Синхронизация процессов и потоков. Межпроцессное взаимодействие

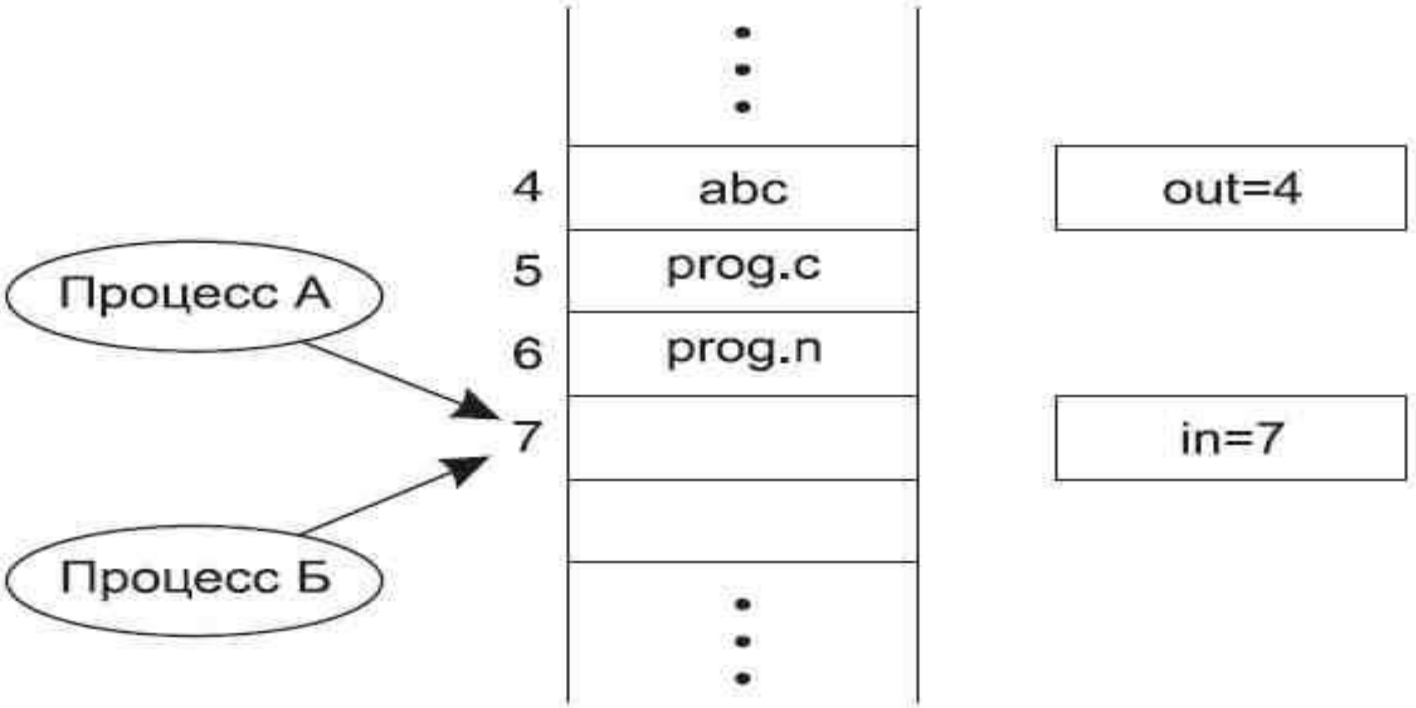
Проблема соревнования

Важной проблемой является обеспечение совместной работы процессов без создания взаимных помех, когда, к примеру, два процесса одновременно пытаются получить доступ к ресурсу. Другая проблема состоит в определении правильной последовательности действий: если процесс *А* вводит данные, а процесс *Б* их обрабатывает, то процесс *Б* перед обработкой должен подождать, пока процесс *А* не введет нужные данные.

Представим, что в системе существует каталог файлов, которые нужно вывести на печать. Когда процессу необходимо распечатать какой-нибудь файл, он помещает имя этого файла в этот каталог файлов. Процесс, связанный с принтером, периодически проверяет наличие файлов для печати и в том случае, если такие файлы имеются, распечатывает их и удаляет их имена из каталога.

В нашем каталоге имеется большое количество областей памяти с номерами 0, 1, 2..., в каждой из которых может храниться имя файла. Также представьте, что есть две общие переменные: *out*, указывающая на следующий файл, предназначенный для печати, и *in*, указывающая на следующую свободную область в каталоге.

В какой-то момент времени области от 0 до 3 пустуют (файлы уже распечатаны). Почти одновременно процессы *А* и *Б* решают, что им нужно поставить файл в очередь на печать. Эта ситуация показана на рисунке.



Процесс *А* считывает значение переменной *in* и сохраняет значение 7 в локальной переменной по имени *next_free_slot* (следующая свободная область). Сразу же после этого центральный процессор решает, что процесс *А* проработал достаточно долго, и переключается на выполнение процесса *Б*. Процесс *Б* также считывает значение переменной *in* и также получает число 7. Он также сохраняет его в своей локальной переменной *next_free_slot*. К текущему моменту оба процесса полагают, что следующей доступной областью будет 7.

Процесс *Б* продолжает выполняться. Он сохраняет имя своего файла в области 7 и присваивает переменной *in* обновленное значение 8. Затем он переходит к выполнению каких-нибудь других действий. Через некоторое время выполнение процесса *А* возобновляется с того места, где он был остановлен. Он считывает значение переменной *next_free_slot*, видит там число 7 и записывает имя своего файла в область 7, затирая то имя файла, которое только что было в него помещено процессом *Б*. Затем он вычисляет *next_free_slot* + 1, получает значение 8 и присваивает его переменной *in*. В результате процесс *Б* никогда не получит вывода на печать.

Подобная ситуация, когда два или более процесса считывают или записывают какие-нибудь общие данные, а окончательный результат зависит от того, какой процесс и когда именно выполняется, называется **проблемой соревнования**. Результаты большинства прогонов программы могут быть вполне приемлемыми, но до тех пор, пока не случится описанная ситуация. К сожалению, с ростом параллелизма из-за все большего количества ядер проблемы соревнования встречаются все чаще.

Критические области

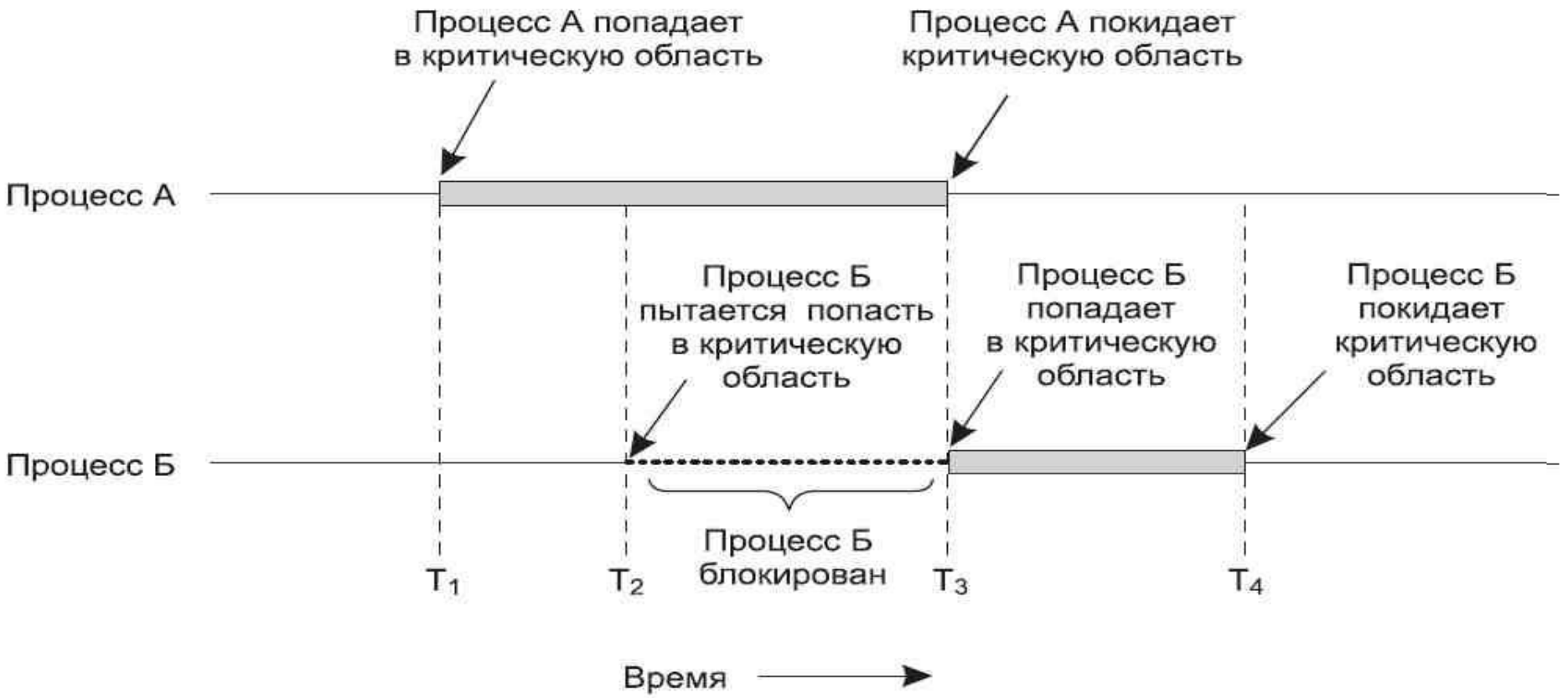
Ключом к предупреждению проблемы соревнования в этой и во многих других ситуациях использования общей памяти, общих файлов и вообще чего-нибудь общего может послужить определение способа, при котором в каждый конкретный момент времени доступ к общим данным для чтения и записи может получить только один процесс. Иными словами, нужен способ **взаимного исключения**, то есть некий способ, обеспечивающий правило, при котором если общие данные или файл используются одним процессом, возможность их использования всеми другими процессами исключается. Описанные выше трудности произошли благодаря тому, что процесс *Б* стал использовать общие переменные еще до того, как процесс *А* завершил работу с ними.

Проблемы обхода состязательных ситуаций могут быть сформулированы также в абстрактной форме. Какую-то часть времени процесс занят внутренними вычислениями и чем-нибудь другим, не создающим соревновательных ситуаций. Но иногда он вынужден обращаться к общей памяти или файлам либо совершать какие-нибудь другие значимые действия, приводящие к соревнованиям. Та часть программы, в которой используется доступ к общей памяти, называется **критической областью** или **критической секцией**. Если бы удалось все выстроить таким образом, чтобы никакие два процесса не находились одновременно в своих критических областях, это позволило бы избежать соревнования.

Хотя выполнение этого требования позволяет избежать соревновательных ситуаций, его недостаточно для того, чтобы параллельные процессы правильно выстраивали совместную работу и эффективно использовали общие данные. Для приемлемого решения необходимо соблюдение четырех условий:

a) Два процесса не могут одновременно находиться в своих критических областях.
b) Не должны выстраиваться никакие предположения по поводу скорости или количества центральных процессоров.
c) Никакие процессы, выполняемые за пределами своих критических областей, не могут блокироваться любым другим процессом.
d) Процессы не должны находиться в вечном ожидании входа в свои критические области.

В абстрактном смысле необходимое нам поведение показано на рисунке.



Процесс *А* входит в свою критическую область во время *Т*1. Когда наступает время *Т*2, процесс *Б* пытается войти в свою критическую область, но терпит неудачу, поскольку другой процесс уже находится в своей критической области. Следовательно, *Б* временно приостанавливается до наступления времени *Т*3, когда *А* покинет свою критическую область, позволяя *Б* тут же войти в свою критическую область. Со временем (в момент *Т*4) *Б* покидает свою критическую область.

В отличие от процессов, потоки имеют общее адресное пространство (взаимодействующие потоки, реализованные в различных адресных пространствах, подпадают под категорию взаимодействия процессов). Механизмы синхронизации, применяемые к процессам, в полной мере применимы и к потокам: сходные проблемы и сходные методы их решения. В данном изложении проблемы рассматриваются в контексте процессов, но нужно иметь в виду, что те же проблемы и решения применяются и в отношении потоков.

Аппаратные инструкции синхронизации

Аппаратные инструкции синхронизации реализуют **атомарные** примитивные операции, на основе которых можно строить механизмы синхронизации более высокого уровня. Атомарность означает, что вся операция выполняется как целое и не может быть прервана посредине. Атомарные примитивные операции для синхронизации, как правило, выполняют вместе 2 действия: запись значения и проверку предыдущего значения. Это дает возможность проверить условие и сразу записать такое значение, которое гарантирует, что условие больше не будет выполняться.

Try-and-set lock (TSL)

Инструкции типа try-and-set записывают в регистр значение из памяти, а в память — значение 1. Затем они сравнивают значение в регистре с 0. Если в памяти и был 0 (те. доступ к критической области был открыт), то сравнение пройдет успешно, и в то же время в память будет записан 1, что гарантирует, что в следующий раз сравнение уже не будет успешным, те. доступ закроется.

Compare-and-swap (CAS)

Инструкции типа compare-and-swap записывают в регистр новое значение и при этом проверяют, что старое значение в регистре равно запомненому ранее значению.

```
TEMP ← DEST
IF accumulator = TEMP
THEN
        ZF ← 1;
        DEST ← SRC;
ELSE
        ZF ← 0;
        accumulator ← TEMP;
        DEST ← TEMP;
FI;
```

В x86 инструкция называется CMPXCHG.

CAS инструкции не могут отследить ситуацию, когда значение в регистре было изменено на новое, а потом снова было возвращено к предыдущему значению. В большинстве случаев это не влияет на работу алгоритма, а в тех случаях, когда влияет, необходимо использовать инструкции с проверкой на такую ситуацию, такие как LL/SC.

Среди других инструкций можно отметить Двойной CAS и Load-link/store-conditional (LL/SC).

Системные механизмы синхронизации

Семафор

Семафор — это примитив синхронизации, позволяющий ограничить доступ к критической секции только для N процессов. При этом семафор позволяет реализовать это без использования ожидания.

Концептуально семафор включает в себя неотрицательный целочисленный счетчик и очередь ожидания для процессов. Интерфейс семафора состоит из двух основных операций: *вниз* **(down)** и *вверх* (up). Операция *вниз* атомарно проверяет, что счетчик больше 0 и уменьшает его. Если счетчик равен 0, процесс блокируется и ставится в очередь ожидания. Операция *вверх* увеличивает счетчик и посылает ожидающим потокам сигнал пробудиться, после чего один из этих процессов сможет повторить операцию *вниз* .

Бинарный семафор — это семафор с N = 1.

Приведем решение задачи производителя-потребителя при помощи семафоров. Для этого операцию изменения буфера поместим в критическую секцию. Далее нам нужно организовать ожидание производителя в случае полного буфера и ожидание потребителя в случае пустого буфера.

Для организации критической секции используем двоичный семафор *lock*. Для производителя нам понадобится семафор *empty*, текущее значение которого равно количеству свободных мест в буфере. Для потребителя создадим семафор *full*, текущее значение которого равно количеству занятых мест в буфере.

Псевдокод для решения выглядит так

```
semaphore mutex = 1;
semaphore empty = N;
semaphore full = 0;
 void producer(void) {
  int item;
  while (TRUE) {
  item = produce_item( );
  down(&empty);                    /* уменьшение счетчика пустых мест */
down(&mutex);                      /* вход в критическую область */
insert_item(item);                 /* помещение новой записи в буфер */
up(&mutex);                        /* покинуть критическую область */
  up(&full); } }                   /* увеличение счетчика занятых мест */
void consumer(void) {
  int item;
  while (TRUE) {
  down(&full);                     /* уменьшение счетчика занятых мест*/
  down(&mutex);                    /* вход в критическую область */
  item = remove_item( );           /* извлечение записи из буфера */
  up(&mutex);                      /* выход из критической области */
  up(&empty); }                    /* увеличение счетчика пустых мест */
```

Мьютекс (mutex)

Мьютекс — от словосочетания mutual exclusion, те. взаимное исключение — это примитив синхронизации, напоминающий бинарный семафор с дополнительным условием: разблокировать его должен тот же поток, который и заблокировал.

Монитор

Монитор — это механизм синхронизации в объектно-ориентированном программировании, при использовании которого объект помечается как синхронизированный и компилятор добавляет к вызовам всех его методов (или только выделенных синхронизированных методов) блокировку с помощью мьютекса. При этом код, использующий этот объект, не должен заботиться о синхронизации. В этом смысле монитор является более высокоуровневой конструкцией, чем семафоры и мьютексы.

Условная переменная (condition variable)

Условная переменная — примитив синхронизации, позволяющий реализовать ожидание какого-то события и оповещение о нем. Над ней можно выполнять такие действия:

- ожидать *(wait)* сообщения о каком-то событии
- сигнализировать *(signal)* событие всем потокам, ожидающим на данной переменной.

Большинство мониторов поддерживают внутри себя использование переменных условия. Это позволяет нескольким потоком заходить в монитор и передавать управление друг другу через эту переменную.

Механизмы синхронизации в Win32 API

Функции ожидания.

Процедурные методы синхронизации в Windows используются по отношению к объектам, реализованным в ядре операционной системы. Для такой синхронизации применяются только объекты ядра, которые могут находиться в сигнальном (свободном) и несигнальном (занятом) состоянии. К ним относятся рассмотренные ранее объекты: задания, процессы, потоки. Эти объекты переходят в сигнальное состояние при завершении исполнения. Имеется группа объектов, используемых специально для синхронизации потоков и процессов, - это события, мьютексы, семафоры и таймеры. Отдельную группу образуют объекты, предназначенные для синхронизации ввода-вывода.

С точки зрения программиста все перечисленные объекты имеют общее свойство: их описатели можно использовать в функциях ожидания *WaitForSingleObject*, *WaitForMultipleObject* и некоторых других. Если объект находится в несигнальном состоянии, то вызов функции ожидания с описателем объекта блокируется. Дескриптор потока, который вызвал функцию, помещается в очередь этого объекта ядра, а сам поток переходит в состояние ожидания. Когда объект ядра переходит в сигнальное состояние, выполняются действия, специфичные для данного типа объекта ядра. Например, может быть разблокирован один или все потоки, ожидающие на данном объекте ядра.

В функцию *WaitForSingleObject* передаются два параметра: описатель объекта и значение таймаута. Таймаут определяет предельное время нахождения потока в блокированном состоянии. В функцию *WaitForMultipleObjects* передается массив описателей объектов ядра. Для этого указывается адрес начала массива и количество описателей в нем. Также указывается параметр, определяющий семантику ожидания на группе описателей: можно ждать перехода всех объектов ядра в сигнальное состояние или какого-то одного объекта. Указывается также таймаут.

При возврате из функции ожидания обычно требуется определить причину завершения функции. В случае ошибки функция возвращает значение WAIT_FAILED. Для уточнения состояния ошибки далее используют *GetLastError* и другие функции. В случае завершения по таймауту возвращается значение WAIT_TIMEOUT. Если причиной возврата является переход в сигнальное состояние, возвращается значение WAIT_OBJECT_0.

Если выполняется ожидание на функции *WaitForMultipleObjects*, то, чтобы узнать, какой именно объект перешел в сигнальное состояние, нужно проверить, что значение, возвращенное функцией, не равно WAIT_FAILED и WAIT_TIMEOUT и вычесть из него константу WAIT_OBJECT_0. В результате мы получим индекс описателя объекта, перешедшего в сигнальное состояние, в массиве описателей, переданном в функцию *WaitForMultipleObject.*

Управление объектами ядра.

Рассмотрим операции, относящиеся как к объектам, использующимся в функциях ожидания, так и к другим объектам ядра в операционной системе Windows.

Для *создания* объектов ядра используются индивидуальные для каждого объекта функции, имеющие префикс *Create*. Например, мьютекс может быть создан вызовом

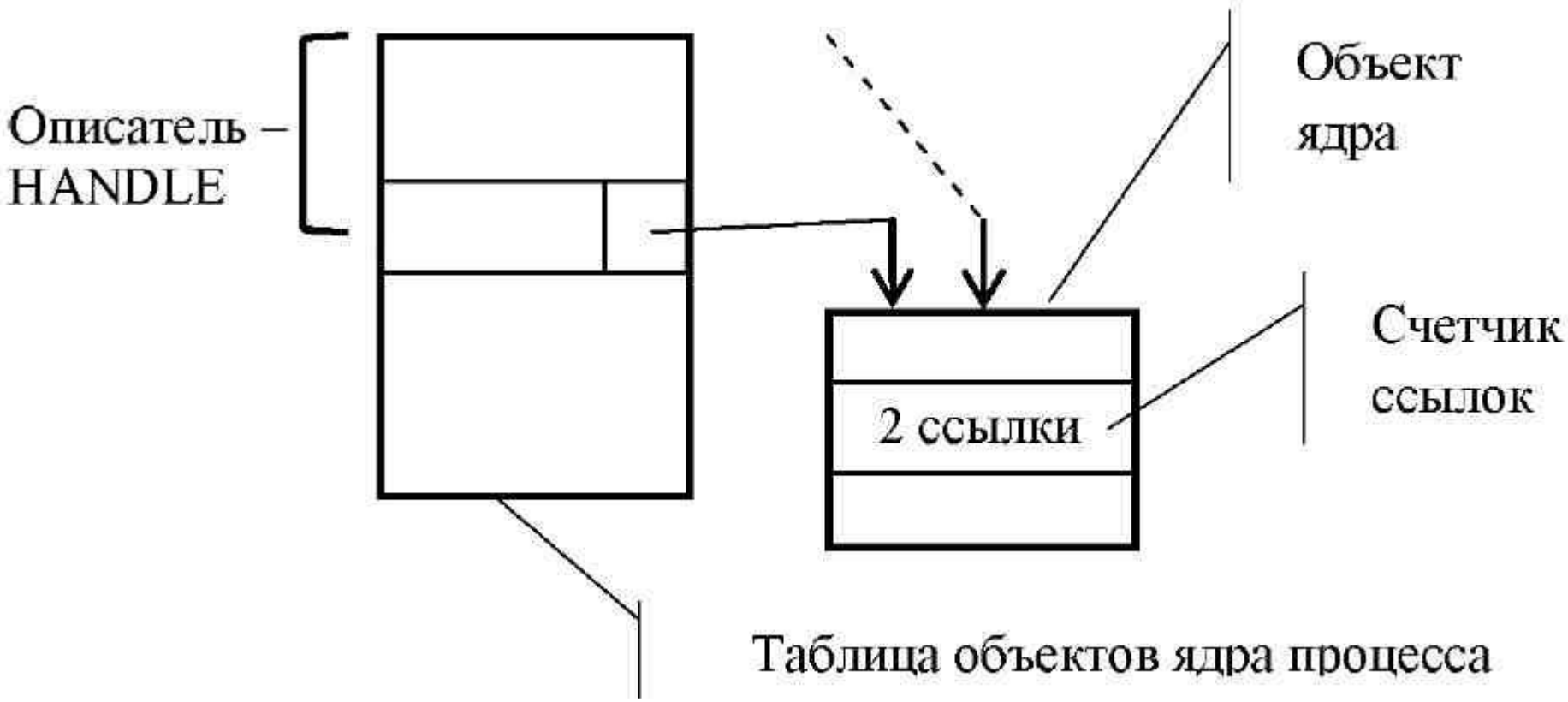
HANDLE mutex=CreateMutex(NULL, FALSE, NULL).

Обычно первым параметром всех функций, создающих объекты ядра, является структура атрибутов безопасности, а последним - имя объекта.

Закрытие описателя любого объекта ядра выполняется вызовом функции *CloseHandle*.

Чтобы понять принцип работы функции, рассмотрим более детально, что представляет собой описатель объекта ядра.

Для каждого процесса в ядре операционной системы создается таблица описателей. Запись в таблице содержит некоторые атрибуты описателя и указатель на объект ядра. На один и тот же объект ядра могут указывать несколько описателей, возможно из таблиц описателей разных процессов. В любом объекте ядра имеется специальный атрибут для подсчета ссылок на объект. Значение описателя, используемое в адресном пространстве процесса, - это смещение на запись в таблице описателей процесса.


Описатель –
HANDLE
Объект
ядра
2 ссылки
Счетчик
ссылок
Таблица объектов ядра процесса

При закрытии описателя происходит модификация атрибута, указывающего на то, что запись теперь недействительна. Далее происходит декремент счетчика ссылок у объекта. Если значение счетчика ссылок достигает нуля, происходит удаление объекта из памяти ядра операционной системы.

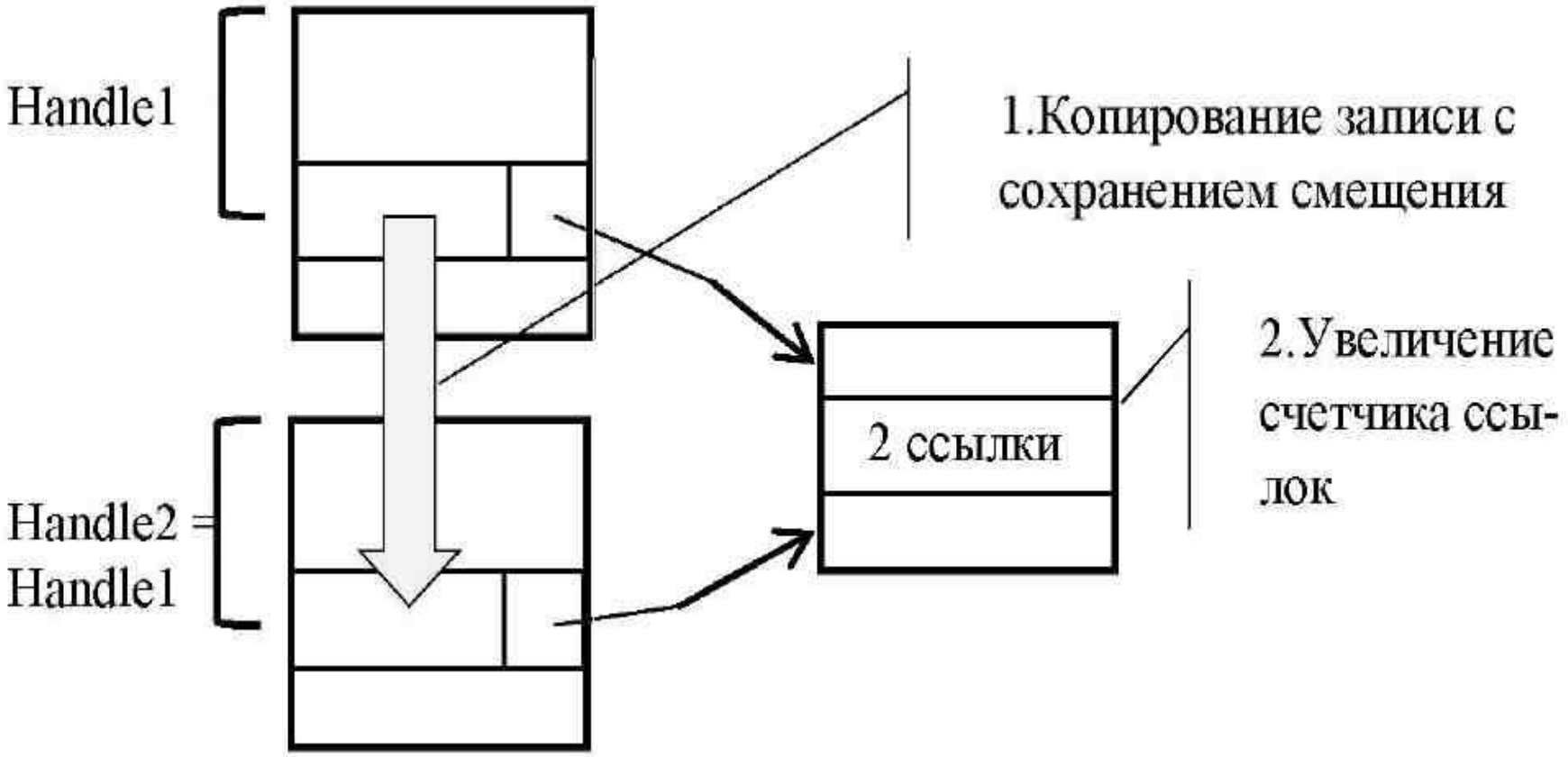
Важное преимущество объектов ядра - возможность их использования для синхронизации потоков, принадлежащих разным процессам. Для этой цели нужно уметь передавать описатели объектов между процессами. Их можно передавать путем *а* -наследования, *б* - именования и *в* - дублирования.

а) При *наследовании* вначале необходимо модифицировать атрибут наследования в таблице описателей процесса. Это можно сделать непосредственно при создании описателя, как показано ниже.

```
SECURITY_ATTRIBUTE sa;
sa.nlength = sizeof(sa);
sa.lpSecurityDescriptor=NULL; /*NULL - защита по умолчанию;
определяет, кто может пользоваться объектом (при NULL -
создатель процесса и администраторы) */
sa.bInheritHandle=TRUE; /*наследовать описатель*/
HANDLE hMutex = CreateMutex (&sa, FALSE, NULL);
```

Можно воспользоваться функциями *GetHandleInformation* и *SetHandleInformation* для изменения атрибутов описателя уже после создания объекта ядра.

Наследование описателей происходит следующим образом. При создании процесса функцией *CreateProcess* для него создается новая таблица описателей. В эту таблицу копируются все записи родительского процесса, имеющие атрибут наследования. Записи помещаются по тем же смещениям, по которым размещались исходные записи родительского процесса. Для каждой скопированной записи производится инкремент счетчика ссылок в объекте ядра.


Handle1
Handle2 =
Handle1
2 ссылки
1.Копирование записи с сохранением смещения
2.Увеличение счетчика ссы-лок

Дочерний процесс может использовать ссылки на унаследованные объекты ядра. Для передачи значений унаследованных описателей в дочерний процесс обычно используется командная строка или переменные окружения. Если дочерний процесс уже создан, то нельзя воспользоваться механизмом наследования.

б) При создании объекта ядра может быть указано его имя:

HANDLE mutex = CreateMutex(NULL, FALSE, "SomeMutex").

В данном случае любой процесс может получить *доступ к объекту ядра по имени,* если такой доступ разрешен настройками безопасности. При повторном вызове функции *Create* проводится проверка: имеется ли объект ядра с данным именем и типом. Если объект с таким именем и типом уже существует, то функция не создает новый объект, а возвращает описатель существующего объекта (при этом остальные параметры в вызове игнорируются). При необходимости можно проверить, создан ли объект ядра или получен описатель ранее созданного объекта:

if(GetLastError()==ERROR_ALREADY_EXISTS) { ... }

Механизм ссылки на объекты ядра подходит для всех случаев взаимодействия, когда можно определить соглашения на имена создаваемых объектов. Этот механизм также часто используется для контроля количества запущенных экземпляров приложения.

в) Универсальным способом передачи описателей объектов ядра между процессами является *дублирование описателей.* Дублирование выполняется функцией *DuplicateHandle*:

BOOL DuplicateHandle(
*HANDLE hSourceProcessHandle, /*описатель исходного процесса*/*
*HANDLE hSourceHandle, /*дублируемый описатель в процессе-источнике*/*
*HANDLE hTargetProcessHandle, /*процесс-приемник*/*
*PHANDLE phTargetHandle, /*описатель в процессе-приемнике*/*
*DWORD dwDesiredAccess, /*настройка атрибутов дублируемого описателя*/*
BOOL bInheritHandle,
DWORD dwOptions).

В дублировании принимают участие (в общем случае) три процесса: процесс, выполняющий дублирование, процесс-источник и процесс-приемник. Если источником или приемником является управляющий процесс, то вместо соответствующего описателя помещается вызов функции *GetCurrentProcess*. Особенностью дублирования является то, что необходимо использовать механизм межпроцессного взаимодействия для передачи продублированного описателя в процесс-приемник. Функция *DuplicateHandle* его сформирует, но нужно также поставить в известность сам процесс-приемник о том, что в его таблицу описателей добавлен новый описатель.

Теперь рассмотрим объекты синхронизации и специфические для каждого объекта функции процедурной синхронизации.

События

События используются при реализации кооперативной синхронизации, когда один процесс (поток) ждет поступления данных от другого. При наступлении события объект переходит в сигнальное состояние.

Если событие не наступило, объект находится в несигнальном состоянии. В зависимости от того, каким образом осуществляется перевод события в несигнальное состояние, существуют два типа событий: событие со сбросом вручную и событие с автоматическим сбросом, а также две функции *SetEvent* и *PulseEvent*. Любую функцию можно использовать с любым типом события. В сочетании это дает четыре варианта действий.

Оба типа событий создаются функцией *CreateEvent*:

```
HANDLE CreateEvent(
PSECURITY_ATTRIBUTES sa, /*атрибуты безопасности*/
BOOL  fManualReset, /* TRUE  -  со сбросом вручную*/
BOOL flnitialState, /*начальное состояние события*/
LPCTSTR name); /*имя события*/.
```

Во время выполнения функции *SetEvent* событие устанавливается в сигнальное состояние. Дальнейшие действия зависят от типа события.

- Для событий с автоматическим сбросом возобновляется выполнение одного процесса, который ожидает на событии. Если ни один процесс не ожидает на событии, событие остается в сигнальном состоянии до тех пор, пока какой-либо процесс не перейдет в ожидание на этом событии. После этого процесс немедленно продолжит выполнение, а состояние процесса автоматически сбрасывается (другие процессы будут ожидать).
- Для событий с ручным сбросом возобновляются все процессы, которые ожидают, после чего событие остается в сигнальном состоянии. Поэтому все последующие процессы немедленно продолжит выполнение в случае попытки выполнить ожидание. Событие остается в таком состоянии, пока какой либо процесс не сбросит его вручную вызовом *ResetEvent.*

Выполнение функции *PulseEvent* подобно последовательным вызовам *SetEvent* и *ResetEvent.* При этом событие переходит в сигнальное состояние и возобновляет один из процессов, который ожидает на событии (для событий с автоматическим сбросом) или все процессы (для событий с ручным сбросом), после этого состояние события сбрасывается. Если нет ожидающих процессов, событие немедленно сбрасывается и факт сигнализации исчезает.

Вызов функции *SetEvent* сохраняет состояние события, а вызов функции *PulseEvent* – нет.

Семафоры

Семафоры - универсальные объекты процедурной синхронизации, предложены Э. Дейкстра. Семафоры выполняют роль счетчика числа доступных ресурсов. С семафорами определены две операции. Для возврата ресурсов в пул доступных ресурсов используется операция увеличения счетчика (up-операция). Для захвата ресурсов используется операция уменьшения счетчика (down-операция). Если счетчик ресурсов принимает значение ноль, то поток блокируется, до тех пор, пока ресурс не будет возвращен в пул другим потоком с использованием up-операции. В любой момент счетчик не может быть больше максимального числа ресурсов и меньше нуля.

Семафор создается при помощи функции *CreateSemaphore:*

HANDLE CreateSemaphore(
*PSECURITY_ATTRIBUTES sa, /*атрибуты безопасности*/*
*LONG lInitialCount, /*начальное значение счетчика ресурсов */*
*LONG lMaxCount, /*предельное значение счетчика ресурсов */*
*LPCTSTR name); /*имя семафора*/.*

Инкремент счетчика семафора выполняется при помощи вызова функции *BOOL ReleaseSemaphore (HANDLE, LONG ReleaseCount).* В ней, в отличие от абстрактных операций *up*, можно указать, насколько следует увеличить счетчик (параметр *ReleaseCount*). Декремент счетчика выполняется при помощи любой функции ожидания, примененной к семафору, например, *WaitForSingleObject*.

Мьютексы

Мьютекс специально предназначен для решения задачи взаимного исключения, защиты критической секции. Для создания мьютекса используется функция *CreateMutex*:

HANDLE CreateMutex(
*PSECURITY_ATTRIBUTES sa, /*атрибуты безопасности*/*
*BOOL fInitialOwner, /*признак, является ли поток, создающий мьютекс, его владельцем*/*
*LPCTSTR name); /*имя мьютекса*/.*

Операция освобождения мьютекса (*up*) выполняется вызовом функции *BOOL ReleaseMutex (HANDLE)*. Операция захвата мьютекса выполняется при помощи любой функции ожидания, например, *WaitForSingleObject*.

Мьютекс можно рассматривать как бинарный семафор. Также мьютекс похож на критическую секцию. От семафора мьютекс отличается тем, что он имеет атрибут владения (как критическая секция). Отличие от критических секций состоит в том, что при помощи мьютекса можно синхронизировать потоки в разных процессах и указывать таймаут.

Если поток владеет мьютексом, то вызов функции ожидания с этим мьютексом завершится успешно, при этом увеличится внутренний счетчик рекурсий. Функция *ReleaseMutex* выполняет декремент счетчика рекурсий и освобождает мьютекс, если счетчик достигает значения ноль. Также ведет себя и критическая секция.

Если попытаться освободить мьютекс из протока, который им не владеет, то возникнет ошибка и функция *GetLastError* вернет значение *ERROR_NOT_OWNED*.

Если поток, владеющий мьютексом, завершается, то мьютекс переходит в несигнальное состояние и может быть использован другими потоками. Вызов функции ожидания с таким мьютексом будет выполнен, но функция *GetLastError* вернет ошибку *WAIT_ABANDONED*.

Межпроцессное взаимодействие

Реализация межпроцессного взаимодействия выполняется тремя основными методами: совместно используемой памяти (shared memory), отображаемой памяти (mapped memory) и передачи сообщений (message passing).

Методы **совместно используемой памяти** дают возможность обмениваться данными через общий буфер памяти. Перед обменом данными каждый из участвующих процессов должен присоединить этот буфер к своему адресному пространству. Никаких средств синхронизации доступа к этим данным совместно используемая память не обеспечивает, программист должен организовать их сам.

Обычно **отображаемая память** использует интерфейс файловой системы, в этом случае обычно говорят о файлах, отображаемых в память. В Win32 отображение файла в память выполняется функцией *CreateFileMapping*, которая создает *объект отображения*. Несколько процессов могут отобразить в свои адресные пространства один и тот же файл. Изменения, сделанные в памяти одним процессом, будут видны и в других процессах.

Объектам отображения можно давать имена, после чего они могут использоваться несколькими процессами. Таким образом можно реализовать и совместно используемую память.

На практике используют следующие **методы передачи сообщений**.

Канал представляет собой циклический буфер, в который запись выполняет один процесс, а чтение – другой. В конкретный момент времени к каналу имеет доступ только один процесс.Различают *именованные* и *неименованные каналы*. В Win32 они реализуются функциями CreateNamedPipe и CreatePipe. Обмен данными через канал может быть односторонним и двусторонним.

Очередь. Процессы могут создавать очереди, записывать в конкретные очереди и читать оттуда. С очередью может работать одновременно несколько процессов. Чтобы процессы могли различать адресованные им сообщения, каждому из них присваивают тип.

Сокеты. Эта технология предназначена прежде всего для сетевого обмена данными.

Удаленный вызов процедур. Клиент посылает запрос на выполнение процедуры на сервере и переходит в состояние ожидания. Сервер выполняет процедуру и отправляет результат клиенту. Основные виды технологий – Sun RPC и Microsoft RPC.

Неблокирующая синхронизация

Неблокирующая синхронизация — это группа подходов, которые ставят своей целью решить проблемы синхронизации альтернативным путем без явного использования блокировок и основанных на них механизмов.

Shared-nothing (ничего общего)

Архитектуры программ без общего состояния рассматривают вопросы построения систем из взаимодействующих компонент, которые не имеют разделяемых ресурсов и обмениваются информацией только через передачу сообщений. Такие системы, как правило, являются намного менее связными, и поэтому лучше поддаются масштабированию и являются менее чувствительными к отказу отдельных компонент

В теоретических работахы такой подход получил название **взаимодействующие параллельные процессы** (Communicating Parallel Processes, **CPP**). Практическая реализация этой концепции — язык Erlang. В этой модели единицей вычисления является легковесный процесс, который имеет "почтовый ящик", на который ему могут отправляться сообщения от других процессов, если они знают его ID в системе. Отправка сообщений является неблокирующей (асинхронной), а прием является синхронным: те. процесс может заблокироваться в ожидании сообщения. При этом время блокировки может быть ограниченно программно.

CSP

Взаимодействующие последовательные процессы (Communicating Sequential Processes, CSP) — это еще один подход к организации взаимодействия без использования блокировок. Единицами взаимодействия в этой модели являются процессы и каналы. В отличие от модели CPP пересылка данных через канал в этой модели происходит; как правило, синхронно, что дает возможность установить определенную последовательность выполнения процессов. Данная концепция реализована в языке программирования Go.

Программная транзакционная память

Транзакция — это группа последовательных операций, которая представляет собой логическую единицу работы с данными. Транзакция может быть выполнена либо целиком и успешно, соблюдая целостность данных и независимо от параллельно идущих других транзакций, либо не выполнена вообще и тогда она не должна произвести никакого эффекта. В теории баз данных существует концепция ACID, которая описывает условия, которые накладываются на транзакции, чтобы они имели полезную семантику A означает **атомарность** — транзакция должна выполняться как единое целое. C означает **целостность** (consistency) — в результате транзакции будут либо изменены все данные, с которыми работает транзакция, либо никакие из них. I означает **изоляция** — изменения данных, которые производятся во время транзакции, станут доступны другим процессам только после ее завершения. D означает **сохранность** — после завершения транзакции система БД гарантирует, что ее результаты сохраняться в долговременном хранилище данных. Эти свойства, за исключением, разве что, последнего могут быть применимы не только к БД, но и к любым операциям работы с данными. Программная система, которая реализует транзакционные механизмы для работы с разделяемой памятью — это Программная транзакционная память (Software Transactional Memory, STM). STM лежит в основе языка программирования Clojure, а также доступна в языках Haskell, Python (в реализации PyPy) и других.