В начале 70-х годов появилась новая архитектура многозадачных систем, довольно резко отличающаяся от вышеописанной модели последовательных процессов. Речь идет о так называемых системах, управляемых событиями (event-driven systems).

Впервые эта архитектура была реализована в экспериментальных настольных компьютерах Alto, разработанных в 1973 году в исследовательском центре PARC фирмы Xerox. Целью эксперимента было создание операционной среды, удобной для создания интерактивных программ.

В этих системах впервые была реализована многооконная графика, когда пользователь одновременно видит на экране графический вывод нескольких программ и может активизировать любую из них, указав на соответствующее окно при помощи манипулятора "мышь".

При каждом движении мыши, нажатии на ее кнопки или клавиши на клавиатуре генерируется событие. События могут также генерироваться системным таймером или пользовательскими программами. Существуют также так называемые "визуальные" события: например, сдвинув или закрыв одно из окон, мы открыли часть окна, находившегося внизу. Этому окну посылается событие, говорящее о том, что ему нужно перерисовать часть себя.

Каждое событие представляет собой структуру данных, которая содержит код, обозначающий тип события: движение мыши, нажатие кнопки и т.д., а также поля, различные для различных типов событий: для "мышиных" событий это текущие координаты мыши и битовая маска, обозначающая состояние кнопок (нажата/отпущена). Для клавиатурных событий это код нажатой клавиши - обычно, ASCII для алфавитно-цифровой клавиатуры и специальные коды для стрелок и других "расширенных" клавиш - и битовая маска, обозначающая состояние различных модификаторов, таких как SHIFT, CNTRL, ALT и т.д.; для визуальных событий это координаты прямоугольника, который нужно перерисовать, и так далее.

Все события помещаются в очередь в порядке их возникновения.

В системе существует понятие обработчика событий. Обработчик событий представляет собой объект, то есть структуру данных, с которой связано несколько программных модулей - методов. Один из методов вызывается при поступлении события и называется callback ( дословно - "вызов назад").

Представим себе простой объект, такой как меню. При нажатии на кнопку мыши в области этого меню вызывается callback. Он разбирается, какой из пунктов меню был выбран, и вызывает соответствующую функцию обработки этого пункта. Таким образом, вместо последовательно исполняющейся программы, время от времени вызывающей систему для исполнения той или иной функции, мы получаем набор callback'ов, вызываемых системой в соответствии с желаниями пользователя. Отсюда, видимо, и происходит термин "callback" - это "системный вызов, идущий в обратном направлении".

Специальная программа, менеджер событий, непрерывно просматривает очередь и передает поступающие события обработчикам. События, связанные с экранными координатами, передаются обработчику, ассоциированному с соответствующим окном. Клавиатурные события передаются фокусу клавиатуры - текущему активному обработчику.

Легко понять, что менеджер и обработчики событий достаточно органично вписываются в традиционную многозадачную ОС. События становятся дополнительным средством синхронизации и передачи данных, удобным для организации пользовательского интерфейса. В других ситуациях программист может пользоваться другими средствами. Так устроены Windows NT и оконная подсистема OS/2 - Presentation Manager. Особенно любопытен в этом отношении опыт сетевой оконной системы X Windows, где передача событий реализована на основе стандартных средств межпроцессного взаимодействия: труб или разделяемой памяти при работе в пределах одной машины и сокетов при взаимодействии с удаленной машиной.

Однако наиболее распространенные в настоящее время событийные системы: MS Windows 3.x и MacOS реализуют иной подход.

Эти системы, занимающие промежуточное положение между ОС и ДОС, реализуют кооперативную многозадачность. В большинстве же случаев даже кооперативная многопроцессность не используется: менеджер событий синхронно вызывает callback'и, и все программы работают как единый процесс. Предполагается, что обычно callback исполняется быстро, а возможность переключать фокус между различными задачами дает пользователю многие из преимуществ многопроцессной системы. Если долго исполняющиеся callback'и регулярно вызывают функцию GetNextEvent, то можно создать довольно приличную иллюзию полноценной ОС.

Одним из основных недостатков такой архитектуры является то, что система оказывается зависимой от качества реализации приложений: чем реже приложение вызывает GetNextEvent, тем менее работа системы похожа на многопроцессную. Другие недостатки кооперативной многопроцессности обсуждались в разделе 6.1. Кроме того, кооперативное переключение процессов существенно усложняет реализацию сетевых приложений и multimedia: чувствительные к времени реакции программы не могут исполняться как обычные задачи, они вынуждены отбирать управление у системы.

В Windows 3.x для этой цели предоставляется возможность регистрировать так называемые VxD - модули, исполняющиеся с привилегиями ядра системы. Такие модули могут непосредственно обрабатывать внешние события и реагировать на них, не отдавая управления ядру. В Windows 3.x и Windows 95 VxD являются единственным средством, пригодным для реализации приложений, нуждающихся в жесткой синхронизации, например, для синхронизации звука и изображения в multimedia.

Самым интересным во всем этом является тот печальный факт, что программы, использующие VxD, не могут исполняться в ОС Windows NT, которую MicroSoft рекламирует в качестве "системы будущего".

**Событием** называется оповещение о некотором выполненном действии. В программировании события используются для оповещения одного потока о том, что другой поток выполнил некоторое действие. Сама же и задача оповещения одного потока о некотором действии, которое совершил другой поток называется задачейусловной синхронизации или иногда задачей оповещения.

В операционных системах Windows события описываются объектами ядра Events. При этом различают два типа событий:

− события с ручным сбросом;

− события с автоматическим сбросом.

Различие между этими типами событий заключается в том, что событие с ручным сбросом можно перевести в несигнальное состояние только посредством вызова функции ResetEvent, а событие с автоматическим сбросом переходит в несигнальное состояние как при помощи функции ResetEvent, так и при помощи функции ожидания. При этом отметим, что если события с автоматическим сбросом ждут несколько потоков, используя функцию WaitForSingleObject, то из состояния ожидания освобождается только один из этих потоков.

Создаются события вызовом функции CreateEvent, которая имеет следующий прототип:

HANDLE CreateEvent(

LPSECURITY\_ATTRIBUTES lpSecurityAttributes, // атрибуты защиты

BOOL bManualReset, // тип события

BOOL bInitialState, // начальное состояние события

LPCTSTR lpName // имя события

);

Как и обычно, пока значение параметра lpSecurityAttributes будем устанавливать в NULL. Основную смысловую нагрузку в этой функции несут второй и третий параметры. Если значение параметра bManualReset равно TRUE, то создается событие с ручным сбросом, в противном случае – с автоматическим сбросом. Если значение параметра bInitialState равно TRUE, то начальное состояние события является сигнальным, в противном случае – несигнальным. Парметр lpName задает имя события, которое позволяет обращаться к нему из потоков, выполняющихся в разных процессах. Этот параметр может быть равен NULL, тогда создается безымянное событие. В случае удачного завершения функция CreateEvent возвращает дескриптор события, а в случае неудачи – значение NULL. Если событие с заданным именем уже существует, то функция CreateEvent возвращает дескриптор этого события, а функция GetLastError, вызванная после функции CreateEvent вернет значение ERROR\_ALREADY\_EXISTS.