9 Средства взаимодействия процессов и потоков

9.1 Решения для межпроцессного взаимодействия (IPC)

Общая классификация IPC:

– ***сигнальные*** (***signal***): прерывания, сигналы, сообщения и т.п.

– ***канальные*** (***pipe***): трубопроводы и сокеты различного вида

– ***разделяемая память*** (***shared memory***)

– ***очереди сообщений*** (***message queue***, ***MQ***) – обладают свойствами сообщений и каналов

– средства ***синхронизации*** (***ISO***): семафоры, мьютексы, барьеры и т.п.

– комбинированные (комплексные) решения

Критерии сравнения и выбора:

– масштаб: межпоточные (локальные), межпроцессные, меж-системные, межплатформенные

– функциональность: синхронизация, взаимное исключение, обмен данными

– быстродействие (для синхронизации)

– производительность, пропускная способность (для данных)

– универсальность

– трудоемкость, уровень поддержки системой

– переносимость (в широком смысле)

и т.д.

9.2 Сигнальные IPC

Пригодны для извещения о событиях или условиях, а также для передачи простых («однозначных») команд. Передача существенных объемов данных затруднена или невозможна.

Прерывания (***interrupt***), ***исключения*** (***exception***) – аппаратный уровень, обычно недоступен для прикладных процессов и опосредуется операционной системой.

Сигналы (***signal***) – обладают минимальной информативностью: только тип (код, номер) сигнала и (обычно) идентификатор получателя. Фактически программная надстройка над прерываниями, но могут быть чисто программными.

Активное использование сигналов характерно для Unix-систем. В Windows поддержка ограниченная (функции стандартной библиотеки **`signal()`**, **`raise()`**), используются редко.

Сообщения (***message***) – более информативны: обладают дополнительными параметрами, что позволяет использовать их для передачи более сложной информации о событиях и в качестве опосредованных системой вызовов к другим процессам.

Рис. – передача сообщения и ответа

Рис. – блокировки при обмене сообщениями

В Windows – ***оконные сообщения*** (***window messages***), но с оговорками: window messages могут передаваться не только синхронно, но и через очередь (асинхронно).

В отличие от одиночных «синхронных» сообщений, ***очереди сообщений*** (***message queue***) предполагают асинхронное взаимодействие и пригодны для передачи потока данных, поэтому правильнее рассматривать их как средство межпроцессного обмена.

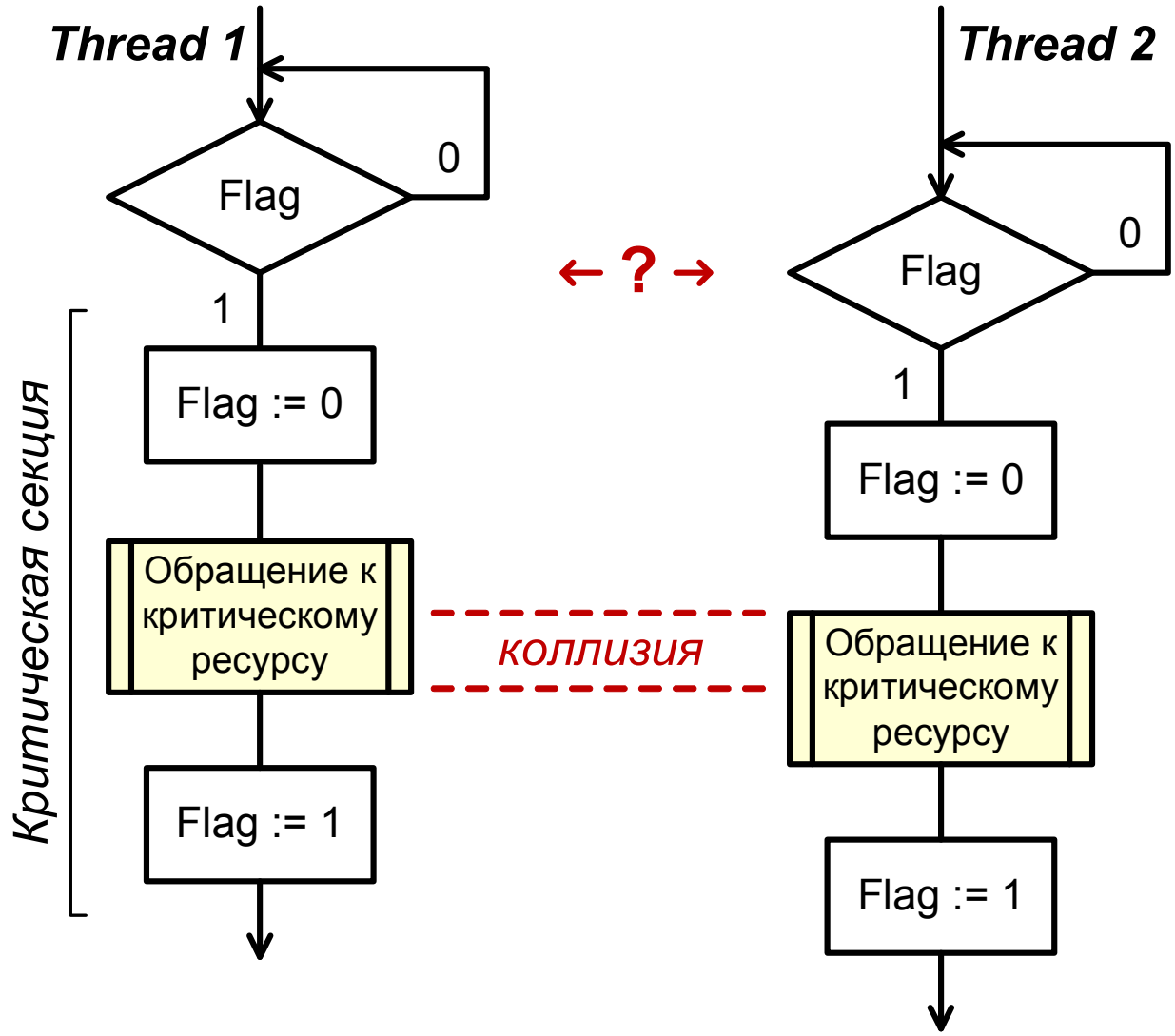
9.3 Средства синхронизации и взаимного исключения

Объекты ***ISO*** – ***Interprocess Synchronization Objects***, разновидность IPC-объектов.

Общая идея: проверка и модификация некоторого признака (флага) перед доступом к критическому ресурсу. Значение флага отражает свободное/занятое состояние ресурса.

Основная проблема: регулятор доступа к критическим ресурсам – тоже критический ресурс, обращения к нему – критическая секция. Рекурсивное замыкание требований.

Вывод: требование ***атомарности*** проверки и изменения условий входа в критическую секцию: они должны выполняться как единый примитив, не прерываемый другими процессами (потоками), претендующими на тот же критический ресурс.



Проблема атомарности входа в критическую секцию

Проблема: выполнение требования атомарности требует привилегированных операций управления аппаратными ресурсами, потенциально опасных и недоступных для непривилегированных процессов.

Решение – обеспечение атомарности средствами системы.

Альтернативное решение – самостоятельный запрет прерываний на время нахождения в критической секции. Однако инструкции управления прерываниями – обычно привилегированные, для процессов пользователя недоступны; кроме того, сложности создают вложенные блокировки.

9.3.1 Простейшие решения

Исторически наиболее ранние, основаны на использовании ***глобальных флагов*** с дополнительными мерами для обеспечения атомарности. Решение ***Деккера*** – начало 60-х гг.

Рис. – Блокировка явной проверкой переменной

Спин-блокировка (***spinlock***) – обеспеченное системой непрерываемое ожидание заданного значения переменной путем его проверки в цикле, в активном режиме, без перехода в состояние ожидания.
Достоинство – высокое быстродействие: выход из блокировки не связан с переключением контекста.
Недостаток – постоянная загрузка процессора.

Очевидно, для выхода из спин-блокировки необходимо, чтобы управляющая переменная были изменена «извне», независимо от заблокированного потока: в результате аппаратного собы-тия, более приоритетным потоком или потоком, выполняющим-ся на другом процессоре/ядре.

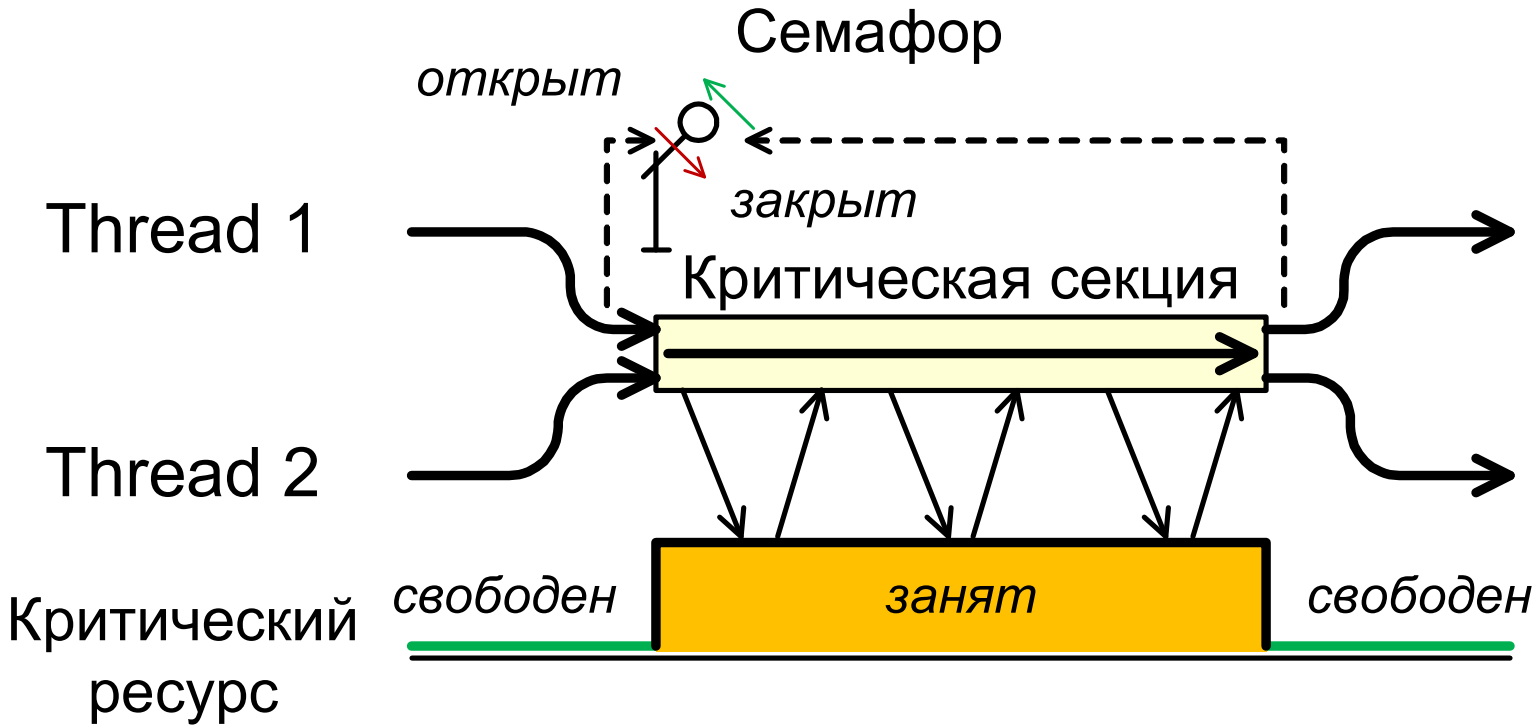
9.3.2 Семафоры, мьютексы, барьеры

В основе – глобальные переменные, но доступ к ним контролируется системой, что и обеспечивает ***атомарность*** выполняемых над объектами базовых операций (***примитивов***).

Два схожих вида объектов: семафоры и мьютексы.

Условно считается, что такой объект отражает состояние контролируемого критического ресурса, например его доступность или занятость. Решение задач синхронизации и взаимного исключения основано на добровольном следовании общему правилу: перед захватом критического ресурса (входом в критическую секцию) необходимо захватить защищающий их семафор (мьютекс). В случае его занятости поток будет либо заблокирован, либо извещен о неуспешности попытки.

Возможно и иное использование объектов, например для передачи сигнала.



Семафор (мьютекс) для управления доступом в критическую секцию

Семафор (***semaphore***) – глобальная переменная-счетчик ***S***, целочисленные неотрицательные значения, атомарно выполняемые примитивы для доступа ***P(S)*** и ***V(S)***:

P(S) – условный декремент: если значение достигло 0, то ожидание ненулевого значения

V(S) – безусловный инкремент счетчика.

Механизм предложен Э. Дейкстрой.
Терминология:
Sentinel – «часовой»
Passaren – «пропускать»
Vrygeven – «освобождать».

Существует много вариантов реализации семафоров. Пример: векторные «программируемые» семафоры System V IPC (среди операций есть также и ожидание обнуления счетчика.

Мьютекс (***mutex***, сокращение от ***mu***tual ***ex***clusion) – можно рассматривать как упрощенный двузначный семафор, состояние которого интерпретируют как «свободность» и «занятость», а примитивы доступа – «захват» и «освобождение». Попытка повторного захвата блокирует поток-инициатор до освобождения мьютекса другим потоком.

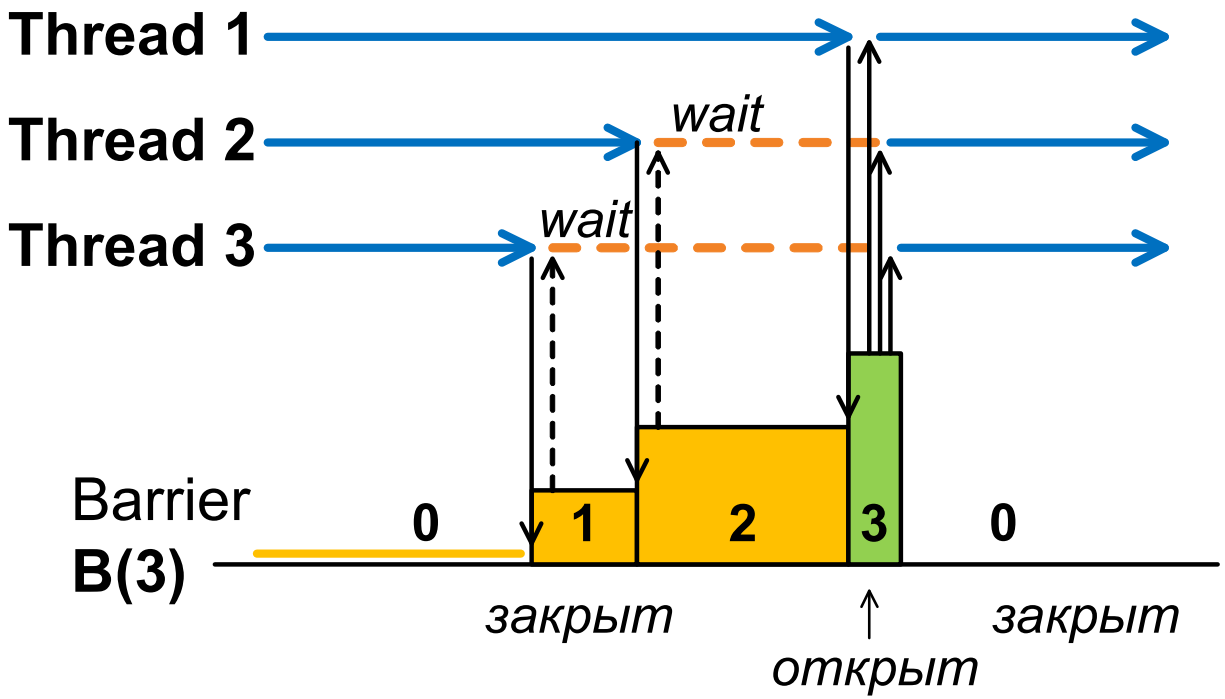
Потенциально мьютекс может быть реализован проще и эффективнее, особенно если ограничить его доступность пределами одного процесса.

Также может допускаться повторный (рекурсивный, «вложенный») захват мьютекса, но только одним и тем же потоком, что позволяет избежать самоблокировки. Такое поведение предполагает наличие внутреннего счетчика (т.е. подобие семафору, но с противоположной интерпретацией: ненулевое значение «счетчика захватов» соответствует блокировке мьюеткса).

Барьер (***barrier***) – объект ISO, обеспечивающий синхронизацию достигших его процессов (потоков): каждый из них посредством системного вызова запрашивает синхронизацию барьером и переводится в состояние ожидания до тех пор, пока количество таких запросов не достигнет заранее заданного. После этого все получают возможность выполняться дальше, а барьер «освобождается».

Возможная реализация барьера – счетчик запросов (например, доработанный семафор).

Пример использования: декомпозиция вычислительной задачи на несколько параллельных потоков и ожидание завершения их всех; синхронный запуск нескольких параллельных потоков после завершения предыдущих итераций в них.


Thread 1
Thread 2
Thread 3
wait
wait
Barrier
B(3)
0
1
2
3
0
закрыт
открыт
закрыт

Барьер

9.4 Средства синхронизации в Windows

9.4.1 «Блокированные» обращения

Потокобезопасная (атомарная) системная реализация некоторых операций над переменными. Возможно также использование для межпроцессного взаимодействия, но потребуется размещение переменной в глобальной (разделяемой) памяти. Поддерживаемый тип переменной – целое число «общего вида» или указатель.

```
InterlockedDecrement()
InterlockedIncrement()
InterlockedExchange()
InterlockedExchangeAdd()
InterlockedCompareExchange()
```

9.4.2 Механизм Critical Section

Реализация классической критической секции.
Фактически это упрощенный мьютекс, действующий в рамках одного процесса: вместо системного объекта используется структура **CRITICAL_SECTION** в адресном пространстве процесса пользователя. Структура создается произвольным образом, но требует отдельной инициализации и деинициализации.

Идентификация: указатель на структуру **CRITICAL_SECTION***.

Функции API для использования:

```
InitializeCriticalSection(CRITICAL_SECTION &cs);
DeleteCriticalSection(CRITICAL_SECTION &cs);
EnterCriticalSection(CRITICAL_SECTION &cs);
TryEnterCriticalSection(CRITICAL_SECTION &cs);
LeaveCriticalSection(CRITICAL_SECTION &cs);
```

Поддерживается повторное (***рекурсивное***) вхождение в критическую секцию («захват» секции), но только одним и тем же потоком. Количество «покиданий» секции должно соответствовать количеству вхождений, иначе секция останется заблокированной для других потоков.

Внутренняя реализация – счетчик, подобный семафору, но с «обратной» логикой: секция считается свободной и доступной для входа при нулевом значении счетчика, занятой – при ненулевом. Этот же счетчик отражает «глубину» блокировки секции при рекурсивных захватах.

9.4.3 Функции ожидания WaitFor…

Функции, применимые к любым системным объектам, для которых определено свойство (флаг) «***signaled***». Так, кроме специализированных объектов синхронизации это могут быть процессы, потоки, файлы и проч.

Типичная интерпретация состояния «signaled»:

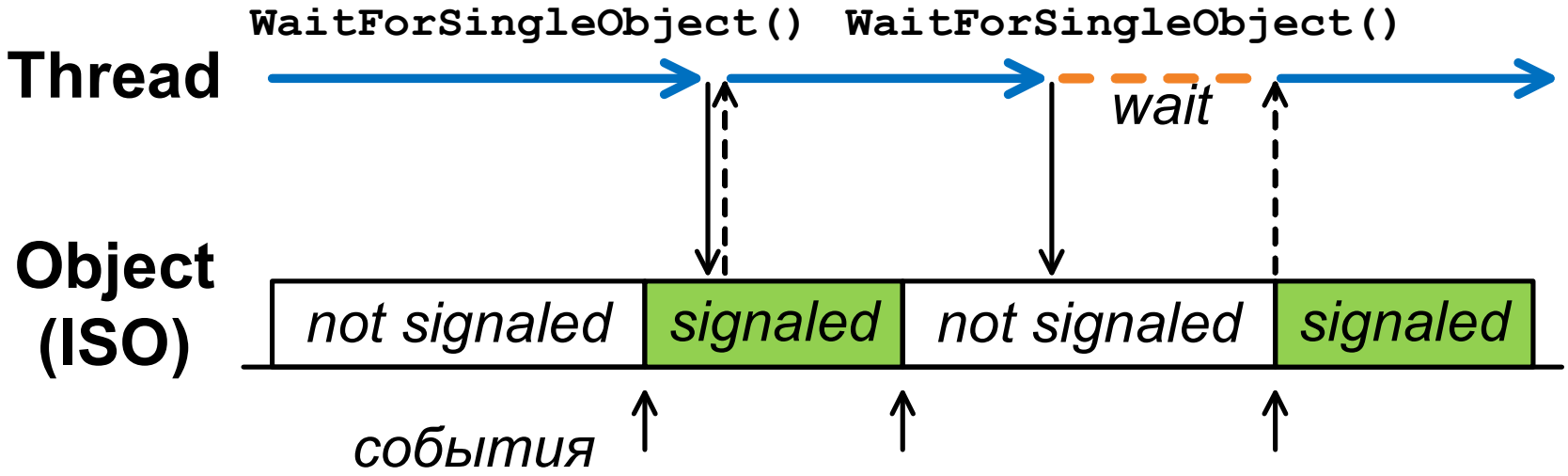
– Файлы, каналы, устройства ввода-вывода: завершение операции ввода-вывода

– Процессы, потоки: завершение (переход в состояние «зомби») либо прекращение действия дескриптора (Handle)

– Объекты синхронизации – в соответствии с логикой конкретного вида объекта.

Ожидание единственного объекта:

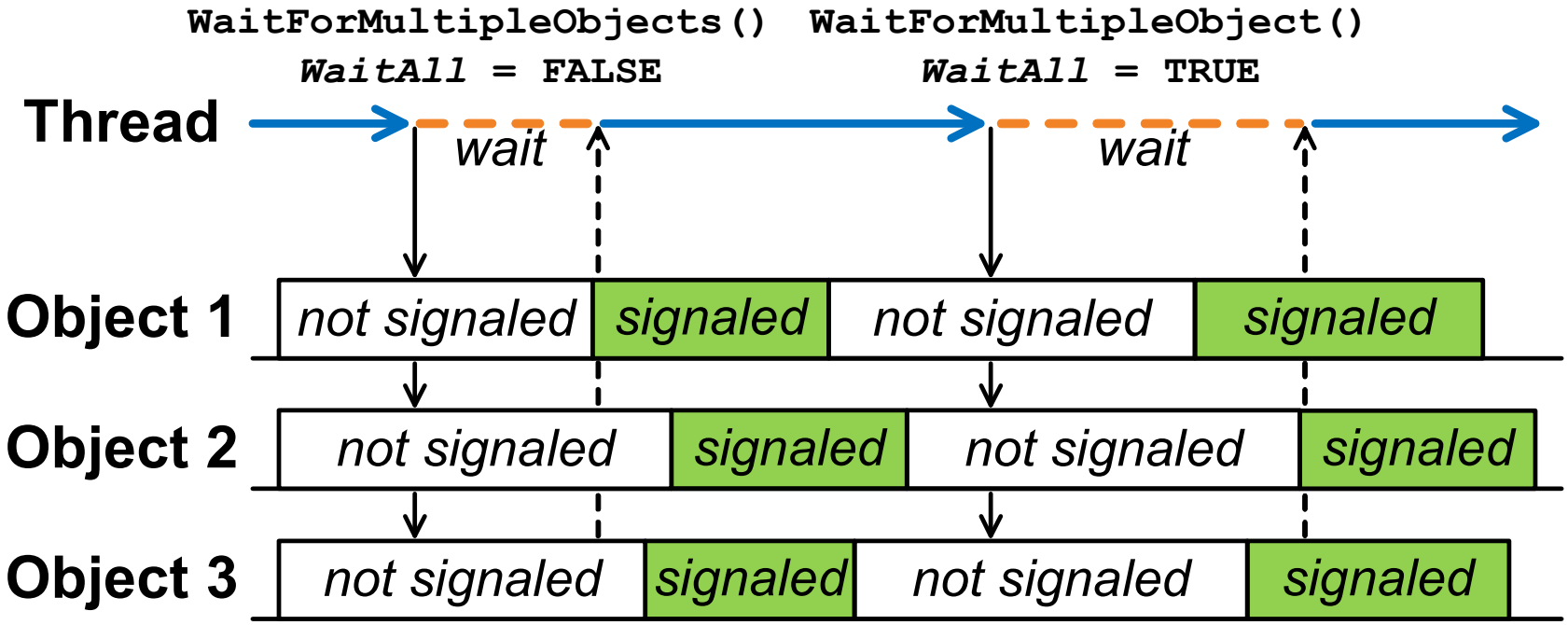
```
WaitForSingleObject()
WaitForSingleObjectEx()
```



Ожидание одного объектов

Ожидание одного или всех (в зависимости от флага) объектов из списка

`WaitForMultipleObjects()`
`WaitForMultipleObjectsEx()`



Ожидание нескольких объектов

Перевод в состояние «signaled» одного объекта и ожидание другого:

`SignalObjectAndWait()`

Рис. – Использование двух объектов синхронизации

Ожидание объектов, прерываемое событиями (сообщениями):

`MsgWaitForMultipleObjects()`
`MsgWaitForMultipleObjectsEx()`

Ожидание в системном потоке из пула, с возвратом управления в виде «обратного» вызова (callback):

`RegisterWaitForSingleObject()`
`UnregisterWaitEx()`

9.4.4 Объект Mutex

Системный ISO-объект, реализация классического мьютекса.

Идентификация: уникальное имя (необязательное), дескриптор (Handle). Может использоваться как в рамках одного процесса, так и потоками разных процессов.

Состояния:
– «signaled» – мьютекс «свободен»
– «not signaled» – мьютекс «занят/захвачен»

Допускает повторный (рекурсивный) захват, но только одним и тем же потоком. Количество освобождений должно соответствовать количеству захватов.

Если ожиданием мьютекса заблокировано несколько потоков, то право выполняться, захватив освободившийся мьютекс, получит только один из них, прочие продолжат ожидание.

Создание объекта «мьютекс»:

`CreateMutex()`

Открытие объекта по его имени:

`OpenMutex()`

Отказ от объекта:

`CloseHandle()`

Освобождение захваченного мьютекса:

`ReleaseMutex()`

Захват мьютекса (блокирующий) – одна из функций ожидания

9.4.5 Объект Semaphore

Системный ISO-объект, соответствующий классическому семафору.

Идентификация: уникальное имя (необязательное), дескриптор (Handle). Может использоваться как в рамках одного процесса, так и потоками разных процессов.

Состояния:
– «signaled» – значение семафора больше 0
– «not signaled» – значение семафора равно 0

Реализация: переменная-счетчик с неотрицательными значениями. Повторный (рекурсивный) захват не актуален – блокировка зависит только от текущего значения счетчика семафора.

Создание объекта «семафор»:

`CreateSemaphore()`

Открытие существующего объекта по имени:

`OpenSemaphore()`

Отказ от объекта:

`CloseHandle()`

Безусловный инкремент счетчика семафора V(S):

`ReleaseSemaphore()`

Условный (блокирующий) декремент семафора P(S) – одна из функций ожидания.

9.4.6 Объект Event

Системный ISO-объект – программное отражение «события».

Идентификация: уникальное имя (необязательное), дескриптор (Handle). Может использоваться как в рамках одного процесса, так и потоками разных процессов.

Состояния:
– «signaled» – событие наступило и еще не обработано
– «not signaled» – событие не наступило или уже обработано

Особенность по сравнению с мьютексами и семафорами: событие не «захватывается» потоком, а используется как сигнал для блокировки и пробуждения, причем синхронизировать можно сразу несколько потоков.

Типы «событий»:

– с ***автоматическим*** сбросом (***auto reset***) – состояние «signaled» сбрасывается функцией ожидания автоматически перед возвратом управления (следовательно, разблокирован будет только один из ожидавших события потоков)

– с ***ручным*** сбросом (***manual reset***) – состояние «signaled» сохранятся до явного сброса соответствующей функцией (следовательно, разблокированы будут все потоки, синхронизирующиеся этим событием).

Создание объекта «событие»:

```
HANDLE CreateEvent(
  LPSECURITY_ATTRIBUTES lpEventAttributes,
  BOOL bManualReset, BOOL bInitialState, LPTSTR
lpName
)
```

Установка состояния объекта в «signaled»:

```
BOOL SetEvent( HANDLE hEvent)
```

Явный сброс состояния в «not signaled»:

```
BOOL ResetEvent( HANDLE hEvent)
```

Временная установка в «signaled» до срабатывания всех функций ожидания, относящихся к указанному объекту Event, и затем сброс его состояния в «not signaled»:

BOOL PulseEvent(HANDLE *hEvent*)

Последняя функция считается ненадежной: она не влияет на потоки, которые в этот момент не заблокированы ожиданием, и работает полноценно с объектами Event типа manual-reset. Для типа auto-reset активизируется только один ожидающий поток, после чего состояние объекта сбрасывается автоматически.

Рис. – логика работы Event

9.4.7 Объект Waitable Timer

Системный ISO-объект – таймер с несколькими параметрами и несколькими путями использования.
Идентификация: уникальное имя (необязательное), дескриптор (Handle). Может использоваться как в рамках одного процесса, так и потоками разных процессов.

Типы таймеров:

– ***manual-reset*** – сброс таймера путем новой установки величины задержки явным вызовом функцией **`SetWaitableTimer()`**

– ***synchronization*** – сброс таймера автоматически при завершении Wait-функции

– ***periodic*** manual reset, ***periodic*** synchronization – аналогично, но перевод таймера в состояние «signaled» повторяется периодически через заданные интервалы времени

Создание таймера, подключение к существующему, отказ:

```
CreateWaitableTimer()
CreateWaitableTimerEx()

OpenWaitableTimer()

CloseHandle()
```

Настройка и запуск таймера:

```
BOOL SetWaitableTimer( HANDLE hTimer,
  const LARGE_INTEGER *lpDueTime, LONG lPeriod,
  PTIMERAPCROUTINE pfnCompletionRoutine,
  void* pArg,
  BOOL fResume
)
```

Остановка таймера:

```
BOOL CancelWaitableTimer( HANDLE hTimer)
```

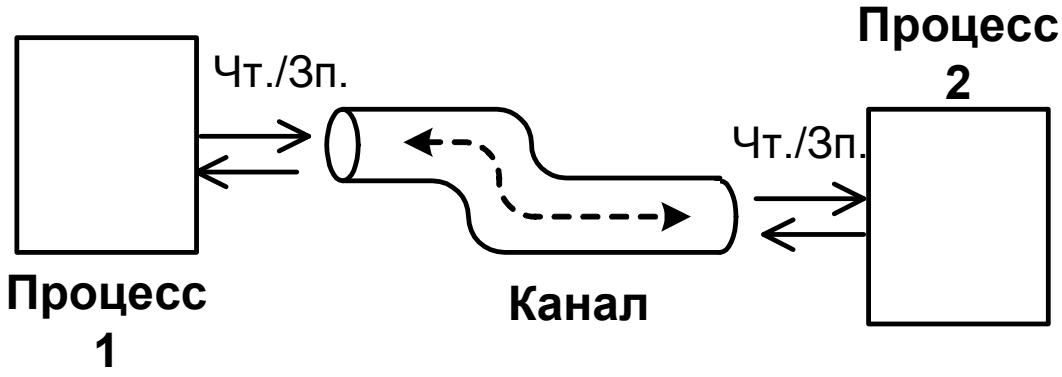
9.5 Средства обмена данными и совместного доступа к ним

9.5.1 Канальные IPC: каналы и «почтовые ящики»

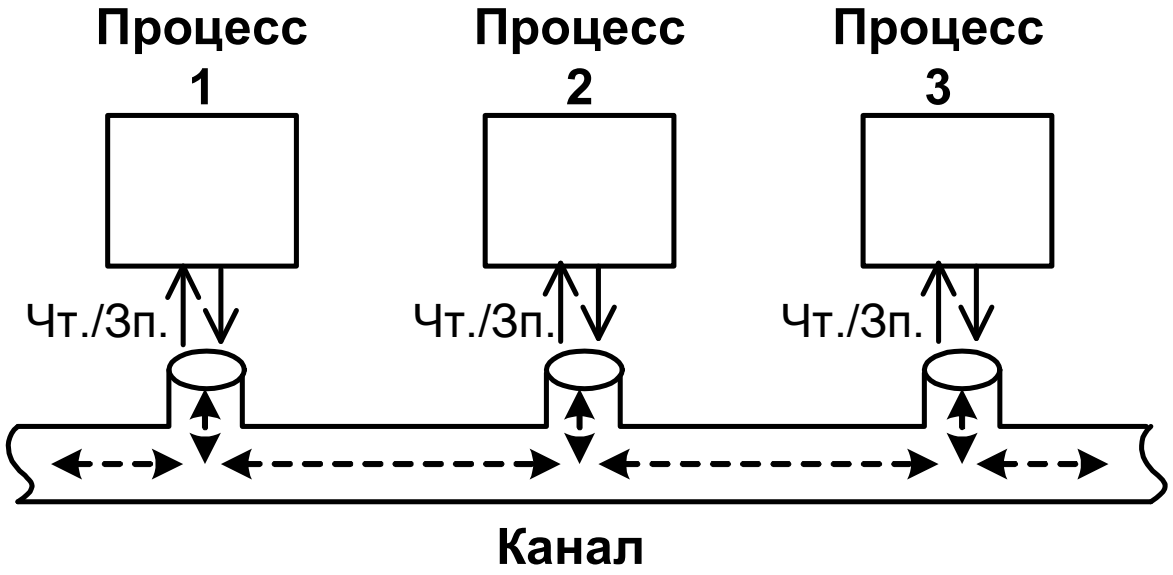
Каналы («***трубопроводы***», ***транспортеры***, ***pipe***, ***FIFO***) – объекты, унифицированные с файлами по идентификации (дескрипторы, Handle) и методам доступа. Традиционно неименованный канал – «***pipe***», именованный – «***FIFO***».

Неименованный канал не может быть открыт посторонним процессом, поэтому он может использоваться только между процессами-«родственниками» (благодаря наследованию дескрипторов родительского процесса).

Именованный канал – полноценный объект файловой системы, имеющий физическое место для хранения данных, доступен потенциально всем процессам в системе.



Канал: взаимодействие двух процессов



Канал: взаимодействие более чем двух процессов

Основное отличие каналов от обычных файлов – доступ только последовательный, причем чтение – разрушающее (с изъятием прочитанных данных).

Обработка ситуаций доступа (типично):

– Чтение непустого канала – считывание порции данных

– Чтение пустого канала – блокировка (ожидание)

– Чтение пустого канала без «писателей» – ошибка (для pipe)

– Запись в канал – запись порции данных или ожидание свободного места

– Запись в канал без «читателей» – ошибка (для pipe)

Поддержка каналов того или иного вида присутствует практически во всех ОС.

Каналы могут обеспечить соблюдение только общего порядка следования байт, но не их структурирование, целостность и адресную доставку: параллельная запись данных в канал может приводить к их смешиванию, параллельное чтение – к считыванию невалидных фрагментов. Встроенных средств предотвращения коллизий, как правило, нет.

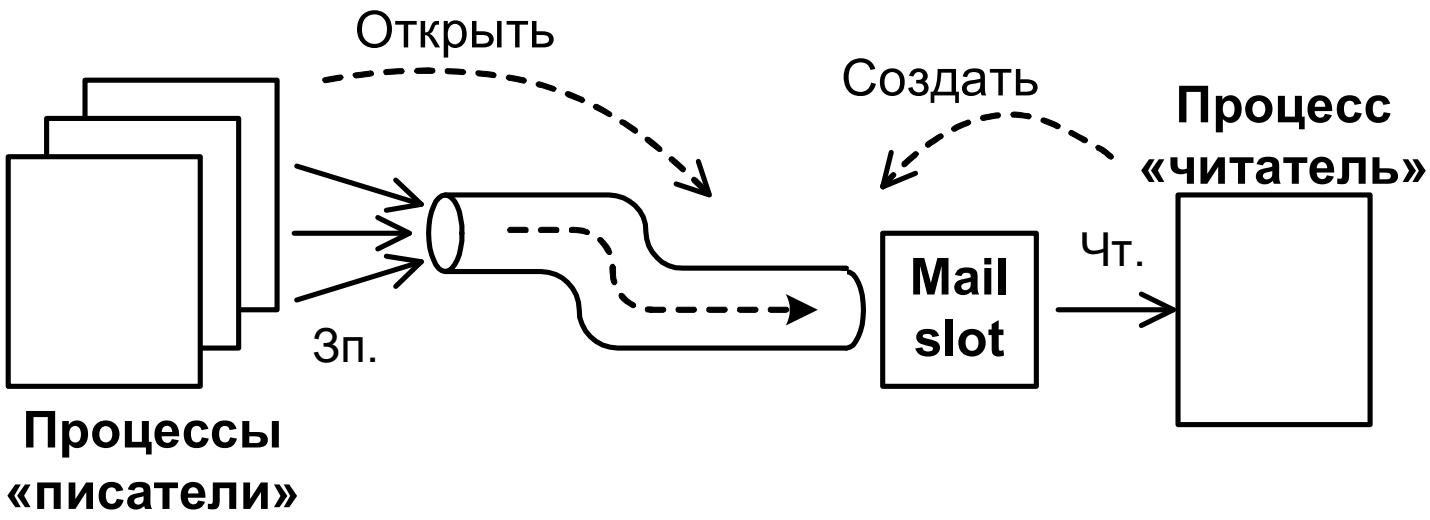
Возможные решения:

– Ограничение числа участников взаимодействия до двух

– Отказ от двунапрвленного (дуплексного) обмена в пользу только однонаправленного

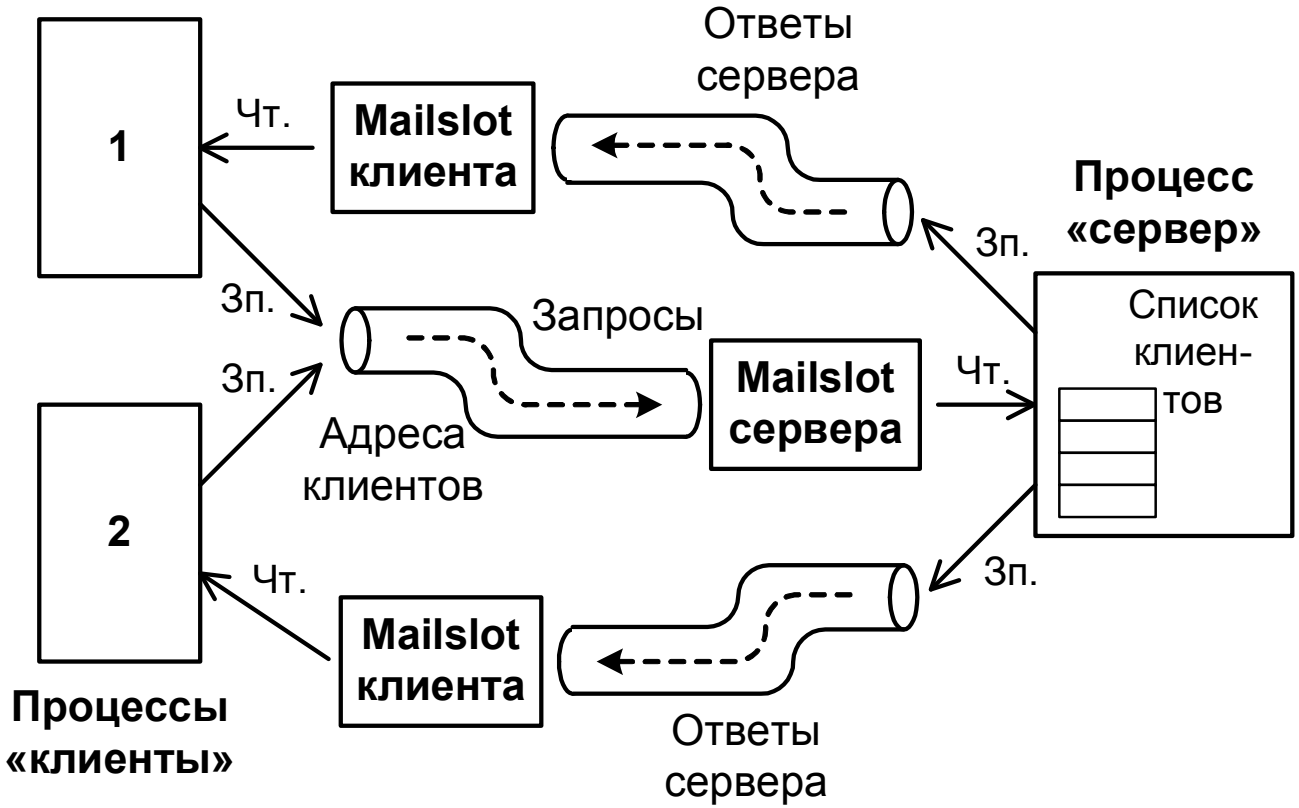
– Использование дополнительных средств синхронизации для упорядочивания доступа.

Почтовый ящик (***mailslot***) – отличается от канала несиммет-ричностью: передача данных строго однонаправленная. Это ослабляет проблемы как нарушения целостности данных (по крайней мере, «читатель» у ящика будет единственный), так и идентификации (ящик однозначно связан с этом единственным конкретным «читателем»).



Почтовый ящик: однонаправленный обмен

Двунаправленный обмен реализуется двумя однонаправленными соединениями (и двумя почтовыми ящиками).



Почтовые ящики: двунаправленный обмен с несколькими клиентами

9.5.2 Канальные IPC в Windows

В ОС Windows канальные IPC представлены неименованными и именованными каналами и «почтовыми ящиками».

Win API: неименованный канал (pipe):

Идентификация канала: только Handle, имени нет.

Создание канала:

```
BOOL CreatePipe(
  HANDLE* phReadPipe, HANDLE* phWritePipe,
  LPSECURITY_ATTRIBUTES pSecAttr,
  DWORD nSize
)
```

Доступ к существующему каналу: наследование дескрипторов.

Отказ от дескриптора доступа к каналу: **`CloseHandle( hPipe)`**

Особенности:

– Создание двух отдельных дескрипторов: чтения и записи, любой из которых может быть закрыт независимо от другого

– Неименованный канал может существовать только между процессами-«родственниками»

– После отключения последнего пользователя такой канал полностью удаляется, а остававшиеся в нем данные теряются.

Win API: именованный канал (FIFO):

Идентификация: дескрипторы (Handle) для открытого канала, глобальные имена объектов-FIFO в файловой системе:

\\.\pipe*pipename* , \\.\pipe\LOCAL*pipename*

Создание нового канала:

```
HANDLE CreateNamedPipe(
  LPCSTR lpName,
  DWORD dwOpenMode, DWORD dwPipeMode,
  DWORD nMaxInstances,
  DWORD nOutBufferSize, DWORD nInBufferSize,
  DWORD nDefaultTimeOut,
  LPSECURITY_ATTRIBUTES pSecAttr
)
```

Доступ к существующему каналу: **CreateFile()**, наследование

Отказ от дескриптора доступа к каналу: **CloseHandle(*hPipe*)**

Особенности именованных каналов Windows:

– Несимметричность: выделяются клиент и сервер

– Наличие нескольких «экземпляров» открытого канала для использования множеством клиентов

– После закрытия «экземпляра» канала конкретного клиента непрочитанные данные теряются

– Обращение с дескрипторами, удаление объекта из файловой системы – аналогично прочим файлам

– Дескрипторы открытого именованного канала могут быть одно- или двунаправленными в зависимости от режима

– Два типа каналов: «byte-type» и «message-type» (задается флагиами параметра **`dwPipeMode`**). Возможности «канала сообщений» приближены к задачам взаимодействия «клиент-сервер» и «пакетным» сокетам (функции **`CallNamedPipe()`**, **`TransactNamedPipe()`** и др.)

– Дополнительные возможности управления каналом, например неразрушающее чтение (функция **`PeekNamedPipe()`**), обратная связь (задержка выполнения **`FlushFileBuffers()`** до считывания ранее записанных данных), и т.д.

Win API: почтовый ящик (Mailslot):

Создание «сервером» нового «почтового ящика»:

```
HANDLE CreateMailslot( LPCSTR lpName,
  DWORD nMaxMessageSize,
  DWORD lReadTimeout,
  LPSECURITY_ATTRIBUTES pSecAttr
)
```

Подключение «клиента» к существующему «почтовому ящику»:

```
HANDLE CreateFile()
```

Имена объектов – почтовых ящиков – в файловой системе:

```
\\.\mailslot\slotname
\\computername\mailslot\slotname
\\domainname\mailslot\slotname
\\*\mailslot\slotname
```

Особенности «почтовых ящиков»:

– Владельцем «почтового ящика» становится создавший его процесс, ему «ящик» доступен только для чтения, прочим процессам – только для записи

– Почтовый ящик может быть открыт не только локально, но и удаленно – на другом узле сети, через стек сетевых протоколов (побочный эффект – наличие «запрещенных» значений размера записываемого в него блока данных).

9.5.3 Разделяемая память

Разделяемая память (***shared memory***) – обеспечение совместного доступа к данным в специальным образом оформленной области памяти.
Наиболее производительный метод, эффективный при любых объемах данных.

Подробно рассматривалось в теме управления памятью. Рекомендуемая в Windows реализация – ***отображение файлов*** в память.

Типичная проблема – коллизии параллельного доступа к ячейкам памяти. Корректно разрешаются на физическом уровне, однако нет защиты от потери целостности и корректности содержимого на логическом уровне.
Решение – использование дополнительных средств синхронизации и взаимного исключения: семафоров, мьютексов и т.п.

9.5.4 Очереди сообщений

Очереди сообщений (***message queue***, ***MQ***) – асинхронный механизм взаимодействия, совмещающий характеристики каналов и сообщений: асинхронная (с буферизацией в очереди) передача потока данных в виде законченных блоков, для каждого из которых обеспечены тип, порядок в очереди и целостность. Эффективны для взаимодействия «сервис-сервис».

Существует множество реализаций различного уровня сложности, функциональности и пр.:
– Microsoft MQ (MSMQ) – начиная с Win 95 и Win NT 4.0 SP3
– IBM MQ и IBM MQ Interface (MQI)
– Amazon Simple Queue Service
– Java Message Service
– очереди в составе System V IPC и POSIX IPC
и т.д.

9.5.5 Другие средства обмена данными в Windows

Например:

– Оконное сообщение **`WM_COPYDATA`**

– ***Буфер обмена*** (***clipboard***)

– Глобальные данные модулей ***DLL***

– ***Удаленный вызов процедур*** (***Remote Procedure Call***, ***RPC***)

и т.д.

9.6 Другие решения. Комплексные механизмы

9.6.1 Сокеты

Сокеты (socket) – механизм комплексного решения задач взаимодействия, наиболее близкий к каналам, но имеющий много разновидностей. Подробно рассматривать его имеет смысл в рамках сетевого программирования (хотя существуют и используются также и локальные сокеты).

9.6.2 Монитор

Монитор Хоара – исчерпывающий процедурный интерфейс для работы с общими данными, содержащий все необходимые средства синхронизации. Таким образом, осуществляется скрытие критических ресурсов от непосредственного доступа со стороны прикладных программ. На практике монитор может оказаться недостаточно гибким и эффективным; в частности, он не устраняет полностью риск тупиков, но зато хорошо согласуется с объектно-ориентированной моделью.

9.6.3 Рандеву

Основная концепция – обеспечение непосредственного взаимодействия между процессами, например через доступ к их памяти. Для этого необходимо привести оба процесса в состояние готовности к взаимодействию – многошаговая процедура:

1) заявить о намерении взаимодействовать (назначить рандеву)

2) подтвердить согласие на взаимодействие

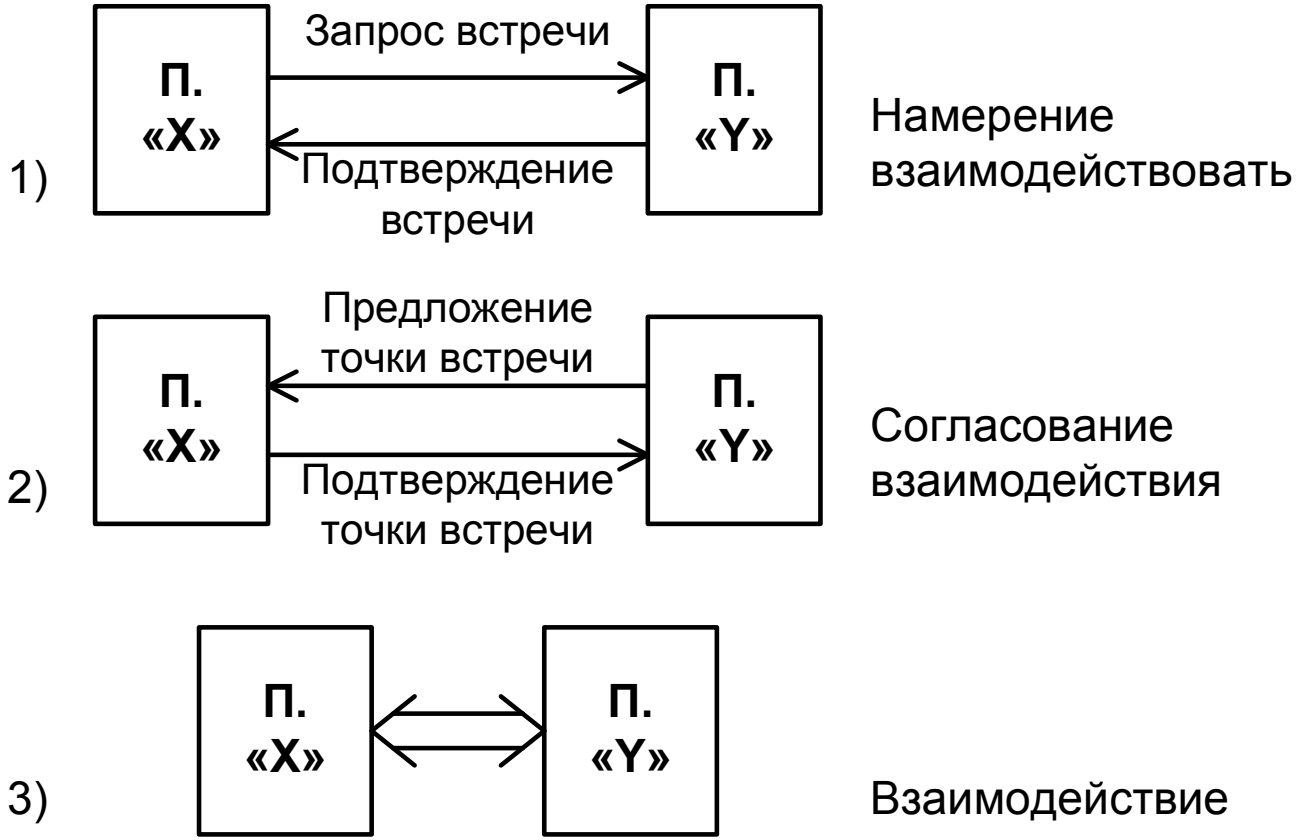
3) объявить точку взаимодействия

4) подтвердить точку взаимодействия

После выполнения всех шагов процессы могут приступить к взаимодействию.

Механизм рандеву ресурсоемкий, обычно реализуется с использованием вспомогательных процессов.

Симметричное рандеву (Хоар, 1978 г.) – процессы-участники равноправны, направленность процедуры при каждом взаимодействии зависит от конкретной решаемой задачи.



Взаимодействие посредством симметричного рандеву

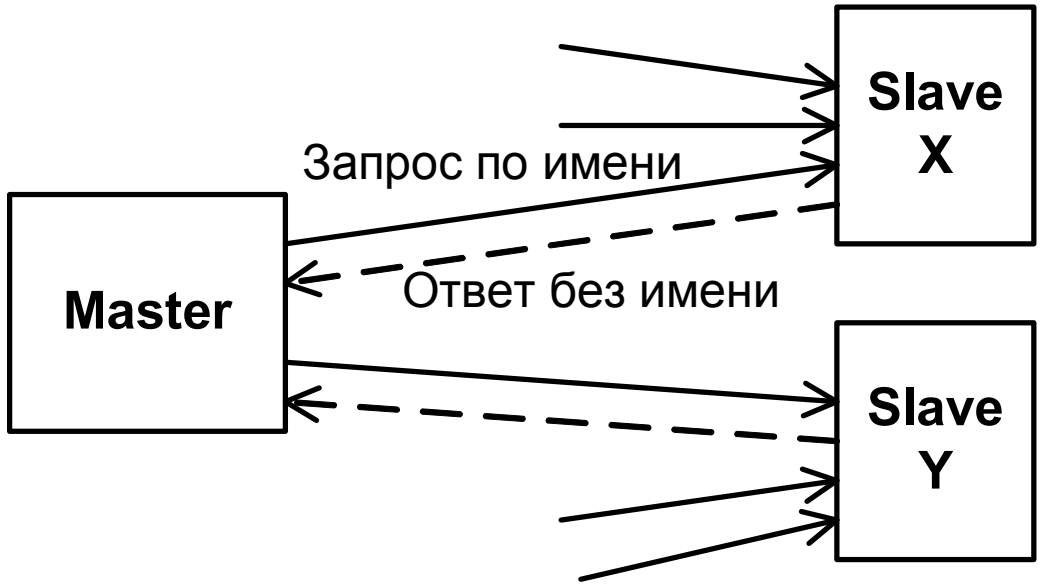
Операторы Хоара для обмена данными посредством рандеву:
«**!**» – передача (вывод) данных
«**?**» – прием (ввод) данных

Проблема идентификации участников взаимодействия – все процессы-участники должны быть известны друг другу, но набор процессов, особенно пользовательских, может быть заранее не известен.

Асимметричное рандеву (Хансен) – роли участников заранее распределены:

– «***хозяин***» (***master***) – посылает команды (запросы) и принимает результаты (ответы)

– «***слуга***» (***slave***) – принимает запросы, исполняет их и возвращает результаты



Взаимодействие посредством асимметричного рандеву

Принципиальное решение для асимметричного рандеву: каждый процесс-«слуга» соответствует определенной функции (группе функций), и вызов функции выполняется через обращение к «слуге». Слуги идентифицируются именами (фактически это имена функций), обращение к «слуге» – по имени. Состав «слуг» заранее известен и ограничен, их имена могут быть описаны и сделаны доступными прочим процессам в одностороннем порядке. Однако именование «хозяев» не требуется: «слуга» обращается к «хозяину», не используя имени, а лишь возвращая ответ источнику запроса.

Асимметричное рандеву снимает проблему взаимной идентификации процессов, но не ресурсоемкости механизма. Эффективная реализация требует поддержки на аппаратном уровне (например, процессор Intel iAPX432 для языка Ada).

9.6.4 Схемы взаимодействия «клиент-сервер», «агент-менеджер»

Развитие идеи асимметричного рандеву.

«***Клиент-сервер***» (***Client-Server***) – клиент обращается к серверу с запросом и получает ответ на него.

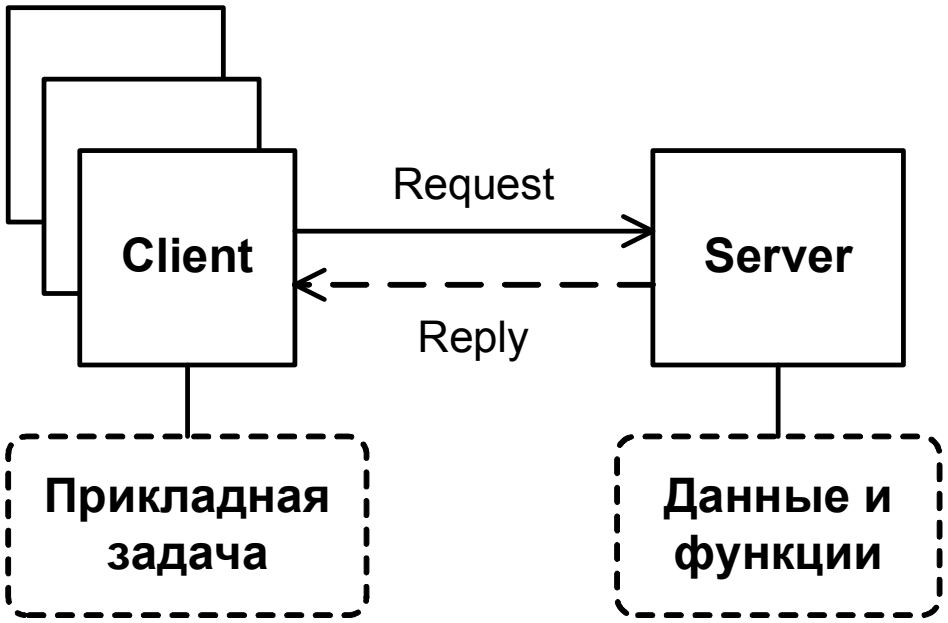


Схема взаимодействия «клиент-сервер»

Подразумевается, что серверов в системе меньше, чем клиентов, но клиент может обращаться также и к нескольким серверам.

Пример: клиент-серверные СУБД.

«***Агент-менеджер***» (***Agent-Manager***) – «менеджер» рассылает множеству «агентов» запросы, собирает ответы и формирует общее управление подконтрольной системой (собственно управление реализуется также через агентов, посредством рассылаемых им команд).

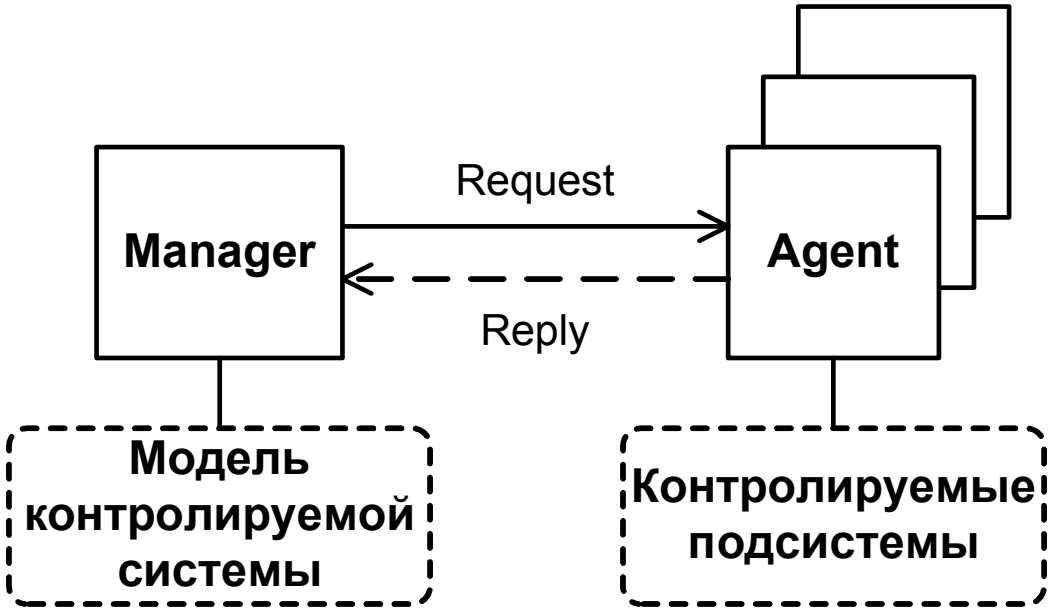


Схема взаимодействия «агент-менеджер»

Подразумевается, что агентов больше, чем менеджеров, но один агент может работать также и на нескольких менеджеров.

Примеры: управляющий сетевой протокол и служба SNMP с базой MIB.