопоточной архитектуре.**

Из такого описания понятны также возможные подходы к построению гибридной архитектуры. Действительно, **при всех своих достоинствах, событийно-ориентированная архитектура предъявляет довольно жесткие требования к обработчикам событий – они должны завершаться за небольшое время и нигде не блокироваться.** Далеко не для всех задач легко выполнить эти требования. Обработка некоторых событий может требовать длительных вычислений. Другие события могут иметь такую природу, что их нельзя включить в цикл опроса диспетчера событий. Например, если диспетчер событий опрашивает источники событий при помощи select/poll, мы не можем включить в этот цикл события, связанные с примитивами синхронизации POSIX Threads API. В третьем случае **мы можем быть вынуждены в рамках обработки события запускать код, написанный другими программистами для других целей; этот код может ничего не знать про событийно-ориентированную архитектуру и про то, что ему нельзя блокироваться.**

В этих случаях может оказаться целесообразной архитектура с несколькими менеджерами событий, исполняющимися в нескольких разных потоках.

Существует несколько вариантов такой архитектуры – асинхронная очередь сообщений, **preforked processes**, **thread pool** (пул нитей), рабочие нити (**worker threads**). В действительности, классификация вариантов этой архитектуры не является общепринятой и поэтому разные поставщики программного обеспечения часто называют одно и то же разными названиями, а разные архитектуры – одинаковыми терминами.

**Так, например, популярный HTTP-сервер Apache 1.x имел следующую архитектуру:** При запуске, основной процесс Apache привязывал сокет к порту 80 и, если это необходимо, создавал сокеты и привязывал их к другим обслуживаемым портам. Затем этот процесс несколько раз исполнял fork(2), создавая свои копии. Все эти копии наследовали слушающие сокеты, и блокировались в вызове accept(3SOCKET) (в действительности, на select(3C) с ожиданием всех слушающих сокетов). При приходе запроса на



соединение, у одного из процессов ассерт разблокировался и этот процесс начинал обработку этого запроса. **Разумеется, обработка запроса включает в себя многочисленные блокировки на вызовах read(2) и write(2);** кроме того, обработка может включать в себя загрузку модулей и cgi-скриптов, которые исполняют практически произвольный код и, вообще говоря, **могут блокироваться на чем угодно, чаще всего – на обращениях к базе данных.** Если в это время приходит еще один запрос, его должен подхватить другой процесс.

Когда процесс Apache завершает обработку запроса, он не завершается, а возвращается к вызову ассерт и блокируется в ожидании следующего соединения. Таким образом, сервер не тратит время и ресурсы на запуск процесса при поступлении каждого нового запроса на соединение, но, тем не менее, каждое соединение все равно обслуживается отдельным процессом. Именно эта архитектура и называется `preforked processes`, или, если вместо процессов мы используем нити, пулом нитей (thread pool). Видно, что это частный случай событийно-ориентированной архитектуры, при которой обрабатываемым событием считается приход нового запроса на соединение.

Количество процессов, запускаемых Apache, подбирается эмпирически исходя из производительности сервера, потока запросов, скорости каналов, по которым приходят эти запросы, и среднего времени исполнения скриптов на страницах веб-сайта.

Несмотря на простоту этой архитектуры, она обладает очевидным недостатком. Если все процессы сервера заблокированы на том или ином системном вызове, это означает, что сервер имеет ресурсы (во всяком случае, процессорное время) для обработки еще одного соединения – но для этого ему надо запускать еще один процесс!

Чтобы обойти эту проблему, нам следует считать отдельным событием не приход нового запроса на соединение, а готовность каждого из соединений к приему и передаче данных. На предыдущей лекции мы изучали средства, которые специально предназначены для диспетчеризации таких событий – `select(3C)`, `poll(2)`, порты Solaris. Можно также считать событием не готовность соединений к приему и передаче данных, а завершение предыдущего запроса или группы запросов к этому соединению. В таком случае необходимо использовать асинхронный ввод/вывод, а для диспетчеризации событий можно использовать `aio_suspend(3AIO)`, `aio_wait(3AIO)`, `sigwait(2)` (в сочетании с использованием сигналов для оповещения о завершении запроса) или опять-таки порты Solaris.

Обсуждаемую архитектуру невозможно реализовать при использовании `preforked` процессов. Действительно, чтобы один процесс мог обслуживать несколько соединений, он либо должен сделать ассерт(3SOCKET) на все эти соединения, либо должен быть запущен уже после того, как родитель делает этот ассерт. Оба варианта противоречат самой идее `preforked processes`. Поэтому наиболее естественным вариантом для реализации предлагаемой архитектуры является многопоточное приложение.