Санкт-Петербургский политехнический университет
Институт компьютерных наук и технологий

СИСТЕМНОЕ ПРОГРАММНОЕ ОБЕСПЕЧЕНИЕ АВТОМАТИЗИРОВАННЫХ СИСТЕМ УПРАВЛЕНИЯ

VII СЕМЕСТР

Лектор: *Ерофеев Сергей Анатольевич*

Автор: *Шкалин Кирилл Павлович*

осень 2022

Содержание

1 Системное программное обеспечение ЭВМ

Это комплекс программ, которые обеспечивает взаимодействие приложений пользователя с аппаратурой и эффективное управление аппаратурой, к которой относятся:

- Процессор
  `Регистры, контекст (из смены контекста), тактовая частота, элементарные операции (атомарные)`
- ОЗУ
  `Виртуальная память`
- Устройство ввода/вывода
- Сетевое оборудование
- Коммуникационное оборудование

В состав СПО входят 6 базовых компонентов:

1. ОС.
2. Система управления файлами (СУФ).
3. Интерфейсные оболочки для взаимодействия пользователя с ОС и операционные среды.
4. Система программирования.
5. Утилиты.
6. СУБД (система управления базами данных).

ОС — базовый комплекс управляющих и обрабатывающих программ, которые управляют аппаратно-программными ресурсами ЭВМ и задачами, при выполнении которых используются эти ресурсы. ОС выполняет следующий задачи:

1. Обеспечение работы пользовательских приложений и систем программирования.
2. Прием и обработка пользовательских команд (в том числе с консоли).
3. Прием и выполнение запросов на запуск, приостановку и остановку других программ.
4. Загрузка программ подлежащих исполнению в оперативную память.
5. Передача управление программе и выполнение программы процессором.
6. Идентификация программ и данных.
   `Каждому объекту должен сопоставляться собственный идентификатор.`
7. Обеспечение работы системы управления файлами и системы управления базами данных.

[1] СПО — системное программное обеспечение.

8. Управление операциями ввода/вывода.
9. Распределение памяти.
10. Диспетчеризация задач.
    `Выборка задачи для смены контекста`
11. Поддержка механизма обмена данными между исполняемыми программами.
12. Защита памяти.

СУФ — система организации данных, хранения их и обращения к ним по средствам файлов вместо низкоуровнего доступа по физическим адресам. Файл — цепочка кластеров во вторичной памяти. Кластер — минимальлно адресуемая единица памяти 4 кБ. Сектор — минимальная единица вторичной памяти 512 Байт. С точки зрения ОС весь диск представляет из себя набор кластеров.

Драйверы файловой системы привязывают кластеры к файлам и каталогам. **Каталог** — файл специального формата, который содержит список файлов в этом каталоге. Эти же драйверы отслеживают, какие из кластеров в настоящие время используются, какие свободные, а какие помечены как неисправные. Вместе с тем файловая система не обязательно напрямую связана с физическим носителем информации. Существуют виртуальные и сетевые файловые системы, которые являются всего лишь способом доступа к файлам, находящимся на удаленном компьютере.

Операционные среды — интерфейс необходимый прикладным программам для обращения к системным ресурсам ОС с целью получения определенного сервиса. Работа программной среды определяется прикладными программными интерфейсами — API. **API** — Application Program Interface. Примеры: Explorer, XWindow. В семейство ОС Microsoft с интерфейсом Explorer заменяемой является только интерфейсная оболочка, а операционная среда является неизменной. К этому классу СПО относятся эмуляторы виртуальных машин (VMWare создает образ одной ОС на базе другой).

Система программирования — включает в себя:

1. Трансляторы — Специальные программы переводчики, которые переводят программы пользователей, написанные на различных ЯП, в машинный код. 3 вида: ассемблер, компиляторы (исходного модуля → объектный модуль), интерпретаторы (системная программа, которая транслирует каждый оператор исходной прграммы в промежуточный код, интерпретирует его по средствам одной или нескольких команд и выполняет эти команды).
2. Библиотеки функций
3. Редакторы
4. Компоновщики
5. Отладчики
6. Специальные программы для выполнения вспомогательных функций.

1.1 Определение Ядра ОС

Все модули ОС делятся на две группы

1. Модули ядра. Включают в себя:
   (a) Планировщик (диспетчер)
   (b) Драйверы устройств ввода/вывода
   (c) Файловаю систему
   (d) Сетевую систему

   Управляют задачами (потоками и процессами), памятью, устройствами и т.д. Функции такого типа являются внутрисистемными и недоступны для приложений. Ряд функций ядра служит для поддержки приложений, создавая для них, так называемую, прикладную программную среду.
2. Модули выполняющие вспомогательные функции ОС (утилиты)

Функции ядра являются наиболее часто используемыми функциями операционной системы, поэтому скорость их выполнения определяет производительность всей системы в целом. Для обепечения высокой скорости работы операционной системы все модули ядра или большая их часть постоянно находятся в оперативной памяти (являются резидентными) и не выгружаются оттуда. Такие модули операционной системы, как утилиты, системные обработчики и библиотеки обычно загружаются в оперативную память только на время выполнения своих функций (являются транзитными). Ядо оформляется в видет отдельного модуля специального формата, отличного от формата пользовательских приложений.

Ключевым свойством операционной системы, основанной на ядре, является защита кодов и данных операционной системы засчет выполнения функций ядра в привилегированном режиме.

Операционная система должна иметь по отношению к приложениям определенные привилегии, то есть защиту от приложений, а также должна контролировать доступ приложений к ресурсам компьютера в многозадачном режиме.

Ни одно приложение не должно иметь возможности без разрешения операционный системы получать дополнительную область памяти, занимать процессор дольше разрешенного времени, а также непосредственно управлять совместно используемыми внешними устройствами.

Привелегии обеспечиваются засчет специальных средств аппаратной поддержки. Для этого аппаратная платформа должна поддерживать минимум два режима работы: пользовательский и привилегированный. Соответственно приложение ставится в подчиненное положение засчет запрета выполнения в пользоватльском режиме некоторых критичных команд, например связанных с переключением процессора с задачи на задачу, управлением устройствами ввода-ввывода, доступом к механизмам распределения и защиты памяти.

Выполнение некоторых команд в пользовательском режиме запрещается безусловно, тогда как другие команды запрещается выполнять только при определенных условиях (пример: команды ввода-вывода могут быть запрещены приложениям при доступе к контроллеру жесткого диска, который хранит данные общие для операционной системы и всех приложений, но разрешены при доступе к последовательному порту, который выделен в монопольное владение для определенного приложения). Аналогичным образом

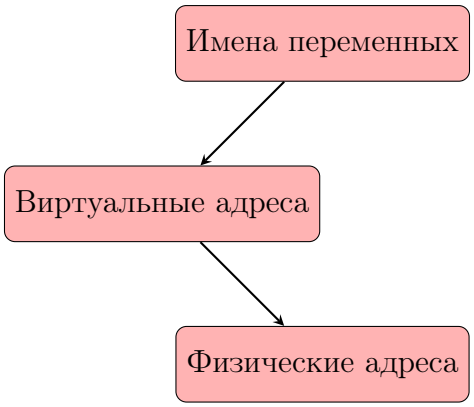


Рис. 1: Механизм распределения памяти.

обеспечиваются привилегии при доступе к памяти (например: выполнение команды доступа к памяти для приложения разрешается тогда, когда команда обращается к области памяти, которая отведена данному приложению операционной системой, и запрещается при обращение к областям памяти занимаемым операционной системой или другим приложением).

Существуют две операции работы с памятью: выделение и резервирование. Полный контроль операционной системы над доступом к памяти достигается засчет того, что команды конфигурирования механизмов защиты памяти (напримр: изменение параметров защиты) разрешается выполнять только в привелигированном режиме.

2 Классификация операционных систем по функциональности

1. По количеству пользователей одновременно обслуживаемых системой операционные системы делятся на
   (a) Однопользовательские;
   (b) Многопользовательские.
2. По числу потоков, которые могут одновременно выполняться под управлением операционной системы
   (a) Однозадачные;
   (b) Многозадачные.

Наиболее характерными критериями эффективности вычислительных систем являются:

1. Пропускная способность — количество задач, выполняемых в единицу времени.
2. Удобство работы пользователей, которое заключается в частности в том, что они имеют возможность интерактивного диалога одновременно с несколькими приложениями на одной машине.
3. Реактивность системы, то есть способность системы выдерживать заранее заданные временные интервалы между запуском программы и получением результата.

По критериям эффективности системы делятся на:

1. Системы пакетной обработки. Главный критерий эффективности — максимальная пропускная способность, то есть решение максимального количества задач в единицу времени. Используется следующая схема функционирования:

   В начале формируется пакет заданий → В момент начала работы системы из пакета формируется многозадачный пулл, одновременно выполняемых задач. При формировании пулла выбираются задачи, для выполнения которых используются разные ресурсы. При этом должна обеспечиваться сбалансированная загрузка всех устройств ЭВМ с минимальным простоем. Пока одна задача ожидает какого-либо события, процессор не простаивает.

   В системах под управлением пакетных операционных систем невозможно гарантировать выполнение конкретного задания в течении определенного периода времени. В системах пакетной обработки переключение процессора с одной задачи на другую выполняется по инициативе самой активной задачи. Поэтому существует высокая вероятность того, что одна задача может на долго занять процессор и выполнение интерактивных задач станет невозможным.

   Взаимодействие пользователя с вычислительной машиной, на которой установлена пакетная операционная система сводится к тому, что пользоваетль передает задание и ждет получения результата.

2. Системы разделения времени. Каждому приложению попеременно периодически выделяется квант процессорного времени, по истечении которого переключаются на другое приложение (при его наличии). `POSIX требует наличие в системе двух политик`

$$\begin{cases} FIFO \\ RR \end{cases}$$

Каждое приложение регулярно получает процессорное время. Пользователь регулярно получает возможность диалога с приложением. Производительность операционной системы определяется скоростью смены контекста.

Пропускная способность меньше, чем в пакетной обработке. Возрастают накладные расходы.

Единственный критерий — удобство и эффективность работы пользователя.

3. Системы реального времени. Предназначены, как правило, для управления аппаратно-программными комплексами, для которых должны быть выполнены требования по их реактивности. Критерием эффективности является способность к выполнению критических задач в течении заданного интервала времени. Этот интервал называется временем реакции на событие (событие — появление сигнала в операционной системе). Если задержка реакции системы на событие дольше определенного интервала недопустима, то система называется «Системой жесткого реального времени» (hard real time system). Иначе (если допустима) — soft real time system.

Многозадачный пулл представляет собой фиксированный набор заранее разработанных программ, а выбор программы для выполнения осуществляется по сигналам прерывания или в соответствии с расписанием плановых работ.

Своппинг (подкачка) запрещены в системах реального времени.

3 Классификация операционных систем по архитектуре

3.1 Монолитное ядро

Монолитное ядро — архитектура операционной системы, при которой все ее компоненты являются не самостоятельными модулями, а составными частями одной программы, используют общие структуры данных и взаимодействуют друг с другом путем непосредственного вызова процедур.

3.2 Слоистые (многослойные) системы

Вся система разбивается на ряд последовательных слоев с четко определенными связями между ними, с тем, чтобы объекты слоя n могли вызывать только объекты слоя $n-1$.

Например:

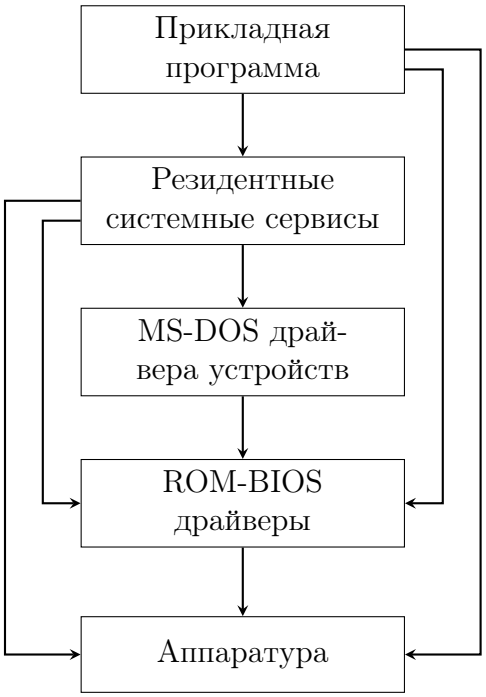
0. Аппаратура.

1. Планирование задач (планировщик).

2. Управление памятью.
3. Драйвер устройства связи оператора и консоли.
4. Управление вводом/выводом.
5. Интерфейс пользователя.

Преимущества слоистых систем заключается в разделении функционала по слоям и организации быстрой и удобной отладки и тестирования.

3.3 Уравневые системы



3.4 Микроядерная (клиент-сервер)

Перенос значительной части на уровень пользователя с одновременной минимизацией ядра (создание микроядра). При такой микроядерной архитектуре операционной системы, большинство ее модулей являются самостоятельными программами, а взаимодействие между ними осуществляет специальный модуль ядра, называемый микроядром.

Микроядро обеспечивает:

1. Взаимодействие между программами.
2. Планирование использования процессора (диспетчеризация).
3. Первичную обработку прерываний.
4. Операции ввода/вывода.
5. Базовое управление памятью.

Обработка прерывания:

1. Идентификация прерывания.
2. Устанавливается тип прерывания.
3. Поиск обработчика по таблице и передача управления ему.

Остальные компоненты системы взаимодействуют друг с другом путем передачи сообщений через микроядро. Пользовательские приложения являются клиентами системы, а микроядро выполняет роль посредника (деспетчера сообщений) между клиентскими (пользовательскими) приложениями (клиентами) и системными сервисами (серверами).

Преимущества микроядерной архитектуры:

1. Упрощается процесс добавления и исключения новых компонентов. Такие операции называются масштабированием, а поддержка этой операции называется масштабируемость (свойство системы).
2. Упрощается процесс отладки компонентов ядра (засчет масштабируемости и за счет того, что компоненты ядра ничем не отличаются от пользовательских приложений).
3. Повышается отказоустойчивость системы.

Минусы:

1. Передача данных от модуля к модулю через сторонние сообщения. Снижается производительность. Возрастают накладные расходы вычислительной мощности.

Пример смешанной системы

Есть смешанные системы с сочетанием монолитного ядра и микроядра. Примеры:

1. $\left.\begin{matrix}\text{mkLinux}\\ \text{4.4 BSD}\end{matrix}\right\}$ Микроядро Mach

   Микроядро обеспечиает работу виртуальной памяти и работу низкоуровневых драйверов. Все остальные функции, в том числе взаимодействие с пользовательскими приложениями выполняются монолитным ядром.

2. WindowsNT. Компоненты ядра WindowsNT располагаются в вытесняемой памяти и взаимодействуют друг с другом путем передачи сообщений. Все компоненты ядра работают в одном адресном пространстве и активно используют общие структуры данных. Кроме того существует разделение между режимом ядра и режимом пользователя.

4 Лекция от 29.10.2022

4.1 Определение задач и ее свойства

`Задачи лежат в структуре стека. Задача или процесс или поток. Ядро выбирает из очереди задач ту, которую нужно загрузить первой в зависимости от свойств задач.`

Задача — набор операций для выполнения логических законченных функций системы. Задачей может быть процессом или потоком.

Процесс — независимый программный модуль, который во время выполнения имеет отдельное адресное пространство в памяти.

Поток — операции ялвяющиеся составной частью процесса. Преимущества:

- Количество потоков не ограничено.
- Общее адресное пространство.
- Время переключения между потоками гораздо меньше.

Минусы:

- Общее адресное пространство. (Один поток никак не защищен от другого)

Свойства:

1. Приоритет — целое число. Может быть или статическим или динамическим. `Посмотреть, как приоритет потока зависит от приоритета процесса.`

Дескриптор — содержит ифнормацию о состоянии задачи, расположение образа задачи на диске и в оперативной памяти, приоритете, идентификаторе пользователя создавшего процесс, информацию о родственных процессах, о событиях осуществления которых осуществляет данный процесс и так далее.

Состояния задачи:

- Активное — выполняется на процессоре.
- Блокированное — находится в ожидании какого-либо события.
- Готовое — ждет своей очереди.

Образ — сочетание дескриптора и контекста. Контекст нужен, чтобы можно было продолжить выполнение со снятого места. Дескрипторы объединяются в стек (списке).

Контекст содержит менее оперативную, но более объемную информацию о задаче, необходимую для возобновления выполнения задачи с прерванного места. Содержится:

1. Содержимое регистров процессора.
2. Коды ошибок выполняемых процессором системных вызовов.
3. Информацию о файлах открытых в ходе выполнения данной задачи.
4. Незавершенных операциях ввода/вывода.
5. И другие данные описывающие состояние среды в момент прерывания.

Контекст как и дескриптор доступен только программам ядра, но хранится не в области ядра, а перемещается при необходимости из оперативной памяти на диск. Блок TSB (именованная структура). HANDLE — id дескриптора.

5 Лекция от 12.11.2022

...

Системный вызов является частью интерфейса между операционной системой и пользовательской программой. Пользовательская программа запрашивает сервис у операционной системы, осуществляя системный вызов, и задача переходит в режим ядра.

Кроме того есть исключения. Исключения — события, возникающие в результате выполнения программой недопустимой команды, доступу к ресурсу при отсутствии ресурса или необходимых прав или обращению к отсутствующей странице памяти. Исключения, как и системные вызовы, являются синхронными событиями и делятся на исправимые и неисправимые (фатальные).

Есть прерывания игнорируеммые и неигнорируемые (маскируемые и неигнорируемые).

Каждому прерыванию назначается свой уникальный приоритет.

5.1 Диспетчеризация программ

В рамках диспетчеризации выполняются следующие функции:

1. Определение момента времени для смены исполняемой задачи.
2. Выбор задачи из очереди готовых.
3. Переключение контекстов задач. Выполняется на аппаратном уровне (привязано к платформе)

Адаптивная диспетчеризация:
Если задача слишком долго не запускается на выполнение, ее приоритет уменьшается.
А если слишком долго выполняется, то ее приоритет уменьшается.

QNX: По истечении кванта времени ее приоритет уменьшается на единицу, если есть другая задача с такимже приоритетом в готовом состоянии. Если задача с пониженным приоритетом не выполняется в течении одной секунды, ее приоритет повышается на единицу. Приоритет задачи не может превысить начальный уровень. Если задача блокируется ей немедленно возвращается начальный приоритет.

Адаптивная диспетчеризация используется тогда, когда интенсивно выполняющиеся фоновые процессы разделяют компьютер с пользовательскими процессами работающими в диалоговом режиме.

5.2 Проблема межзадачного взаимодействия

Синхронизация задач заключается в согласовании времени их выполнения с помощью приостановки до наступления некоторого события и последующей активизацией. (в третьей работе можно использовать критические секции (стр 218). Также засекать время выполнения процесса (например GetProcessTime (172 стр)), и каждого потока (GetThreadTime)).

Синхронизация задач нужна, когда задачи взаимосвязанные, когда используются разделяемые данные, когда необходимо синхронизировать потоки по времени.

Следующие функции:

- `WaitForSingleObject`
- `WaitForMultipleObjects`

- `CreateEvent`
- `CreateMutex`

6 Лекция от 26.11.2022

Важным объектом синхронизации является критическая секция программы, то есть ее часть с доступом к разделяемым данным и непредсказуемым результатом выполнения при параллельном изменении этих данных разными потоками.

Три основные проблемы:

1. Гонки.
2. Тупики.
3. Инверсия приоритетов. В системе есть три задачи. Задача n имеет низкий приоритет, задача v имеет высокий, а задача s низкий. Допустим активная задача n захватила ресурс r. Если задача v переходит в состояние готовности, она вытесняет задачу n, и ресурс r остается заблокированым. Решение:
   (a) Наследование приоритетов. Низкоприоритетная задача, захватившая ресурс наследует приоритет от высокоприоритетной задачи, которой этот ресурс нужен. Но в случае когда несколько средне и низкоприоритетных задач разделяют ресурсы с высокоприоритетной задачей, возможна ситуация когда высокоприоритетной задаче придется слишком долго ждать, пока каждая из задач с более низким приоритетом не освободит свой ресурс.
   (b) Протокол предельного приоритета. К стандартным свойствам объекта синхронизации добавляется параметр, который равен максимальному приоритету задачи, которая обращается к этому объекту. Если этот параметр установлен, приоритет любой задачи повышается до указанного уровня и задача не сможет быть вытеснена никакой другой. После разблокирования ресурса приоритет задачи понижается до изначального уровня. Возможность задержки высокоприоритетных задач на время выполнения низкоприоритетных потоков.

6.1 Синхронизация задач на API уровне

6.1.1 Синхронизация в режиме пользователя

Блокирующие переменные

Каждому разделяемому ресурсу или набору критических данных назначается глобальная двоичная переменная, которая равна 1, если ресурс свободен, и значение 0, когда ресурс занят.

Специальные системные вызовы для работы с критическими секциями.

- EnterCriticalSection
- LeaveCriticalSection

Порядок действий:

1. Выделить объект типа CriticalSection
2. Инициализировать объект критической секции функцией InitializeCriticalSection
3. Перед входом в критическую секцию вызвать функцию EnterCriticalSection
4. После завершения работу в критической секции вызвать LeaveCriticalSection
5. После того, как критическая секция становится ненужной вызвать DeleteCriticalSection

Можно войти в критическую секцию без блокировки потока функцией TryEnterCriticalSection.
Время которое поток в очереди ожидает освобождения ресурса можно найти в реестре по пути:

```
HKEY_LOCAL_MACHINE\System\CurrentControlSet\Control\SessionManager ->
CriticalSectionTimeout
```

Фактически этот параметр составляет 30 суток

7 Лекция от 10.12.2022

7.1 Синхронизация задач с помощью функций ожидания

Освобождение объекта — переход в сигнальное состояние.

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

Второй параметр часто задают как `INFINITE`.
Функция может вернуть одно из четырех значений:

- `WAIT_TIMEOUT` — Время ожидания вышло, объект не освободился.
- `WAIT_OBJECT_0` — Объект перешел в сигнальное состояние.
- `WAIT_FAILED` — Ошибка выполнения функции.
- `WAIT_ABANDONED` — только для мьютексов. Означает, что мьютекс освободился в следствии окончания выполнения владевшего им потока.

Функция:

```
DWORD WaitForMultipleObjects(
              DWORD   nCount,
              HANDLE  *lpHandles,
              BOOL    bWaitAll,
              DWORD   dwMilliseconds
);
```

Если bWaitAll установить в FALSE, функция завершает свою работу, когда хотя бы один из объектов переходит в сигнальное состояние. Если bWaitAll равен TRUE, то функция ожидает освобождения всех объектов.

Функция возвращает следующие значения:

Если bWaitAll == TRUE:

- WAIT_TIMEOUT — Время ожидания вышло, объект не освободился.
- WAIT_OBJECT_0 — Объект перешел в сигнальное состояние.
- WAIT_FAILED — Ошибка выполнения функции.
- WAIT_ABANDONED — только для мьютексов. Означает, что мьютекс освободился в следствии окончания выполнения владевшего им потока.

Если bWaitAll == FALSE:

- WAIT_TIMEOUT
- WAIT_FAILED
- [WAIT_OBJECT_0 ... WAIT_OBJECT_0 + nCount-1]
- [WAIT_ABANDONED_0 ... WAIT_ABANDONED_0 + nCount-1]

Чтобы получить индекс объекта надо из вернувшегося значения вычесть WAIT_OBWAIT_OBJECT_0.

В Рихтере посмотреть функцию SignalObjectandWait.

Одним из недостатков всех рассмотренных функций ожидания является то, что поток в котором вызывается любая из этих функций останавливается и не воспринимает поступающие сообщения. Сообщения поступают в очередь, но очередь не разгружается. Решение этой проблемы путем установки малого значения dwMilliseconds и зацикливание вызова, если функция вернула WAIT_TIMEOUT помогает слабо, так как сообщения все равно не успевают обработаться при циклической передачи управления.

Для решения этой проблемы есть функция

```
DWORD MsgWaitForMultipleObjects(
              DWORD   nCount,
              HANDLE  *lpHandles,
              BOOL    bWaitAll,
              DWORD   dwMilliseconds,
              DWORD   dwWakeMask
);
```

Эта функция завершает свою работу не только при наступлении сигнальных состояний объектов или по истечению времени ожидания, но и при поступлении сообщений, определенных маской dwWakeMask. Примеры масок:

- QS_ALLPOSTMESSAGE — функция будет реагировать на асинхронные сообщения отличные от событий аппаратного ввода.
- QS_SENDMESSAGE — функция будет реагировать на синхронные сообщения, которые отправляются другим потоком.

7.2 Синхронизация задач с помощью событий

Событие — Самые примитивные объекты ядра, а по сути своей простыми уведомлениями об окончании каких-либо действий. Событие содержит в своем дескрипторе содержит счетчик числа пользователей и две логические переменные, которые указывают тип события (с ручным сбросом и с автоматическим сбросом) и состояние (свободен или занят).

События с ручным сбросом можно перевести в несигнальное состояние функцией `ResetEvent`. События с автосбросом переводятся в несигнальное состояние функцией `ResetEvent` или функцией `WaitForSingleObject`. Если событие с автоматическим сбросом ожидает несколько потоков с помощью функции `WaitFoWaitForSingleObject`, то из состояния ожидания особождается только один из этих потоков.

```
HANDLE CreateEvent(
          LPSECURITY_ATTRIBUTES lpEventAttributes,
          BOOL                  bManualReset,
          BOOL                  bInitialState,
          LPCSTR                Name
)
```

Первый параметр объединяет в себе два параметра: наследование возвращенного дескриптора дочерними процессами и структура безопасности дескриптора. `bManualReset` устанавливает тип сброса. Если `TRUE`, то ручной, `FALSE` — автоматический. `bInitialState` устанавливает начальное состояние события, `TRUE` — сигнальное, `FALSE` — несигнальное. Чтобы проверить, что такое событие уже есть, надо выполниться следующий код:

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

Устанавливают состояния таймера следующий функции

```
BOOL ResetEvent(HANDLE hEvent);
BOOL SetEvent(HANDLE hEvent);
```

Функция

```
BOOL PulseEvent(HANDLE hEvent)
```

Если событие с ручным сбросом, эта функция позволяет всем потокам, которые ожидают этого события и могут немедленно завершить ожидание, выйти из состояния ожидания. Затем функция `PulseEvent` переводи событие в несигнальное состояние и завершает свою работу.

Для событий с автоматическим сбросом позволит только одному из ожидающих потоков выйти из состояния ожидания, если это возможно, после чего переводит событие в несигнальное состояние и завершает свою работу.

Функция

```
HANDLE OpenEvent(
        DWORD dwDesiredAccess,
        BOOL bInheritHandle,
        LPCSTR name
);
```

`dwDesiredAccess` может быть установлен в трех вариантах:

- `EVENT_ALL_ACCESS` — поток сможет применять к событию любые действия.
- `EVENT_MODIFY_STATE` — в потоке можно использовать функции только `SetEvent` и `ResetEvent`.
- `SYNCHRONIZE` — в потоке можно только использовать функции ожидания.

Пример

Поток 1 готовит данные, которые должны быть обработаны потоками 2 и 3. В потоке 1 размещаем следующий код:

```
HANDLE E[2];
E[0] = CreateEvent(NULL, TRUE, FALSE, "MyEvent2");
E[1] = CreateEvent(NULL, TRUE, FALSE, "MyEvent3");
HANDLE H = CreateEvent(NULL, TRUE, FALSE, "MyEvent1");
// Подготовка данных
...
// Окончание подготовки
SetEvent(H);
WaitForMultipleObjects(2, E, TRUE, INFINITE);
ResetEvent(E[0]);
ResetEvent(E[1]);
ResetEvent(H);
```

С помощью события MyEvent1 поток 1 будет оповещать потоки 2 и 3 об окончании подготовки данных. События из массива `E` будут использоваться для получения информации от потоков 2 и 3. После подготовки вызываем функцию `SetEvent`.

В потоке 2:

```
HANDLE H = CreateEvent(NULL, TRUE, FALSE, "MyEvent1");
WaitForSingleObject(H, INFINITE);
// Обработка данных
...
// Окончание обработки
SetEvent(OpenEvent(EVENT_ALL_ACCESS, TRUE, "MyEvent2"));
```

8 Лекция от 04.02.2023

8.1 Синхрониазция с помощью семафоров

У семафора больше двух состояний. Вводится максимальное число счетчика семафора, назовем его `c`, и вводится два элементарных действия (примитива).
`(Этапы компиляции: Лексический разбор → Синтаксический анализ (строим синтаксическое дерево) → Семантический)`
Две операции:

1. Увелечение счетчика `c` на заданную велечину (атомарная операция).

2. Уменьшение счетчика `c` на заданную велечину (атомарная операция). Если `c` равна 0, то сделать ее меньше нуля невозможно.

Пример:

Есть три принтера. Вводится семафор `c` счетчиком 3. Если `c` равно 0, то все принтеры заняты. Если `c` равна 1, то 1 принтер свободен... Каждый раз при обращениии к печати счетчик `c` уменьшается на 1.

Если семафор может принимать значения только 0 или 1, он приравнивается к блокирующей переменной.

При создании семафора для него задается максимальное число его счетчика.

Есть функция `ReleaseSemaphore`, которая позволяет увеличивать счетчик семафора.

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

При завершении любой функции ожидания, которая обращается к семафору, число в счетчике уменьшается на единичку. Пока `c` больше 0, семафор находится в сигнальном состоянии.

Объект семафор создается вызовом функции `CreateSemaphore`.

```
HANDLE CreateSemaphore(
        PSECURITY_ATTRIBUTE psa,
        LONG lInitialCount,
        LONG lMaximumCount,
        PCTSTR pszName );
```

- `lInitialCount` — Начальное значение счетчика.
- `lMaximumCount` — Максимальное значение счетчика.
- `pszName` — Имя объекта ядра.

```
CreateSemaphore(...);
if (GetLastError() == ERROR_ALREADY_EXISTS) {

}
```

Пример, как получить текущее значение счетчика семафора:

```
long N = MaximumCount; // Зачем в N заносить MaximumCount я не знаю
if (ReleaseSemaphore(s, 1, &N)) {
  WaitForSingleObject(s,1);
}
```

Пример работы с семфорами:
Поток 1 процесс A:

```
HADLE s = CreateSemaphore(NULL, MaximumCount, MaximumCount, "Sem1");
```

Поток 2 процесс B;

```
HANDLE s = OpenSemaphore(SEMAPHORE_ALL_ACCESS,
                         TRUE, "Sem1");
DWORD TimeOut = 2000;
...
// Запрос ресурса
DWORD Result = WaitForSingleObject(S, TimeOut);
if (Resul == WAIT_OBJECT_0) {
  // Действия с ресурсом
  ReleaseSemaphore(s, 1, NULL);
} else {
  // Вывод того, что происходит
}
```

8.2 Mutex

Чем mutex лучше предыдущих объектов — тем, что mutex запоминает поток, в котором вызывается. Mutex гарантирует потокам взаимоисключающий доступ к разделяемому ресурсу. В нем есть:

1. Счетчик числа пользователей.
2. Счетчик числа рекурсии.
3. ID потока.

Если ID потока равно 0, значит mutex находится в сигнальном состоянии. Иначе mutex захвачен одним потоком и находится в несигнальном состоянии.

Функции для работы с mutex.

Поток получает доступ к mutex, вызывая одну из функций ожидания с передачей ей дескриптора mutex. Если функция ожидания опрделяет, что ID потока не равен 0, вызывающий поток переходит в состояние ожидания. Когда ID потока обнуляется в дескриптор, записывается ID ожидающего потока и счетчику рекурсии присваивается 1.

Для mutex сделано одно исключение: если ID потока в дескрипторе mutex'а совпадает с ID потока, который вызвал функцию ожидания для этого mutex'а, то система выделяет потоку процессорное время, хотя mutex еще занят. Счетчик рекурсии содержит количество захватов одного mutex'а потоком. Когда ожидания mutex'а потоком успешно завершается, поток получает монопольный доступ к ресурсу, а все остальные потоки, которые запрашивают этот ресурс переходят в состояние ожидания.

Функция `BOOL ReleaseMutex(HANDLE hMutex)` уменьшает счетчик рекурсии на единицу. Если поток захватывал mutex n раз, то функция `ReleaseMutex` доллжна быть вызвана n раз.

Если какой-либо другй поток попытается вызвать функцию `ReleaseMutex`, то функция не сработает и вернет `FALSE`.

Если какой-либо поток завершается не освободив mutex, то считается, что произошел отказ от mutex, и система переводит его в сигнальное состояние.

Функция `OpenMutex` позволяет открыть mutex с именем `pszName`. fdwAccess может принимать два значения ...

9 Лекция от 11.02.2023

9.1 Pipe

pipe — разделяемые участки памяти (псевдофайлы), которые используются для передачи данных между разными потоками или процессами. Поток, который создает канал, называется сервером. Потоки, которые подключаются к созданному каналу, называются клиентами. Для передачи данных используются функции `ReadFile` и `WriteFile`. Каналы бывают:

- односторонними (полудуплексными);
- двусторонними (дуплексными);
- анонимными (безымянные);
- именованные.

Функция для создания канала:

```
BOOL CreatePipe(
      PHANDLE hReadHandle,
      PHANDLE hWriteHandle,
      SECURITY_ATTRIBUTES AttrStr,
      DWORD dwSize
      )
```

Для соединения клиентского потока с анонимным каналом необходимо передать клиенту один из дескрипторов анонимного пайпа. Передаваемый дескриптор должен быть наследуемым. Наследование дескрипторов анонимного канала определяется значением поля `bInheritHandle` в структуре `SECURITY_ATTRIBUTES`. Если значение этого поля равно `TRUE`, то дескрипторы создаваемого файла создаются наследуемыми. Если дескрипторы анонимного канала создаются наследуемыми, то тот дескриптор, который не передается клиенту, должен быть сделан ненаследуемым и наоборот. Для этого есть функция `DuplicateHandle`, которая создает копию дескриптора, но у этой копии меняется наследование.

```
BOOL DuplicateHandle(
        HANDLE hSourceProcessHandle,
        HANDLE hSourceHandle,
        HANDLE hTargetProcessHandle,
        PHANDLE hTargeHandle,
        DWORD dwDesiredAccess,
        BOOL bInheritHandle,
        DWORD dwOptions
)
```

Передача наследуемого дескриптора клиенту может выполняться одним из следующих способов:

1. через командную строку;

2. через поля (`hStdInput hStdOutput hStdError`);

При создании консольного (дочернего процесса) стандартные потоки ввода/вывода связываются с дескрипторами, которые заданы в полях (пункт 2), для передачи данных по анонимному каналу можно использовать функции стандартного ввода/вывода. Такая процедура называется перенаправлением стандартного ввода/вывода

Функции ввода вывода:

```
BOOL WriteFile(
      _In_ HANDLE hAnonPipe,
      _In_ LPCVOID Buffer
      _In_ DWORD dwNumberOfButesToWrite,
      _Out_ LPDWORD lpNumberOfBytesWritten,
      _OptIn_ LPOVERLAPPED lpOverLap
)

BOOL ReadFile(
      _In_ HANDLE hAnonPipe,
      _In_ LPCVOID Buffer
      _In_ DWORD dwNumberOfButesToRead,
      _Out_ LPDWORD lpNumberOfBytesReaded,
      _OptIn_ LPOVERLAPPED lpOverLap
)
```

Чтобы закрыть канал, нужно вызвать функцию `CloseHandle(...)`

Порядок работы с анонимным каналом:

Сервер:

1. Создает анонимный pipe с помощью функции `CreatePipe`.
2. Создает дубликат дескриптора, устанавливает нужное наследование с помощью функции `DuplicateHandle`.
3. Закрывает ненужный старый дескриптор.
4. Подготавливает входные параметры для функции `CreateProcess`.
5. Создает дочерний процесс с помощью функции `CreateProcess`, задавая `bInheritHandle` как TRUE, и задавая флаг создания `dwCreationFlags` как консольное приложение.
6. Закрывает дескрипторы нового процесса.
7. Выполняет чтение из анонимного канала.
8. Закрывает дескриптор чтения из канала.

Клиент:

1. Записывает данные в канал.
2. Закрывает дескриптор записи.

9.2 Синхронизация задач с помощью ожидаемых таймеров

Ожидаемый таймер — объект ядра, который самостоятельно переходит в сигнальное состояние в определенное время или через регулярные промежутки времени. Чтобы создать таймер необходимо вызвать функцию

```
HANDLE CreateWaitableTimer(
            LPSECURITY_ATTRIBUTES attributes,
            BOOL bManualReset,
            LPCSTR lpTimerName
)
```

10 Лекция от 4.03.2023

10.1 Непрерывное распределение виртуальной памяти

Простое непрерывное распределение — вся память распределяется на три части:

1. область ОС;
2. область выполняемой задачи;
3. освободная бласть памяти.