Операционные системы

Межпроцессное взаимодействие

Виды межпроцессного взаимодействия (IPC)

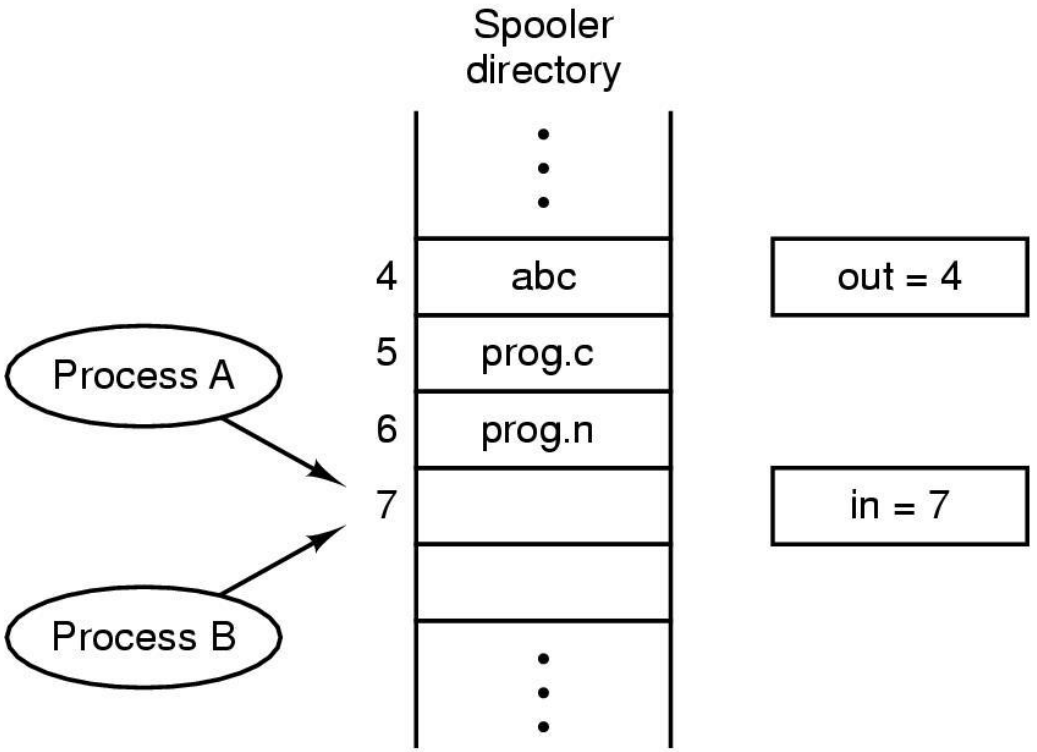
- Предотвращение критических ситуаций
- Синхронизация процессов
- Передача информации от одного процесса другому

Межпроцессное взаимодействие

Предотвращение критических ситуаций и средства синхронизации процессов

Возникновение гонок (состязаний)

- Два процесса хотят получить доступ к общей памяти в одно и тоже время.



Возникновение гонок (состязаний)

- Если процессу требуется вывести на печать файл, он помещает имя файла в специальный каталог спулера.
- Другой процесс, “демон печати”, периодически проверяет наличие файлов, которые нужно печатать, печатает файл и удаляет его имя из каталога.
- Представим, что каталог спулера состоит из большого числа сегментов, нумерованных 0,1, 2…, в каждом из которых может храниться имя файла. Также есть две совместно используемые переменные: out, указывающая на следующий файл для печати, и in, указывающая на следующий свободный сегмент.
- Эти две переменные можно хранить в одном файле, доступном для всех процессов. Пусть в данный момент времени сегменты с 0 по 3 пусты (файлы уже напечатаны), а сегменты 4-6 заняты (файлы ждут печати).
- Более или менее одновременно процессы A и B решают поставить файл в очередь на печать.

Возникновение гонок (состязаний)

- Возможна следующая ситуация:
  - Процесс А считывает значение (7) переменной in и сохраняет его в локальной переменной next_free_slot. После этого происходит прерывание по таймеру, и процессор переключается на процесс В. Процесс В, в свою очередь, считывает значение переменной in и сохраняет его (опять 7) в своей локальной переменной next_free_slot.
  - Теперь оба процесса считают, что следующий свободный сегмент – седьмой.
  - Процесс В сохраняет в каталоге спулера имя файла и заменяет значение in на 8, затем продолжает заниматься своими задачами, не связанными с печатью.
  - Наконец управление переходит к процессу А, и он начинает с того места, на котором остановился. Он обращается к переменной next_free_slot, считывает ее значение и записывает в седьмой сегмент имя файла (удаляя значение, записанное процессом В).
  - В результате файл процесса В не напечатается.

Критические секции

- Важным понятием синхронизации потоков для решения проблемы состязаний является понятие «критической секции» программы.
- Критическая секция – это часть программы, результат выполнения которой может непредсказуемо меняться, если переменные, относящиеся к этой части программы, изменяются другими потоками в то время, когда выполнение этой части еще не завершено.

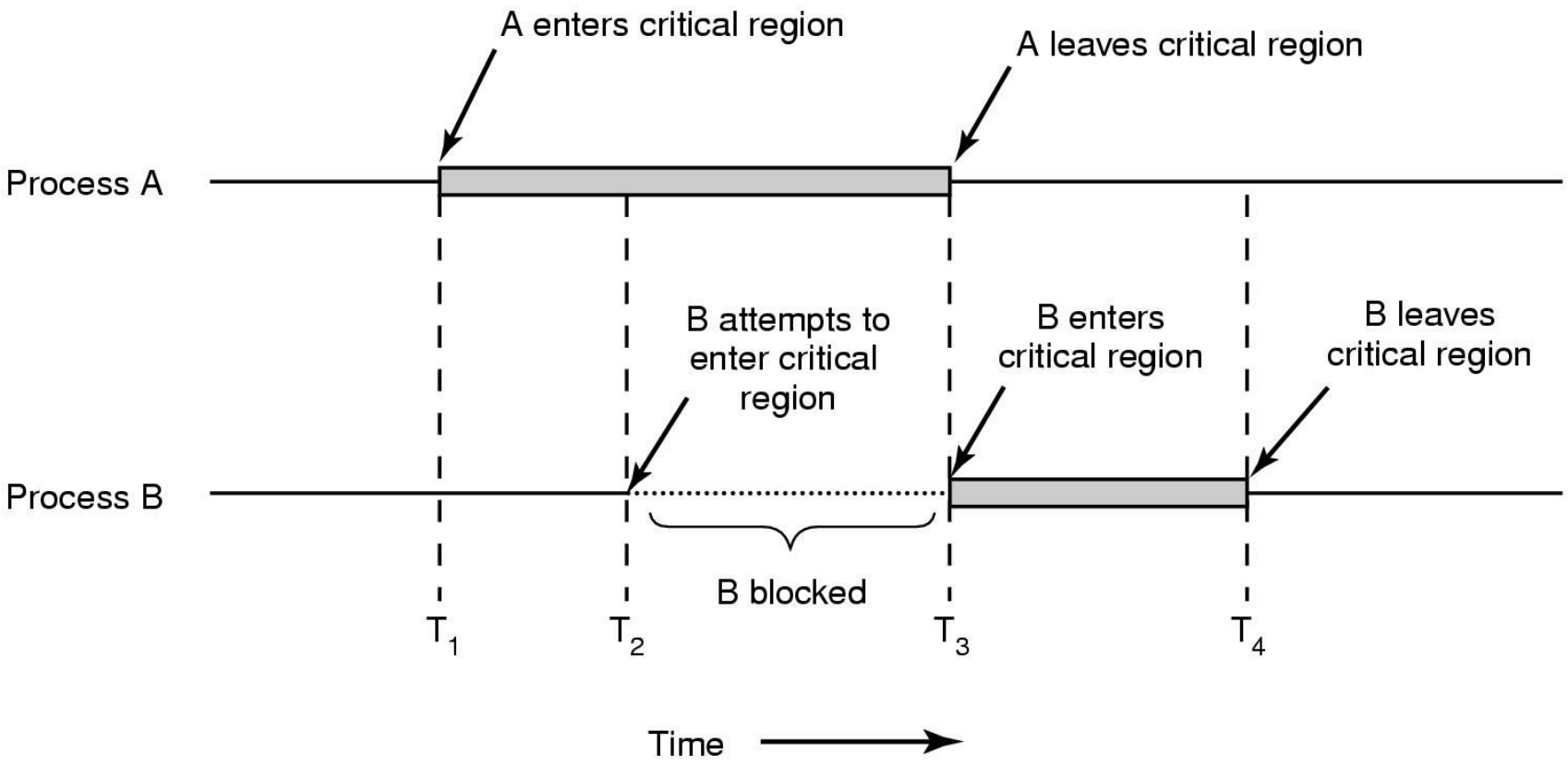
Критические секции

- Во всех потоках, работающих с критическими данными, должна быть определена критическая секция.
- В разных потоках критическая секция состоит в общем случае из разных последовательностей команд.
- Самый простой и в то же время самый неэффективный способ обеспечения взаимного исключения состоит в том, что ОС позволяет потоку запрещать любые прерывания на время его нахождения в критической секции. Однако этот способ практически не применяется, так как опасно доверять управление системой пользовательскому потоку – он может надолго занять процессор, а при крахе потока в критической секции крах потерпит вся система, потому что прерывания никогда не будут разрешены.

Условия исключения гонок

- Два процесса не должны одновременно находиться в критической секции
- В программе не должно быть предположений о скорости или количестве процессоров
- Процесс вне критической секции не может блокировать другие процессы
- Должна быть невозможна ситуация, когда процесс вечно ждет попадания в критическую секцию

Взаимное исключение с использованием критических секций

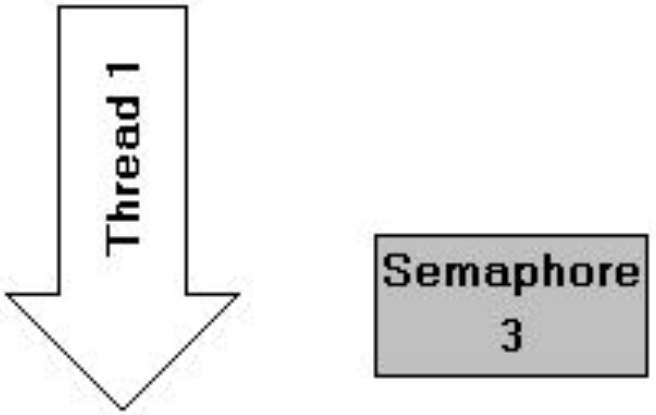


Семафоры

- Дийкстра (Dijkstra) предложил использовать переменные, которые могут принимать целые неотрицательные значения. Такие переменные, используемые для синхронизации вычислительных процессов, получили название семафоров.
- Семафор - неотрицательная целая переменная S >= 0, которая может изменяться и проверяться только посредством двух примитивов:
  - V(S): переменная S увеличивается на 1 единым неделимым действием. К переменной S нет доступа другим потокам во время выполнения этой операции.
  - P(S): уменьшение S на 1, если это возможно. Если S=0 и невозможно уменьшить S, оставаясь в области целых неотрицательных значений, то в этом случае поток, вызывающий операцию P, ждет, пока это уменьшение станет возможным. Успешная проверка и уменьшение также являются неделимой операцией.

Иллюстрация работы семафора

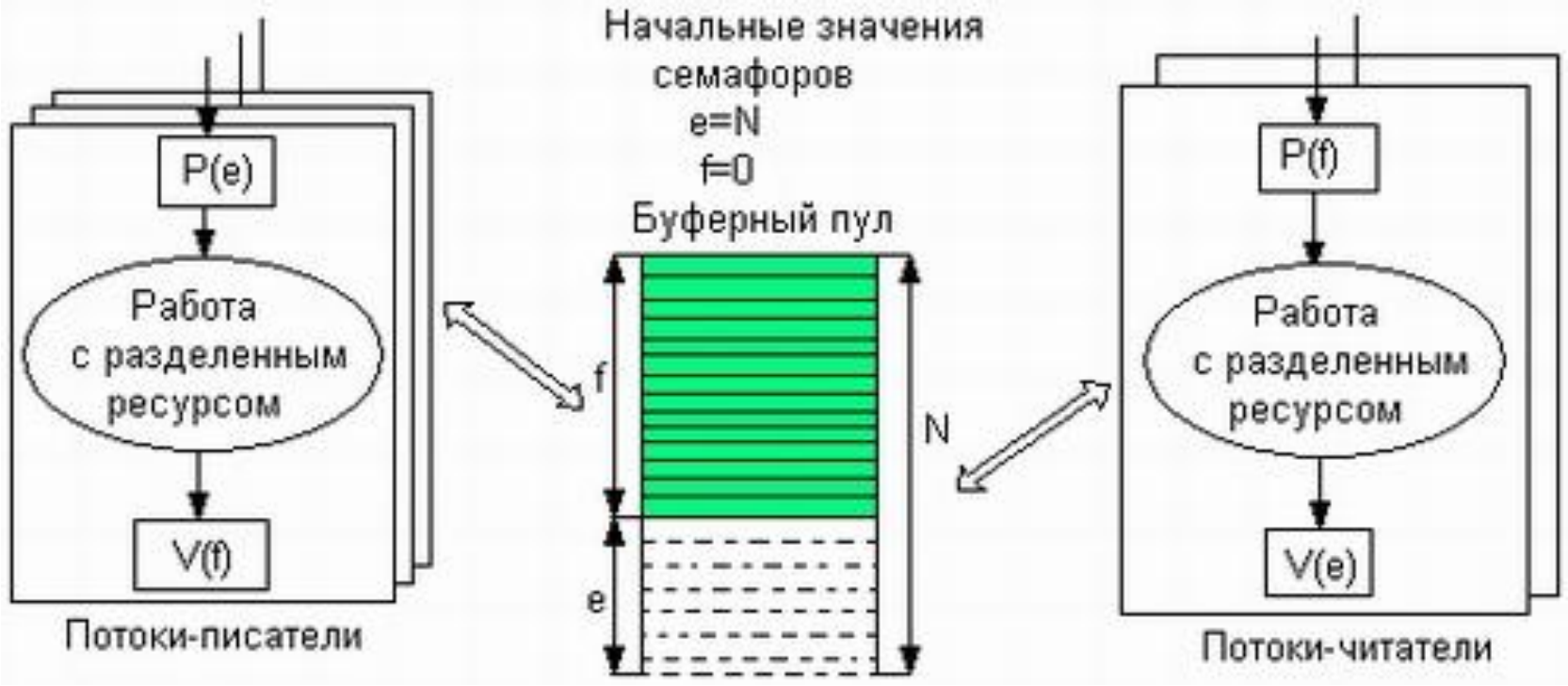
Данный пример демонстрирует использование семафора для ограничения доступа потоков к объекту синхронизации на основании их количества.



Задача о читателях и писателях

- Рассмотрим использование семафоров на классическом примере взаимодействия двух выполняющихся в режиме мультипрограммирования потоков, один из которых пишет данные в буферный пул, а другой считывает их из буферного пула.
- Пусть буферный пул состоит из N буферов, каждый из которых может содержать одну запись. В общем случае поток-писатель и поток-читатель могут иметь различные скорости и обращаться к буферному пулу с переменой интенсивностью. В один период скорость записи может превышать скорость чтения, в другой – наоборот.
- Для правильной совместной работы поток-писатель должен приостанавливаться, когда все буферы оказываются занятыми, и активизироваться при освобождении хотя бы одного буфера. Напротив, поток-читатель должен приостанавливаться, когда все буферы пусты, и активизироваться при появлении хотя бы одной записи.

Задача о читателях и писателях



Введем два семафора: e – число пустых буферов, и f – число заполненных буферов, причем в исходном состоянии e =N, а f =0. Тогда работа потоков с общим буферным пулом может быть описана следующим образом.

Задача о читателях и писателях

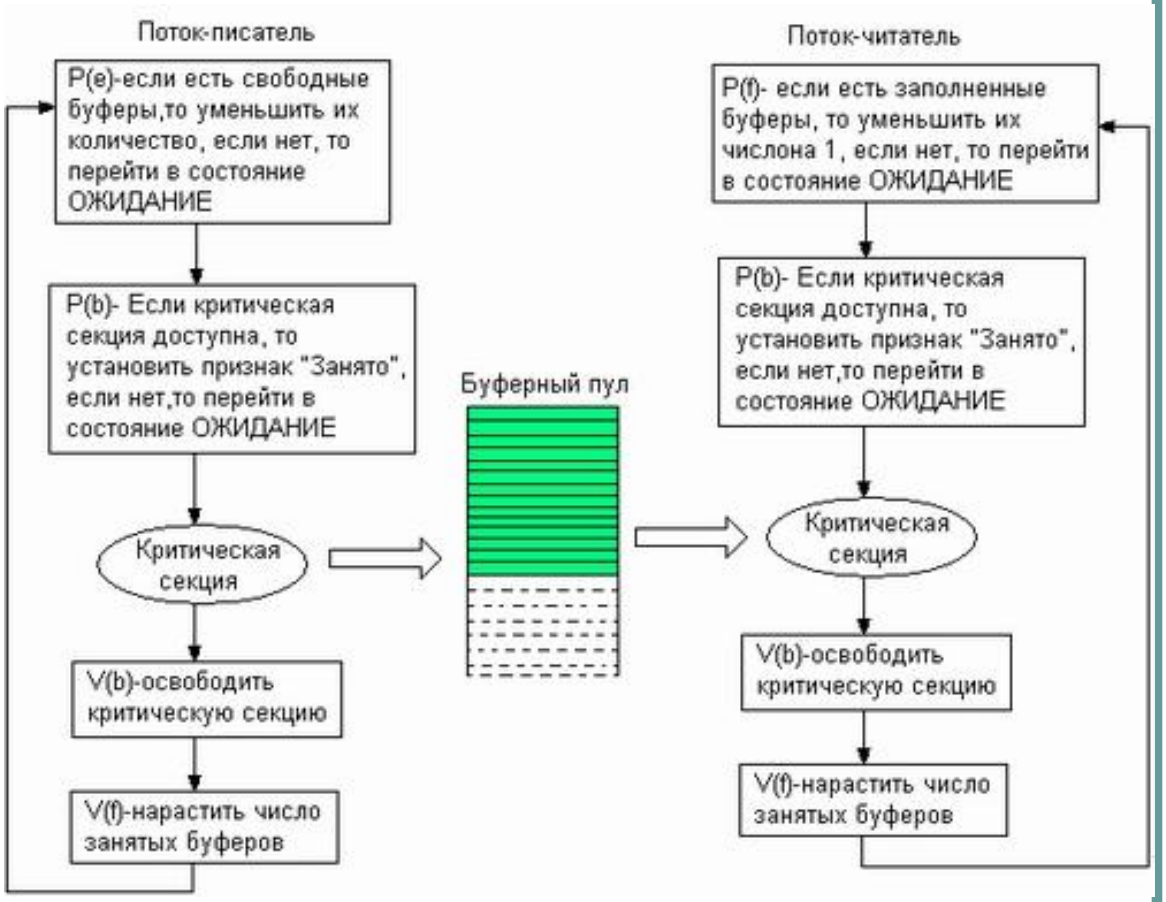
- Таким образом, семафоры позволяют эффективно решать задачу синхронизации Доступа к ресурсным пулам, таким, например, как набор идентичных в функциональном назначении внешних устройств (модемов, принтеров, портов), или набор областей памяти одинаковой величины, или информационных структур. Во всех этих и подобных им случаях с помощью семафоров можно организовать доступ к разделяемым ресурсам сразу нескольких потоков.

Мьютексы

- Иногда используется упрощенная версия семафора – мьютекс (mutex, mutual exclusion – взаимное исключение). Иногда называют еще двоичным семафором.
- Мьютекс – переменная, которая может находиться в одном из двух состояний: блокированном или неблокированном.
- Если процесс хочет войти в критическую секцию – он вызывает примитив блокировки мьютекса.
- Если мьютекс не заблокирован, то запрос выполняется и процесс попадает в критическую секцию.

Использование мьютекса

В рассмотренном примере, для того чтобы исключить коллизии при работе с разделяемой областью памяти, будем считать, что запись в буфер и считывание из буфера являются критическими секциями. Взаимное исключение будем обеспечивать с помощью двоичного семафора (мьютекса) b. Оба потока после проверки доступности буферов должны выполнить проверку доступности критической секции.



Атомарные операции

- В достаточно частых случаях необходимо обеспечить конкурентный доступ к какой-либо целочисленной переменной, являющейся счетчиком. Тогда бывает достаточно просто обеспечить атомарность выполнения операций увеличения, уменьшения или изменения значения переменной.
- Атомарные операции обычно имеют более-менее близкое соответствие с командами процессора. Так, например, они могут сводиться к операциям с блокировкой шины (префикс **lock**) и специальным командам (типа **cmpxchg**) процессора.

Атомарные операции в Windows 2000-2003

- ОС Windows предоставляет функции для увеличения (InterlockedIncrement, InterlockedIncrement64) или уменьшения (InterlockedDecrement, InterlockedDecrement64) значения целочисленных переменных и изменения их значений (InterlockedExchange, InterlockedExchange64, InterlockedExchangeAdd, InterlockedExchangePointer), в том числе со сравнением (InterlockedCompareExchange, InterlockedCompareExchangePointer).

Атомарные операции в Windows 2000-2003

- Пример использования атомарной операции:

```
static DWORD     array [100];
…
for (int i = 0; i < 100; i++)
  InterlockedIncrement(array+i);
```

Мониторы

- Для упрощения написания программ в 1974 г. Хоар (Hoare) и Бринч (Brinch Hansen) предложили примитив синхронизации более высокого уровня – монитор.
- Монитор – примитив синхронизации более высокого уровня.
- Монитор – набор процедур, переменных и других структур данных, объединенных в особый модуль.
- Пользовательские процессы могут вызывать процедуры монитора, но не могут получать доступ к внутренним структурам.
- Реализации взаимных исключений способствует важное свойство монитора – при обращении к монитору в любой момент времени может быть активен только один процесс. Реализация взаимного исключения реализуется с помощью мьютекса.
- Поскольку реализацию взаимного исключения выполняет компилятор, а не программист, вероятность ошибки уменьшается.

```
monitor example
        integer i;
        condition c;

        procedure producer( );

        .
        .
        .
        end;

        procedure consumer( );

        .
        .
        .
        end;
end monitor;
```

Решение задачи читателей и писателей с помощью монитора

```
monitor ProducerConsumer
        condition full, empty;
        integer count;
        procedure insert(item: integer);
        begin
                if count = N then wait(full);
                insert_item(item);
                count := count + 1;
                if count = 1 then signal(empty)
        end;
        function remove: integer;
        begin
                if count = 0 then wait(empty);
                remove = remove_item;
                count := count – 1;
                if count = N – 1 then signal(full)
        end;
        count := 0;
end monitor;
```

```
procedure producer;
begin
        while true do
        begin
                item = produce_item;
                ProducerConsumer.insert(item)
        end
end;
procedure consumer;
begin
        while true do
        begin
                item = ProducerConsumer.remove;
                consume_item(item)
        end
end;
```

Критические секции

- В составе API ОС Windows имеются специальные и эффективные функции для организации входа в критическую секцию и выхода из нее потоков одного процесса в режиме пользователя.
- Они называются *EnterCriticalSection* и *LeaveCriticalSection* и имеют в качестве параметра предварительно проинициализированную структуру типа CRITICAL_SECTION.

Критические секции

- Примерная схема программы может выглядеть следующим образом.

```
CRITICAL_SECTION cs;
DWORD WINAPI SecondThread()
{
    InitializeCriticalSection(&cs);
    EnterCriticalSection(&cs);
    //…критический участок кода
    LeaveCriticalSection(&cs);
}
```

Критические секции

- Функции *EnterCriticalSection* и *LeaveCriticalSection* реализованы на основе атомарных Interlocked-функций.
- Существенным является то, что в случае невозможности входа в критический участок поток переходит в состояние ожидания. Впоследствии, когда такая возможность появится, поток будет "разбужен" и сможет сделать попытку входа в критическую секцию.

Межпроцессное взаимодействие

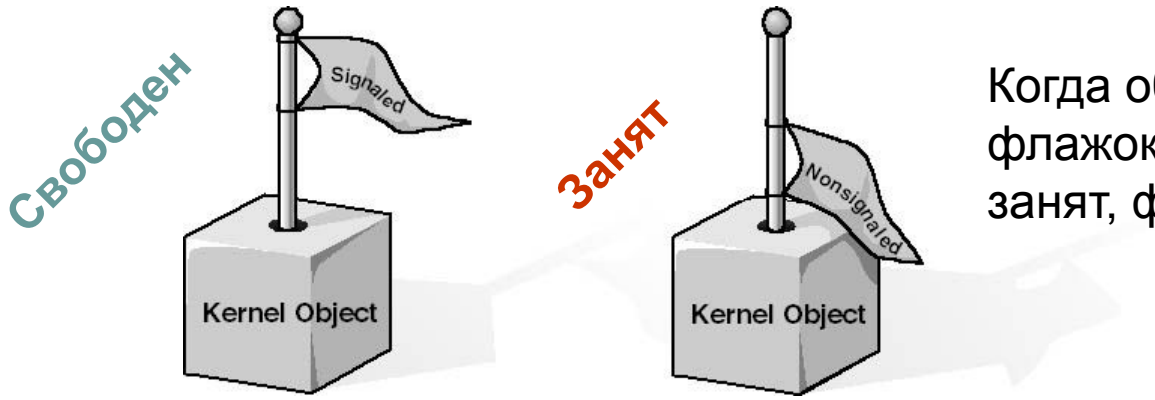
Синхронизация потоков с использованием объектов ядра Windows 2000

Синхронизация потоков

- Простейшей формой связи потоков является синхронизация (synchronization).
- **Синхронизация означает способность потока добровольно приостанавливать свое исполнение и ожидать, пока не завершится выполнение некоторой операции другим потоком.**
- Все ОС, поддерживающие многозадачность или мультипроцессорную обработку, должны предоставлять потокам способ ожидания того, что другой поток что–либо сделает: например, освободит накопитель на магнитном диске или закончит запись в совместно используемый буфер памяти. ОС должна также дать потоку возможность сообщить другим потокам об окончании выполнения операции. Получив такое уведомление, ожидающий поток может продолжить выполнение.

Синхронизационные объекты и их состояния

- процессы
- потоки
- задания
- файлы
- консольный ввод
- уведомления об изменении файлов
- события
- ожидаемые таймеры
- семафоры
- мьютексы

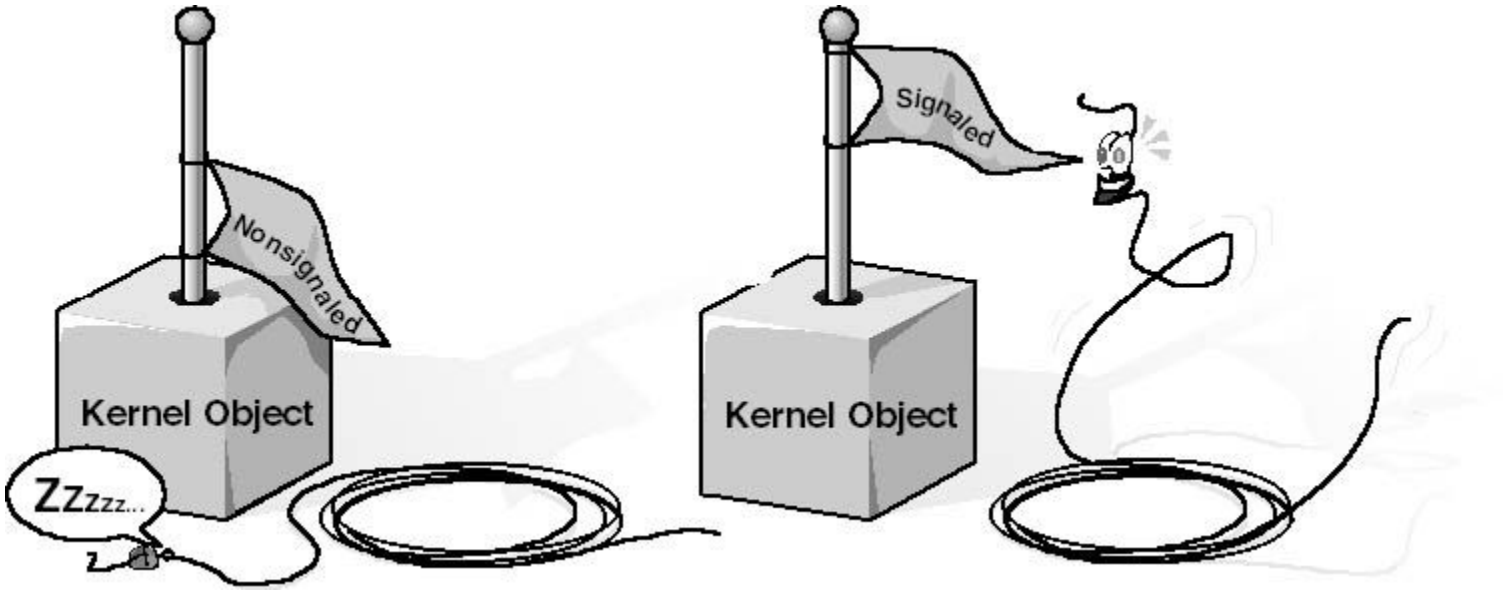


Когда объект свободен, флажок поднят, а когда он занят, флажок опущен.

Объекты синхронизации и их состояния

- Средства ожидания и сообщения реализованы в ядре Windows как часть объектной архитектуры.
- Синхронизационные объекты (synchronization objects) – это объекты ядра, при помощи которых поток синхронизирует свое выполнение.
- В любой момент времени синхронизационный объект находится в одном из двух состояний: свободен (signaled state) или занят.
- Правила, по которым объект переходит в свободное или занятое состояние, зависят от типа этого объекта:
  - Объект-поток находится в состоянии «занят» все время существования, но устанавливается системой в состояние "свободен", когда его выполнение завершается. Аналогично, ядро устанавливает процесс в состояние "свободен", когда завершился его последний поток.
  - В противоположность этому, объект – таймер «срабатывает» через заданное время (по истечении этого времени ядро устанавливает объект – таймер в состояние "свободен").

Спящие потоки



- Потоки спят, пока ожидаемые ими объекты заняты (флажок опущен).
- Как только объект освободился (флажок поднят), спящий поток замечает это, просыпается просыпается и возобновляет выполнение.

Функции ожидания

```
DWORD WaitForSingleObject(
    HANDLE hObject,
    DWORD dwMilliseconds
);
```

```
DWORD WaitForMultipleObjects(
    DWOHD dwCount,
    CONST HANDLE* phObjects,
    BOOL fWaitAll,
    DWORD dwMilliseconds
);
```

```
WaitForSingleObject (hProcess, INFINITE);
```

Функции ожидания

- Wait-функции позволяют потоку в любой момент приостановиться и ждать освобождения какого-либо объекта ядра.
- Когда поток вызывает эту функцию, первый параметр, *hObject,* идентифицирует объект ядра, поддерживающий состояния «свободен-занят» (То есть любой объект, упомянутый в списке из предыдущего раздела.) Второй параметр, *dwMilliseconds,* указывает, сколько времени (в миллисекундах) поток готов ждать освобождения объекта.
- Следующий вызов сообщает системе, что поток будет ждать до тех пор, пока не завершится процесс, идентифицируемый описателем *hProcess.*
- Функция *WaitForMultipleObjects* аналогична *WaitForSingleObject* с тем исключением, что позволяет ждать освобождения сразу нескольких объектов или какого-то одного из списка объектов.
- Параметр *dwCount* определяет количество интересующих Вас объектов ядра Его значение должно быть в пределах от 1 до MAXIMUM_WAIT_OBJECTS (в заголовочных файлах Windows оно определено как 64). Параметр *phObject* — это указатель на массив описателей объектов ядра.
- *WaitForMultipleObjects* приостанавливает поток и заставляет его ждать освобождения либо всех заданных объектов ядра, либо одного из них. Параметр *fWaitAll* как раз и определяет, чего именно Вы хотите от функции. Если он равен TRUE, функция не даст потоку возобновить свою работу, пока не освободятся все объекты.

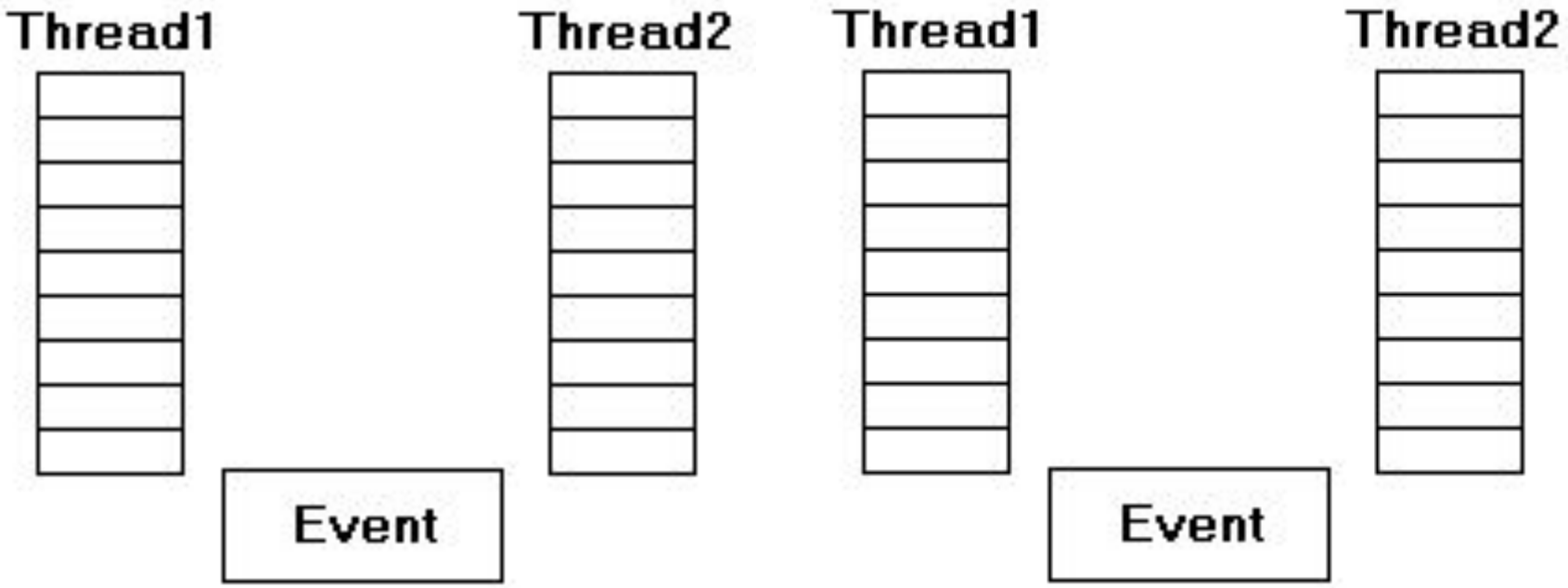
Объекты синхронизации

- События
- Ожидающий таймер
- Семафор
- Мьютекс

События

- События – самая примитивная разновидность объектов ядра.
- События содержат счетчик числа пользователей (как и все объекты ядра) и две булевы переменные: одна сообщает тип данного объекта-события, другая – его состояние (свободен или занят).
- События просто уведомляют об окончании какой-либо операции. Объекты-события бывают двух типов: со сбросом вручную (manual-reset events) и с автосбросом (auto-reset events). Разница в том, что первый вид события нужно применять если событие ждут несколько потоков. Только сброс вручную позволяет это сделать. Иначе первый же обработчик сбросит событие и другие потоки об этом не узнают.
- Объекты-события обычно используют в том случае, когда какой-то поток выполняет инициализацию, а затем сигнализирует другому потоку, что тот может продолжить работу. Инициализирующий поток переводит объект "событие” в занятое состояние и приступает к своим операциям. Закончив, он сбрасывает событие в свободное состояние. Тогда другой поток, который ждал перехода события в свободное состояние, пробуждается и вновь становится планируемым.

Иллюстрация работы “события”



событие с ручным сбросом

событие с автоматическим сбросом

Создание события

```
HANDLE CreateEvent(
    LPSECURITY_ATTRIBUTES psa,
    BOOL fManualReset,
    BOOL fInitialState,
    LPCTSTR pszName);

HANDLE OpenEvent(
    DWORD fdwAccess,
    BOOL fInhent,
    LPCTSTR pszName);
```

Создание события

- Объект ядра “событие" создается функцией *CreateEvent.*
- Параметр *fManualReset* (булева переменная) сообщает системе, хотите Вы создать событие со сбросом вручную (TRUE) или с автосбросом (FALSE).
- Параметр *fInitialState* определяет начальное состояние события – свободное (TRUE) или занятое (FALSE). После того как система создает объект событие, *CreateEvent* возвращает описатель события, специфичный для конкретного процесса.
- Потоки из других процессов могут получить доступ к этому объекту:
  1) наследованием описателя с применением функции *DuplicateHandle;*
  2) вызовом *OpenEvent* с передачей в параметре *pszName* имени, совпадающего с указанным в аналогичном параметре функции *CreateEvent.*

Управление событием

- Перевод события в свободное состояние:
  BOOL SetEvent (HANDLE hEvenet);
- Перевод события в занятое состояние:
  BOOL ResetEvent (HANDLE hEvent);
- Освобождение события и перевод его обратно в занятое состояние:
  BOOL PulseEvent (HANDLE hEvent);

Особенности PulseEvent

- Функция *PulseEvent* устанавливает событие и тут же переводит его обратно в сброшенное состояние.
- Ее вызов равнозначен последовательному вызову *SetEvent* и *ResetEvent*. Если PulseEvent вызывается для события со сбросом вручную, то все потоки, ожидающие этот объект, получают управление.
- При вызове *PulseEvent* для события с автосбросом пробуждается только один из ждущих потоков. А если ни один из потоков не ждет объект-событие, вызов функции не дает никакого эффекта.

Ожидающие таймеры

- Ожидающие таймеры (waitable timers) – это объекты ядра, которые самостоятельно переходят в свободное состояние в определенное время или через регулярные промежутки времени.

Создание ожидающего таймера

```
HANDLE CreateWaitableTimer(
    LPSECURITY_ATTRIBUTES psa,
    BOOL fManualReset,
    LPCTSTR pszName);

HANDLE OpenWaitableTimer(
    DWORD dwDesiredAccess,
    BOOL bInheritHandle,
    LPCTSTR pszName);
```

- По аналогии с событиями параметр *fManualReset* определяет тип ожидаемого тай мера: со сбросом вручную или с автосбросом. Когда освобождается таймер со сбросом вручную, возобновляется выполнение всех потоков, ожидавших этот объект, а когда в свободное состояние переходит таймер с автосбросом — лишь одного из потоков.
- Объекты «ожидающий таймер» всегда создаются в занятом состоянии.

Управление ожидающим таймером

```
BOOL SetWaitableTimer(
    HANDLE hTimer,
    const LARGE_INTEGER *pDueTime,
    LONG lPeriod,
    LPTIMERAPCROUTINE pfnCompletionRoutine,
    LPVOID pvArgToCompletionRoutine,
    BOOI fResume);

BOOL CancelWaitableTimer (HANDLE hTimer);
```

Управление ожидаемым таймером

- Эта функция принимает несколько параметров, в которых легко запутаться. Очевидно, что *hTimer* определяет нужный таймер. Следующие два параметра (*pDueTime* и *lPeriod)* используются совместно, первый из них задает, когда таймер должен сработать в первый раз, второй определяет, насколько часто это должно происходить в дальнейшем.
- Обычно нужно, чтобы таймер сработал только раз – через определенное (абсолютное или относительное) время перешел в свободное состояние и уже больше никогда не срабатывал Для этого достаточно передать 0 в параметре *lPeriod* Затем можно либо вызвать *CloseHandle,* чтобы закрыть таймер, либо перенастроить таймер повторным вызовом *SetWaitableTimer c* другими параметрами.
- Параметр *Resume* полезен на компьютерах с поддержкой режима сна. Обычно в нем передают FALSE, и в приведенных ранее фрагментах кода я тоже делал так. Но если Вы, скажем, пишете программу-планировщик, которая позволяет устанавливать таймеры для напоминания о запланированных встречах, то должны передавать в этом параметре TRUE Когда таймер сработает, машина выйдет из режима сна (если она находилась в нем), и пробудятся потоки, ожидавшие этот таймер. Далее программа сможет проиграть какой-нибудь WAV-файл и вывести окно с напоминанием о предстоящей встрече.
- Функция *CancelWaitableTimer* принимает описатель таймера и отменяет его (тай мер), после чего тот уже никогда не сработает, — если только Вы не переустановите его повторным вызовом *SetWaitableTimer.* Кстати, если Вам понадобится перенастроить таймер, то вызывать *CancelWaitableTimer* перед повторным обращением к *SetWaitableTimer* не требуется; каждый вызов *SetWaitableTimer* автоматически отменяет предыдущие настройки перед установкой новых.

Создание семафора

```
HANDLE CreateSemaphore(
    LPSECURITY_ATTRIBUTE psa,
    LONG lInitialCount,
    LONG lMaximumCount,
    LPCTRTR pszName);

HANDLE OpenSemaphore(
    DWORD fdwAccess,
    BOOL bInheritHandle,
    LPCTSTR pszName);
```

- Параметр *lMaximumCount* сообщает системе максимальное состояние семафора.

Управление семафором

- Поток получает доступ к ресурсу, вызывая одну из *Wait*-функций и передавая ей описатель семафора, который охраняет этот ресурс *Wait*-функция проверяет у семафора счетчик текущего числа ресурсов если его значение больше 0 (семафор свободен), уменьшает значение этого счетчика на 1, и вызывающий поток остается планируемым.

```
BOOL ReleaseSemaphore(
    HANDLE hSem,
    LONG lReleaseCount,
    PLONG plPreviousCount);
```

Управление семафором

- Если *Wait*-функция определяет, что счетчик текущего числа ресурсов равен 0 (семафор занят), система переводит вызывающий поток в состояние ожидания Когда другой поток увеличит значение этого счетчика, система вспомнит о ждущем потоке и снова начнет выделять ему процессорное время (а он, захватив ресурс, уменьшит значение счетчика на 1).
- Поток увеличивает значение счетчика текущего числа ресурсов, вызывая функцию *ReleaseSemaphore*.
- Она просто складывает величину *lReleaseCount* со значением счетчика текущего числа ресурсов. Обычно в параметре *lReleaseCount* передают 1.
- Функция возвращает исходное значение счетчика ресурсов в **plPreviousCount*.

Создание мьютекса

```
HANDLE CreateMutex(
    LPSECURITY_ATTRIBUTES psa,
    BOOL fInitialOwner,
    LPCTSTR pszName);

HANDLE OpenMutex(
    DWORD fdwAccess,
    BOOL fInheritHandle,
    LPCTSTR pszName);
```

- Параметр *fInitialOwner* определяет начальное состояние мьютекса:
  - Если в нем передается FALSE (что обычно и бывает), объект-мьютекс не принадлежит ни одному из потоков и поэтому находится в свободном состоянии. При этом его идентификатор потока и счетчик рекурсии равны 0
  - Если же в нем передается TRUE, идентификатор потока, принадлежащий мьютексу, приравнивается идентификатору вызывающего потока, а счетчик рекурсии получает значение 1. Поскольку теперь идентификатор потока отличен от 0, мьютекс изначально находится в занятом состоянии.

Управление мьютексом

- Поток получает доступ к разделяемому ресурсу, вызывая одну из *Wait*-функций и передавая ей описатель мьютекса, который охраняет этот ресурс. *Wait-функция* проверяет у мьютекса идентификатор потока, если сго значение не равно 0, мьютекс свободен, в ином случае оно принимает значение идентификатора вызывающего потока, и этот поток остается планируемым.
- Если *Wait*-функция определяет, что у мьютекса идентификатор потока не равен 0 (мьютекс занят), вызывающий поток переходит в состояние ожидания. Система за поминает это и, когда идентификатор обнуляется, записывает в него идентификатор ждущего потока, а счетчику рекурсии присваивает значение 1, после чего ждущий поток вновь становится планируемым.
- Когда ожидание мьютекса потоком успешно завершается, последний получает монопольный доступ к защищенному ресурсу. Все остальные потоки, пытающиеся обратиться к этому ресурсу, переходят в состояние ожидания. Когда поток, занимающий ресурс, заканчивает с ним работать, он должен освободить мьютекс вызовом функции *ReleaseMutex*.

BOOL ReleaseMutex (HANDLE hMutex);

Межпроцессное взаимодействие

Передача информации

Взаимодействие между процессами (IPC)

- DDE (Dynamic Data Exchange),
- OLE,
- atom (атомы)
- pipes (анонимные каналы),
- named pipes (именованные каналы)
- почтовые ящики (mailslots)
- RPC
- сокеты
- файлы, проецируемые в память (memory-mapped files)
- разделяемая память (Shared Memory). Отличается от предыдущего способа только тем, что в качестве разделяемого файла используется часть файла подкачки.

Взаимодействие между процессами (IPC)

Сообщения WM_COPYDATA	Лучший способ пересылки блока данных из одной программы в другую.
Анонимные каналы (Anonymous pipes)	Полезны для организации прямой связи между двумя процессами на одном ПК.
Именованные каналы (Named pipes)	Полезны для организации прямой связи между двумя процессами на одном ПК или в сети.
Почтовые ячейки (mailslots)	Полезны для организации связи одного процесса со многими на одном ПК или в сети.
Гнезда (sockets)	Полезны для организации пересылки данных в гетерогенных средах.
Вызов удаленных процедур RPC	Слишком сложен, чтобы использовать его для простых пересылок данных.
Разделяемая память	Непросто выделить вне DLL.
Файлы отображаемой памяти	Обеспечивают одновременный доступ к объектам файла отображения из нескольких процессов.

Атомы

- Атомы - это очень простой и доступный путь IPC. Идея состоит в том, что процесс может поместить строку в таблицу атомов и эта строка будет видна другим процессам. Когда процесс помещает строку в таблицу атомов, он получает 32-х битное значение (атом), и это значение используется для доступа к строке. Система не различает регистр строки.
- Набор атомов собирается в таблицу (**atom table**). Система обеспечивает несколько таблиц атомов для разных задач. По типу доступа их два типа:
  - Локальные (доступны только из приложения)
  - Глобальные (доступны из всех приложений)

Атомы

- GlobalAddAtom
- GlobalGetAtomName
- GlobalFindAtom
- GlobalDeleteAtom

Сообщение WM_COPYDATA

Отправитель:

```
COPYDATASTRUCT cds;
cds.cbData = (DWORD) nSize;
cds.lpData = (PVOID) pBuffer;
SendMessage (hWndTarget, WM_COPYDATA,
    (WPARAM) hWnd, (LPARAM) &cds);
```

Получатель:

```
PCOPYDATASTRUCT pcds = (PCOPYDATASTRUCT)
    lParam;
PBYTE pBuffer = (PBYTE) pcds -> lpData;
```

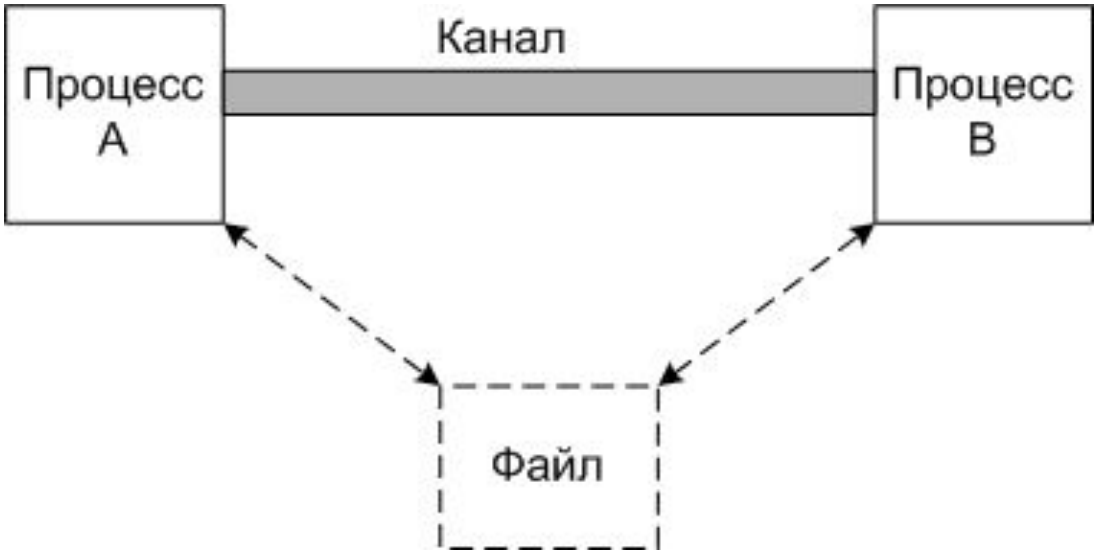
Сообщение WM_COPYDATA

- Сообщение WM_COPYDATA позволяет приложениям копировать данные между их адресными пространствами. При этом приложения не обязательно должны быть 32-разрядными — для 16-разрядных приложений поддерживается автоматическая трансляция указателей.
- Перед отправкой сообщения WM_COPYDATA необходимо инициализировать структуру COPYDATASTRUCT с информацией о предстоящей пересылке данных, в том числе с указателем на блок данных. Затем с помощью функции SendMessage сообщение WM_COPYDATA пересылается в принимающую программу; при этом параметр wParam содержит дескриптор окна вашей программы, а lParam - адрес структуры COPYDATASTRUCT.
- Когда сообщение поступает обработчику WM_COPYDATA принимающей программы, его средствами указатель может быть скопирован из структуры COPYDATASTRUCT и использован, как любой другой указатель.
- В принимающем процессе размер блока данных, адрес которого содержит lpData, извлекается из элемента cbData. Значение, считываемое из элемента lpData принимающей программой, возможно, будет отличаться от значения, помещенного туда отправляющей программой. Этого следовало ожидать, поскольку перед передачей сообщения WM_COPYDATA операционная система Windows NT выделяет в адресном пространстве принимающего процесса блок памяти, копирует данные из отправляющего процесса и обновляет значение lpData. По всей вероятности, в адресных пространствах этих процессов адреса размещении блока не совпадут.
- В структуре COPYDATASTRUCT имеется третье, необязательное поле, dwData, в котором можно передать 32-разрядное значение, определяемое в программе. Только не передавайте в этом поле указатель, потому что в принимающем процессе он не будет воспринят.
- При использовании метода WM_COPYDATA необходимо помнить об одной детали: сообщения должны именно пересылаться, а не просто регистрироваться. Для того чтобы освободить память, выделенную в адресном пространстве принимающего процесса, операционная система должна быть информирована о моменте завершении пересылки. Кроме того, получатель сообщения WM_COPYDATA должен обращаться с блоком данных так, словно он предназначен только для чтения. Чтобы внести изменении в полученные данные, в принимающей программе необходимо подготовить их локальную копию. И наконец, не следует сохранять указатель, переданный в элементе lpData, дл применения его в дальнейшем.

Анонимные каналы

- Анонимные каналы не имеют имен.
- Не пригодны для обмена через сеть.
- Главная цель – служить каналом между родительским и дочерним процессом или между дочерними процессами.
- Односторонний обмен.
- Не возможен асинхронный обмен.

Каналы



Канал представляет собой псевдофайл с организацией типа буфера FIFO (first input and first output - первый вошел, первый вышел). Образно говоря, канал представляет собой трубу (pipe) с двумя открытыми концами, в который один процесс пишет, а другой читает.

Использование анонимных каналов

- Главная цель – служить каналом между родительским и дочерним процессом или между дочерними процессами.
- Родительский процесс может быть консольным или GUI-приложение, но дочернее приложение должно быть консольным. Как вы знаете, консольное приложение использует стандартные дескрипторы для ввода и вывода.
- Если мы хотите перенаправить ввод/вывод консольного приложения, мы можем заменить один дескриптор другим дескриптором одного конца канала. Консольное приложение не будет знать, что оно использует один конец канала. Оно будет считать, что это стандартный дескриптор. Это вид полиморфизма на ООП-жаргоне.
- Это мощный подход, так как нам не нужно модифицировать родительский процесс ни каким образом.

Создание анонимных каналов

```
BOOL CreatePipe(
    LPHANDLE hReadPipe,
    LPHANDLE hWritePipe,
    LPSECURITY_ATTRIBUTES lpPipeAttributes,
    DWORD nSize
);
```

ReadFile – чтение из канала

WriteFile – запись в канал

Создание анонимных каналов

- pReadHandle - это указатель на переменную типа dword, которая получит дескриптов конца чтения канала.
- pWriteHandle - это указатель на переменную типа dword, которая получить дескриптов конца записи канала.
- pPipeAttributes указывает на структуру SECURITY_ATTRIBUTES, которая определяет, наследуется ли каждый из концов дочерним процессом.
- nBufferSize - это предполагаемый размер буфера, который канал зарезервирует для использования. Это всего лишь предполагаемый размер. Вы можете передать NULL, чтобы указать функции использовать размер по умолчанию.
- Не забудьте установить параметр bInheritable структуры SECURITY_ATTRIBUTES в TRUE, чтобы дескриптры могли наследоваться.

Передача дескрипторов

- Установить параметр bInheritable структуры SECURITY_ATTRIBUTES в TRUE, чтобы дескрипторы могли наследоваться.
- Вызов функции CreateProcess с параметром bInheritHandles = TRUE
- Передача дескрипторов (командная строка, сообщения…)
- Вызов функции DuplicateHandle

Дубликаты дескрипторов

```
BOOL DuplicateHandle(
    HANDLE hSourceProcessHandle,
    HANDLE hSourceHandle,
    HANDLE hTargetProcessHandle,
    LPHANDLE lpTargetHandle,
    DWORD dwDesiredAccess,
    BOOL bInheritHandle,
    DWORD dwOptions
);
```

Дубликаты дескрипторов

- Первый и третий параметры функции DuplicateHandle представляют собой описатели объектов ядра, специфичные для вызывающего процесса Кроме того, эти параметры должны идентифицировать именно процессы — функция завершится с ошибкой, если Вы передадите описатели на объекты ядра любого другого типа.
- Второй параметр, hSourceHandle, — описатель объекта ядра любого типа. Однако его значение специфично не для процесса, вызывающего DuplicateHandle, а для того, на который указывает описатель hSourceProcessHandie. Параметр phTargetHandle — это адрес переменной типа HANDLE, в которой возвращается индекс записи с копией описателя из процесса-источника. Значение возвращаемого описателя специфично для процесса, определяемого параметром phTargetProcessHandle.
- Предпоследние два параметра DuplicateHandle позволяют задать маску доступа и флаг наследования, устанавливаемые для данного описателя в процессе-приемнике. И, наконец, параметр dwOptions может быть 0 или любой комбинацией двух флагов. DUPLICATE_SAME_ACCESS и DUPLICATE_CLOSE_SOURCE
- Первый флаг подсказывает DuplicateHandle: у описателя, получаемого процессом-приемником, должна быть та же маска доступа, что и у описателя в процессе-источнике Этот флаг заставляет DuplicateHandle игнорировать параметр dwDesiredAccess.
- Второй флаг приводит к закрытию описателя в процессе-источнике. Он позволяет процессам обмениваться объектом ядра как эстафетной палочкой При этом счетчик объекта не меняется.

Пример использования анонимного канала

- Создаем анонимный канал с помощью CreatePipe.
- Теперь мы должны подготовить параметры, которые передадим CreateProcess (мы используем эту функцию для загрузки консольного приложения).
- Вызоваем CreateProcess, чтобы загрузить дочернее приложение. После того, как вызов прошел успешно, дочерний процесс все еще находится в спящем состоянии. Он загружается в память, но не запускается немедленно.
- Закройте дескриптор записи канала. Это необходимо, так как родительскому процессу нет нужды использовать этот дескриптор, а канал не будет работать, если открыть более чем один дескриптор записи.
- Теперь вы можете читать данные с помощью ReadFile. Вы должны последовательно вызывать ReadFile, пока она не возвратит ноль, что будет означать, что больше данных нет.
- Закроем дескриптор чтения канала.

NPFS (Named Pipe File System)

- **Named Pipe File System** является виртуальной файловой системой, которая управляет каналами **named pipes**.
- Каналы named pipes относятся к классу файловых объектов (API Win32).
- RPC реализован как надстройка над NPFS;
- Канал представляет собой виртуальное соединение, по которому передается информация от одного процесса к другому.
- Канал может быть однонаправленным или двунаправленным (дуплексным).

Работа с именованными каналами

- Серверный процесс создает канал на локальном компьютере с помощью функции программного интерфейса Win32 **"CreateNamedPipe"**.
- Серверный процесс активизирует канал при помощи функции **"ConnectNamedPipe"**, после чего к каналу могут подключаться клиенты.
- Далее производится подключение к каналу \\computer_name\pipe\pipe_name посредством вызова функции **"Create File"**.

Создание именованного канала

```
HANDLE CreateNamedPipe (
    LPCTSTR lpName,
    DWORD dwOpenMode,
    DWORD dwPipeMode,
    DWORD nMaxInstances,
    DWORD nOutBufferSize,
    DWORD nInBufferSize,
    DWORD nDefaultTimeOut,
    LPSECURITY_ATTRIBUTES
    lpSecurityAttributes
);
```

Параметры создания канала

- lpName – имя именованного канала;
- dwOpenMode – определяет направление передачи, возможные варианты - **PIPE_ACCESS_DUPLEX**, **PIPE_ACCESS_INBOUND**, **PIPE_ACCESS_OUTBOUND** ;
- dwPipeMode – способ передачи информации (**PIPE_TYPE_BYTE** или **PIPE_TYPE_MESSAGE**
- nMaxInstances – количество каналов с данным именем которые может открыть пользователь;
- nOutBufferSize и nInBufferSize – размер буферов приема и отправки;
- nDefaultTimeout – максимальное время ожидания при асинхронном вводе/выводе через канал;
- lpSecurityAttributes – указатель на структуру **SECURITY_ATTRIBUTES**, которая задает уровень защиты создаваемого объекта.

Подключение к именованному каналу

```
BOOL ConnectNamedPipe (
    HANDLE hNamedPipe,
    LPOVERLAPPED lpOverlapped
);

BOOL DisconnectNamedPipe (
    HANDLE hNamedPipe
);
```

Подключение к именованному каналу

- После того как канал создан, сервер подключается к нему с помощью функции **ConnectNamedPipe** () и начинает ожидать подключения клиента.
- Необходимо отметить, что подключение сервера к каналу может осуществляться как синхронным, так и асинхронным способом. В первом случае ConnectNamePipe возвращает управление программе лишь после того, как клиент подключился к каналу, во втором же случае, возврат управления происходит сразу же, а уведомление программы о подключении осуществляется через структуру **OVERLAPPED** , указатель на которую передается вторым параметром в ConnectNamedPipe.
- После завершения обмена необходимо отключиться от канала с помощью функции **DisconnectNamedPipe**.
- Затем можно снова открыть канал и ожидать подключения следующего клиента, а по завершению работы с каналом необходимо закрыть его дескриптор функцией **CloseHandle** ().

Обмен данными по именованному каналу

```
BOOL ReadFile/WriteFile (
    HANDLE hFile,
    LPVOID lpBuffer,
    DWORD nNumberOfBytesToRead,
    LPDWORD lpNumberOfBytesRead,
    LPOVERLAPPED lpOverlapped
);
```

Работа с каналом и ее завершение

- После установления виртуального соединение серверный процесс и клиентский процесс могут обмениваться информацией при помощи пар функций **"ReadFile"** и **"WriteFile"**.
- При помощи одного и того же канала сервер может одновременно обслуживать нескольких клиентов. Для этого серверный процесс может создать N-ное количество экземпляров канала, вызвав N-ное количество раз функцию **"CreateNamedPipe"** (при этом в каждом вызове должно быть указано одно и то же имя канала).
- Клиентский процесс может отключиться от канала в любой момент с помощью функции **"CloseHandle"**. Серверный процесс может отключить клиента в любой момент с помощью функции **"DisconnectNamedPipe"**.

Пример клиент-серверного приложения (сервер)

```
HANDLE hPipe =
    CreateNamedPipe("\\\\.\\pipe\\PipeSrv",PIPE_ACCESS_DUPLEX
    | WRITE_DAC, PIPE_TYPE_BYTE,1,100,100,100,NULL);
if (hPipe==INVALID_HANDLE_VALUE) {
    …//Обработка ошибки создания канала
}
ConnectNamedPipe(hPipe,NULL);
DWORD lpBuf; char cName[100];
ZeroMemory(&cName[0],sizeof(cName));
strcpy(&cName[0],"Hello world!");
WriteFile(hPipe,&cName,sizeof(cName),&lpBuf,NULL);
DisconnectNamedPipe(hPipe);
CloseHandle(hPipe);
```

Пример клиент-серверного приложения (клиент)

- Работа с именованным каналом на клиентской стороне еще проще чем на серверной.
- Для чтения и записи информации используются уже знакомые нам функции ReadFile и WriteFile, дескриптор канала закрывается аналогичным образом.
- Вся разница заключается лишь в способе открытия канала. Делается это с помощью функции *CreateFile*, в которую в качестве имени файла передается строка следующего формата: **\\ServerName\pipe\PipeName**.

Пример клиент-серверного приложения (клиент)

```
…
char szName [] = “\\ServerName\pipe\PipeSrv”;
HANDLE hFile = CreateFile(szName,GENERIC_READ |
    GENERIC_WRITE, 0,NULL,OPEN_EXISTING,0,NULL);

if (hFile==INVALID_HANDLE_VALUE) {
    …//Обработка ошибки открытия канала
}

char str[100]; DWORD lpBuff;
ZeroMemory(&str[0],sizeof(str));
ReadFile(hFile,str,sizeof(str),&lpBuff,NULL);
printf("Server sent:\n%s",str);
CloseHandle(hFile);
```

Почтовые ящики (MailSlots)

- *Mailslot* является одним из механизмов, предназначенных для осуществления обмена данными между процессами (IPC). При этом процессы могут быть запущены как на одной ПЭВМ (локально), так и разных ПЭВМ, включённых в одну ЛВС (удалённо).
- Приложение-сервер открывает почтовый ящик, а клиенты могут писать в него. Ящик сохраняет сообщения до тех пор, пока сервер их не прочтет. Разумеется, одно приложение может одновременно быть сервером и клиентом, обеспечивая двунаправленную связь. При этом приложения могут находиться даже на разных компьютерах в сети.

Почтовые ящики (MailSlots)

- *Mailslot* представляет собой псевдофайл, хранящийся в памяти. Для доступа к данным, содержащимся в этом псевдофайле, используются стандартные файловые функции Win32.
- Объект *Mailslot* является временным объектом. После того, как будет закрыт последний дескриптор, ссылающийся на объект *Mailslot*, сам объект с данными будет уничтожен.
- Обмен данными посредством *Mailslot* осуществляется в большинстве случаев между двумя процессами – *клиентом* и *сервером*.

Формат имени сервера

- При создании объекта *Mailslot* сервером, имя объекта должно иметь следующий формат: *\\.\mailslot\[path]name* Имя объекта *Mailslot* должно содержать две наклонные черты влево, точку, ещё одну наклонную черту влево, слово «*mailslot*» и последнюю наклонную черту. После последней наклонной черты указывается собственно имя создаваемого объекта. Имя может также содержать путь. Примеры имени создаваемого объекта *Mailslot*:
  - *\\.\mailslot\Test.msl*
  - *\\.\mailslot\SampleDir\Sample*

Форматы имени клиента

- Для того чтобы записать сообщение в *Mailslot*, клиент обращается к нему по имени. При этом если клиент и сервер запушены на одной ПЭВМ, то формат имени, используемый клиентом, может совпадать с форматом имени сервера. Однако чаще необходимо записывать сообщения в удалённый *Mailslot*, для чего необходимо использовать следующий формат имени объекта *Mailslot*: *\\ComputerName\mailslot\Test.msl* Здесь *ComputerName* – сетевое имя ПЭВМ, на которой расположен сервер *Mailslot*.
- Для того чтобы клиент мог поместить сообщение в каждый *Mailslot* с данным именем, созданный в пределах домена, формат имени объекта *Mailslot* для клиента должен быть следующим: *\\DomainName\mailslot\Test.msl* Здесь *DomainName* – имя домена, включающие в себя те ПЭВМ, на которых запущены серверы *Mailslot*. Для того чтобы клиент мог поместить сообщение в каждый *Mailslot* с данным именем в первичном системном домене, имя объекта *Mailslot* должно иметь следующую форму: **\mailslot\Test.msl*

Клиенты, сервера и имена

- **MailSlot** сервер – является процессом, который создает и, обладает **MailSlot**. Когда сервер создает **MailSlot**, он получает указатель. Этот указатель должен использоваться, когда процесс читает сообщения от **MailSlot**. Только процесс, который создает **MailSlot** или получил указатель некоторым другим механизмом может прочитать данные из **MailSlot**. Все **MailSlot** локальные на процессе, который создает их; процесс не может создать дистанционный **MailSlot**.
- **MailSlot** клиент – является процессом, который пишет сообщение в **MailSlot**. Любой процесс, который имеет имя **MailSlot** может записать в него информацию.

Создание почтового ящика на сервере

```
HANDLE CreateMailslot (
    // имя ящика
    LPCTSTR lpName,
    // максимальный размер сообщения
    DWORD nMaxMessageSize,
    // интервал-тайм аута чтения
    DWORD lReadTimeout,
    // информация о безопасности
    LPSECURITY_ATTRIBUTES  lpSecurityAttributes
);
```

Формат имения ящика

- Для открытия ящика, созданного на другом компьютере в сети, необходимо указать имя в формате

  \\ИмяКомпьютера\mailslot\[Путь] ИмяЯщика
- Можно открыть доменный ящик для передачи информации сразу всем компьютерам указанного домена. Для этого формируется имя

  \\ИмяДомена\mailslot\[Путь]ИмяЯщика
- Для передачи сообщения всем компьютерам первичного домена имя ящика задается в форме

  *\mailslot\[Путь] ИмяЯщика

Пример создания сервера

```
HANDLE hSlot = NULL;
hSlot = CreateMailslot
    ("\\\\computername\\mailslot\\messngr", 0,
    MAILSLOT_WAIT_FOREVER, NULL);

if (hSlot != INVALID_HANDLE_VALUE)
{
    char buffer[255]; DWORD nBytesRead;
    ReadFile(hSlot, &buffer, 255, &nBytesRead, NULL);
    …
}
```

Создание клиента почтового ящика

```
HANDLE hSlot =
    CreateFile(("\\\\computername\\mailslot\\messngr",
    GENERIC_WRITE, FILE_SHARE_READ, NULL,
    OPEN_EXISTING, FILE_ATTRIBUTE_NORMAL,
    NULL);

if (hSlot != INVALID_HANDLE_VALUE)
{
    char buf = "From\0\To\0Message\0";
    uint cb = sizeof(buf);
    WriteFile(hSlot, buf, cb, &cb, NULL);
    …
}
```

Использование mailslot

В **MSDN** написано, что если клиент открывает слот прежде чем слот был создан сервером, то он получит **INVALID_HANDLE_VALUE**

Использование mailslot

- Использование мэйлслотов особенно удобно в системах такого рода, работающих в пределах локальной сети. Программа-сервер создает на своем компьютере мэйлслот с именем, известным всем клиентам (например BWCronServerMailSlot). Программа-клиент производит соединение с сервером путем создания клиентского мэйлслота (BWCronClientMailSlot) и проверки существования сервера. Такую проверку можно произвести путем посылки серверу контрольного сообщения. Сервер дает добро на подключение либо отклоняет запрос. Дальнейший обмен данными будет производиться путем посылки серверу сообщений, и получения ответов от сервера через клиентский мэйлслот.
- Мэйлслоты кроме систем "клиент-сервер" можно использовать, например, для определения, запущена ли еще одна копия программы где-либо в локальной сети. Это делается посылкой сообщения всем компьютерам в заданном домене (второй сервер прикидывается клиентом и пытается установить связь с сервером. Если связь установлена, то работу не продолжаем, а если нет, то можно самому работать сервером).

Получение информации о почтовом ящике

```
BOOL GetMailslotInfo (
    HANDLE hMailslot, // указатель на слот
    LPDWORD lpMaxMessageSize, // максимальный размер
    LPDWORD lpNextSize, // размер следующего
    LPDWORD lpMessageCount, // количество сообщений
    LPDWORD lpReadTimeout // тайм аут
);
```

Изменение настроек почтового ящика

```
BOOL SetMailslotInfo(
    HANDLE hMailslot,
    DWORD lReadTimeout
);
```

Динамически компонуемые библиотеки (DLL)

- Если два приложения используют одну библиотеку, то они разделяют все глобальные переменные этой библиотеки. В действительности, глобальные переменные, как и вся библиотека, отображаются на адресные пространства разных процессов.
- Этот метод не привносит никакой новой функциональности по сравнению с отображением проецируемых файлов и, поэтому, его использование не рекомендуется.

Раздел с общими данными в DLL

```
#pragma data_seg(".shared")
//Общие данные
#pragma data_seg()
```

UsersDll.def:

```
LIBRARY "UsersDll" SECTIONS .shared
   READ WRITE SHARED
```

Удаленный вызов процедур (RPC - Remote Procedure Call)

- RPC (Remote Procedure Call) – это API, позволяющий приложению удаленно вызывать функции в других процессах как на своем, так и на удаленном компьютере.
- Предоставляемая Win32 API модель RPC совместима со спецификациями Distributed Computing Environment (DCE), разработанными Open Software Foundation. Это позволяет приложениям Win32 удаленно вызывать процедуры приложений, выполняющихся на других компьютерах под другими операционными системами. RPC обеспечивают автоматическое преобразование данных между различными аппаратными и программными архитектурами.

Сокеты (программные гнезда)

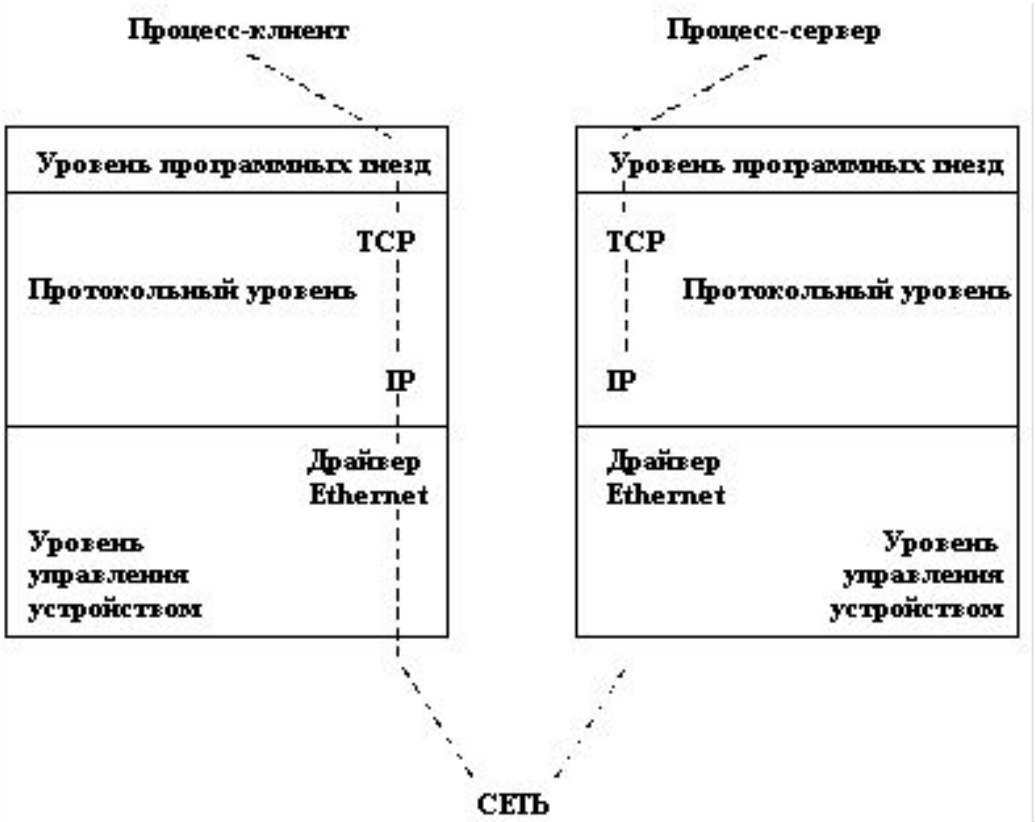


Рис. 2.3. Одна из возможных конфигураций программных гнезд

Сокеты (программные гнезда)

- Взаимодействие процессов на основе программных гнезд основано на модели "клиент-сервер". Процесс-сервер "слушает (listens)" свое программное гнездо, одну из конечных точек двунаправленного пути коммуникаций, а процесс-клиент пытается общаться с процессом-сервером через другое программное гнездо, являющееся второй конечной точкой коммуникационного пути и, возможно, располагающееся на другом компьютере. Ядро поддерживает внутренние соединения и маршрутизацию данных от клиента к серверу.
- Выделяются два типа программных гнезд - гнезда с виртуальным соединением (stream sockets) и датаграммные гнезда (datagram sockets). При использовании программных гнезд с виртуальным соединением обеспечивается передача данных от клиента к серверу в виде непрерывного потока байтов с гарантией доставки. При этом до начала передачи данных должно быть установлено соединение, которое поддерживается до конца коммуникационной сессии. Датаграммные программные гнезда не гарантируют абсолютной надежной, последовательной доставки сообщений и отсутствия дубликатов пакетов данных - датаграмм. Но для использования датаграммного режима не требуется предварительное дорогостоящее установление соединений, и поэтому этот режим во многих случаях является предпочтительным. Система по умолчанию сама обеспечивает подходящий протокол. Например, протокол TCP используется по умолчанию для виртуальных соединений, а протокол UDP - для датаграммного способа коммуникаций.