**Лаб1**

Операционные системы осуществляют две значительно отличающиеся друг от друга **функции:**

1. предоставлять пользователю некую расширенную виртуальную машину, с которой  
   легче работать (легче программировать), чем непосредственно с аппаратурой реального  
   компьютера или реальной сети;
2. управлять ресурсами вычислительной системы

**Операционная система (ОС)** — комплекс системных и управляющих программ, предназначенных для наиболее эффективного использования всех ресурсов вычислительной системы (ВС) и удобства работы с ней.

ОС = расширенная виртуальная машина = система управления ресурсами

Вычислительная система — взаимосвязанная совокупность аппаратных средств вычислительной техники и программного обеспечения, предназначенная для обработки информации

Именно прикладные программы работают с ос и ее абстракциями

Конечные пользователи работают с абстракциями, предоставленные пользовательским интерфейсом

ОС управляет всеми устройствами компьютерной системы (процессорами, оперативной  
памятью, дисками, клавиатурой, монитором, принтерами, сетевыми устройствами и др.)  
и обеспечивает пользователя удобным интерфейсом для работы с аппаратурой.

**Подсистемы ОС:**

* управления процессами,
* управления памятью,
* управления файлами,
* управления внешними устройствами,
* пользовательского интерфейса,
* защиты данных и администрирования

**Функции ОС:**

* прием от пользователя (оператора) заданий или команд, сформулированных на соответствующих языках, и их обработка;
* загрузка в ОП программ и их исполнение;
* инициация программы (передача ей управления);
* прием и исполнение программных запросов на запуск, приостановку, остановку других программ;
* организация взаимодействия между задачами;
* идентификация всех программ и данных;
* обеспечение работы системы управления файлами и/или систем управления БД;
* обеспечение режима мультипрограммирования (многозадачности);
* планирование и диспетчеризация задач;
* обеспечение функций по организации и управлению операциями ввода/вывода;
* удовлетворение жестким ограничениям на время ответа в режиме реального
* времени (для соответствующих ОС);
* управление памятью, организация виртуальной памяти;
* организация механизмов обмена сообщениями и данными между
* выполняющимися программами;
* защита одной программы от влияния другой;
* обеспечение сохранности данных;
* аутентификация, авторизация и другие средства обеспечения безопасности;
* предоставление услуг на случай частичного сбоя системы;
* обеспечение работы систем программирования;
* параллельное исполнение нескольких задач.

**Классификации ОС:**

В зависимости от способа организации вычислений:

* Системы пакетной обработки – основной задачей является организация наибольшего количества вычислительных процессов за единицу времени. Определенные процессы объединяются в пакет, который затем обрабатывает ОС.
* Системы разделения времени – создание возможности единовременного взаимодействия с устройством сразу несколькими людьми. В порядке очереди каждый пользователь получает определенный промежуток процессорного времени.
* Системы реального времени – организация работы каждой задачи за определенный промежуток времени, присущий каждой конкретной задаче.

В зависимости от типа ядра:

* OС с монолитным ядром;
* OС с микроядром;
* OС с гибридным ядром.

В зависимости от количества единовременно решаемых задач:

* однозадачные;
* многозадачные;

В зависимости от количества пользователей:

* однопользовательские;
* многопользовательские.

В зависимости от количества поддерживаемых процессоров:

* однопроцессорные
* многопроцессорные

В зависимости от возможности работы в компьютерной сети:

* локальные – автономные ОС, которые не позволяют работать с компьютерными
* сетями;
* сетевые – ОС с поддержкой компьютерных сетей.

В зависимости от роли в сетевом взаимодействии:

* серверные – ОС, открывающие доступ к ресурсам сети и осуществляющие управление сетевой инфраструктурой;
* клиентские – ОС, которые имеют возможность получения доступа к ресурсам сети.

В зависимости от типа лицензии:

* открытые – ОС с открытым исходным кодом, который можно изучать и редактировать;
* проприетарные – ОС, связанные с определенным правообладателем и, как правило, имеющие закрытый исходный код.

В зависимости от сферы использования:

* ОС мэйнфреймов – больших компьютеров;
* ОС серверов;
* ОС персональных компьютеров;
* OC мобильных устройств;
* встроенные OC;
* OC маршрутизаторов

**Основные этапы развития ОС, поколения ОС**

Первый настоящий цифровой компьютер был изобретен английским математиком Чарльзом Бэббиджем, но он был механическим и так и не заработал должным образом.

**Первое поколение (1945-1955 гг.)**

Первое поколение компьютеров строилось на электронных лампах, программы писались в машинных кодах и загружались вручную или с перфокарт. Системы выполняли только одну операцию одновременно, не имели языков программирования и операционных систем. Электронные лампы часто выходили из строя, что исключало необходимость в постоянно работающем ПО. Позднее появились первые системные программы, такие как компилятор Fortran и ассемблер для IBM-701. Период характеризуется высокой стоимостью, малым числом компьютеров и низкой эффективностью, с большим временем на подготовку и последовательным выполнением программ.

**Второе поколение (1955-1965 гг.)**

Второе поколение компьютеров характеризуется использованием транзисторов, что повысило их надёжность и увеличило время работы. Для уменьшения непроизводительных расходов машинного времени стали использовать небольшие компьютеры для записи заданий на магнитную ленту, а для вычислений — мощные машины. Программа, запускавшая задания с ленты, стала прообразом операционной системы. Ввод заданий осуществлялся с помощью специальных перфокарт ($JOB, $LOAD, $RUN, $END), прототипов команд CLI. Основными языками программирования были Fortran и ассемблер, а типичными операционными системами — FMS и IBSYS. Появились новые языки программирования (LISP, COBOL и др.), компиляторы и библиотеки, что упростило программирование и разделило персонал на программистов, операторов и других специалистов.

**Третье поколение (1965-1980 гг.)**

Компьютеры третьего поколения использовали малые интегральные схемы, что сделало их надёжнее и дешевле по сравнению с машинами второго поколения. Повысилась производительность процессоров, а вычислительные задачи стали сложнее и разнообразнее. Главным достижением стала многозадачность: память делилась на разделы, позволяя выполнять несколько заданий одновременно, что решало проблему простоя процессора во время операций ввода-вывода. Важной инновацией стала подкачка данных (спулинг), позволяющая загружать задания с диска по мере их поступления, что устраняло необходимость в переносах магнитных лент. Многозадачность требовала аппаратной и программной поддержки, включая защитные механизмы, системные вызовы, управление памятью и синхронизацию программ. Хотя ОС третьего поколения хорошо справлялись с научными и коммерческими вычислениями, они оставались разновидностью систем пакетной обработки, и задержка между вводом задания и получением результата могла составлять несколько часов. Желание сократить время ожидания привело к разработке режима разделения времени, позволяющего интерактивную работу с несколькими пользователями через терминалы. Также появилась идея создания семейств совместимых машин под управлением одной ОС, что привело к созданию мощных, но сложных систем, таких как OS/360, CTSS и MULTICS. Упрощённая версия MULTICS, UNIX, стала популярной в академических и коммерческих кругах.

**Четвертое поколение (1980-2005 гг.)**

Разработка больших интегральных схем (БИС), содержащих тысячи транзисторов на одном квадратном сантиметре, привела к созданию микрокомпьютеров, где весь процессор размещался в одной микросхеме. Это снизило размеры и стоимость компьютеров, что позволило каждому человеку приобрести персональный компьютер (ПК). Операционные системы для микрокомпьютеров относятся к четвертому поколению. Первой ОС для микропроцессора Intel 8080 стала CP/M, а для Intel 8086 — MS-DOS. Изначально ПК работали в однопрограммном режиме без защиты файлов и памяти, но с ростом сложности задач их архитектура вновь стала походить на большие вычислительные системы. Сначала ПК использовали текстовый режим, позже появился графический интерфейс, что привело к появлению таких ОС, как Mac OS X, Windows и UNIX-подобных систем. В середине 1980-х годов начали активно развиваться компьютерные сети и распределенные ОС, которые обеспечивали масштабируемость и поддерживали распределенную обработку данных.

**Пятое поколение (2005 г. - по н.в.)**

Хотя идея объединения телефона и компьютера появилась ещё в 1970-х годах, первый настоящий смартфон появился в середине 1990-х годов с выходом Nokia N9000 — комбинации телефона и КПК. В 1997 году Ericsson ввела термин "смартфон" для GS88 "Penelope". Радикальные изменения начались в 2005 году с появлением многоядерных процессоров и адаптацией операционных систем Windows и Linux для многоядерных архитектур. На рынке мобильных устройств развернулась конкуренция между ОС. Доминирующей стала Google Android, за ней следовала Apple iOS, но ситуация остаётся динамичной.

В первые годы большинство смартфонов работало на Symbian OS, но со временем её вытеснили другие системы, такие как RIM Blackberry OS (2002) и Apple iOS (2007). В 2011 году Nokia отказалась от Symbian в пользу Windows Phone. Однако Android, выпущенная Google в 2008 году, быстро обошла конкурентов благодаря открытым исходным кодам и разрешительной лицензии, что позволило производителям адаптировать её под своё оборудование. Разработчики также активно создавали приложения для Android на Java. Несмотря на доминирование Android, конкуренты продолжают бороться за долю рынка.

Внедрение многоядерных процессоров привело к созданию мощных вычислительных систем, используемых для высокопроизводительных вычислений в кластерной архитектуре. Виртуализация стала популярной технологией, позволяющей абстрагировать вычислительные ресурсы от аппаратной реализации. Она используется в облачных технологиях, обеспечивая изоляцию ресурсов и возможность их совместного использования клиентами в дата-центрах.

**Системы реального времени**

Системы реального времени характеризуются тем, что для них время является ключевым параметром. В таких системах, как управление производственными процессами, компьютеры должны собирать данные и управлять оборудованием в строго заданные моменты. Например, на сборочном конвейере операции должны выполняться точно в определённое время — иначе продукция может быть испорчена. Такие системы называются системами жёсткого реального времени, так как они требуют строгого соблюдения временных ограничений. Они применяются в управлении производством, авиационно-космическом оборудовании, военной технике и других критически важных областях.

Системы мягкого реального времени допускают небольшие задержки в выполнении задач, которые не приводят к серьёзным последствиям. Примеры таких систем включают цифровые аудио- и мультимедийные приложения, а также смартфоны. В этих системах временные требования менее жёсткие, и небольшие отклонения не приводят к фатальным ошибкам.

**POSIX**

Для создания программ, работающих на любой UNIX-системе, Институт инженеров по электротехнике и электронике (IEEE) разработал стандарт UNIX, названный POSIX. Этот стандарт определяет минимальный интерфейс системных вызовов, которые должны поддерживать совместимые системы UNIX, и поддерживается большинством версий UNIX, а также рядом других операционных систем.

POSIX (Portable Operating System Interface) — это набор стандартов, описывающих интерфейсы между операционной системой и прикладной программой, библиотеку языка C и набор приложений с их интерфейсами. Он был создан для обеспечения совместимости UNIX-подобных операционных систем и переносимости приложений на уровне исходного кода, но применим и к не-UNIX системам.

Стандарты POSIX разработаны комитетом IEEE 1003, а Международная организация по стандартизации (ISO) и Международная электротехническая комиссия (IEC) приняли их как ISO/IEC 9945. POSIX является основой для стандартов Single UNIX Specification, которые определяют реализацию интерфейса и позволяют операционным системам использовать торговую марку UNIX.

**Эксплуатационные требования к ОС**

* способность выполнения основных функций: эффективного управления ресурсами и обеспечения удобного интерфейса для пользователя и прикладных программ.
* должна реализовывать мультипрограммную обработку, виртуальную память, поддерживать многооконный интерфейс, а также выполнять многие другие совершенно необходимые функции.
* расширяемость. Код операционной системы должен быть написан таким образом, чтобы можно было легко внести дополнения и изменения, если это потребуется, и не нарушить целостность системы;
* переносимость. Код операционной системы должен легко переноситься с процессора одного типа на процессор другого типа и с аппаратной платформы (которая включает наряду с типом процессора и способ организации всей аппаратуры компьютера) одного типа на аппаратную платформу другого типа;
* надежность и отказоустойчивость. Система должна быть защищена как от внутренних, так и от внешних ошибок, сбоев и отказов. Ее действия должны быть всегда предсказуемыми, а приложения не должны наносить вред операционной системе;
* совместимость. Операционная система должна иметь средства для выполнения прикладных программ, написанных для других операционных систем. Кроме того, пользовательский интерфейс должен быть совместим с существующими системами и стандартами;
* безопасность. Операционная система должна обладать средствами защиты ресурсов одних пользователей от других;
* производительность. Система должна обладать настолько хорошим быстродействием и временем реакции, насколько это позволяет аппаратная платформа.

Современные операционные системы уделяют особое внимание безопасности из-за высокой ценности обрабатываемой информации и увеличения уровня угроз при передаче данных по сетям, особенно в Интернете. Они включают шифрацию данных, аутентификацию и авторизацию. Кроме того, ОС стали многоплатформенными и удобными для пользователя благодаря развитым графическим интерфейсам и интеллектуальному взаимодействию.

Для сетевых операционных систем целью является высокая прозрачность сетевых ресурсов и организация распределенных вычислений, превращающих сеть в виртуальный компьютер.

\*\*Квантовые вычисления\*\* — перспективное направление развития компьютеров, основанное на квантовой механике. Они используют кубиты, которые могут находиться в состоянии одновременно 0 и 1, в отличие от классических битов. Квантовые компьютеры обладают возможностью выполнения задач, недоступных классическим компьютерам.

\*\*Недостатки\*\* квантовых компьютеров включают нестабильность кубитов, чувствительность к тепловому шуму, вероятность ошибок в расчетах и необходимость охлаждения вычислительных ячеек жидким азотом.

\*\*Перспективы\*\*: Квантовые технологии активно развиваются по всему миру. Япония, США и крупные корпорации, такие как Google, IBM, и Microsoft, вкладывают значительные средства в их разработку, планируя создание коммерческих квантовых компьютеров к 2029–2030 годам.

|  |  |
| --- | --- |
| PATH | Переменная окружения представляет собой список директорий, в которых операционная система ищет исполняемые файлы. Когда вы вводите команду в командной строке (cmd.exe), Windows проверяет наличие соответствующего исполняемого файла в директориях, перечисленных в переменной PATH, в том порядке, в котором они указаны.  Переменная PATH позволяет запускать программы и утилиты из командной строки без необходимости указывать полный путь к файлу.Например, команда notepad работает без указания полного пути, так как исполняемый файл notepad.exe находится в одной из директорий, перечисленных в PATH. |

**BAT-файлы**

Файл с расширением .bat – это исполняемый файл Windows. По сути это обыкновенный текстовый файл, в котором записаны разные команды. Они используются для повторяющихся задач, или для осуществления последовательного запуска групп сценариев.

Файлы bat часто также называют пакетными или командными, кроме того вы можете встретить следующие их обозначения: исполняемые сценарии, пакетные программы, а также сценарии оболочки. Расширение .bat нередко заменяется .CMD.

Практически в любом ПО применяются файлы bat для всевозможных целей, в том числе создания копий или удаления файлов, начала или завершения работы процессов.

В 1-ой сроке ("@echo off") префикс '@ ' означает, что команда не будет выведена на экран(попробуйте его убрать :-) ), сама же команда (echo) выводит текст следующий за ней(см. строку 2), НО, если подать строку "off", все последующие команды будут действовать так, как будто перед ними стоит префикс '@', "echo on", соответственно, выключает этот режим. Чтобы вывести символ "%", его нужно дублировать, потом поймете почему :-), для вывода пустой строки поставьте "." после "echo"

@echo off отключает вывод команд в консоль.

echo Привет, мир! выводит сообщение.

pause останавливает выполнение, пока пользователь не нажмет любую клавишу.

В бат-файле нулевой параметр (%0) представляет собой имя самого бат-файла или команду, с помощью которой он был вызван.

В командной строке Windows в бат-файле можно передать до **9 позиционных параметров** напрямую: %1, %2, %3, ..., %9. Эти параметры соответствуют аргументам, переданным в скрипт при его запуске.

if "%~1"=="": Проверяет, был ли передан параметр в BAT-файл. %~1 — это первый параметр, переданный BAT-файлу. Если параметр отсутствует (т.е., строка пуста), то выполняется блок кода в скобках.

echo Не указан параметр: Выводит сообщение, что параметр не был указан.

pause: Приостанавливает выполнение программы и ждет, пока пользователь нажмет любую клавишу, чтобы продолжить. Это позволяет пользователю увидеть сообщение.

exit /b 1: Завершает выполнение BAT-файла с кодом возврата 1, что обычно указывает на ошибку.

**Лаб2**

**Задание 1**

|  |  |
| --- | --- |
| pwd | Корневая папка ( в какой директории вы сейчас находитесь) |
| ls | Название всех папок в текущей директории |
| Cd | Команда для навигации с указанием пути |
| Cd ~ | Переход в основную папку |
| Cd ../ | Перейти на уровень выше |
| Cd ~/ | Перейти в другую папку |
| clear | Скрыть предыдущие команды |
| Команда --help |  |
| touch | Создать файл |
| mv | Переместить файл |
| cp | Копировать |
| man | отображает справочную информацию для команды |
| echo | Вывод текста или переменной на экран |
| mkdir | Создание нового каталога |
| rmdir | Удаление пустого каталога |
| rm | Удаление файлов или каталогов |
|  |  |
| su | Переключение на другого пользователя |
| whoami | Отображение имени текущего пользователя |
| cat | Вывод содержимого файла на экран |
| less | Постраничный просмотр содержимого файла |
| more | Постраничный просмотр файла (аналог less, но проще) |
| head | Вывод первых строк файла |
| tail | Вывод последних строк файла |
| man | Отображение справки по команде |
| whereis | Поиск местоположения команды или программы |
| whatis | Краткое описание команды |
| apropos | Поиск команд по описанию |
| lpstat | Отображение статуса очереди печати |
| lpr | Отправка задания на печать |
| lpq | Отображение состояния очереди печати |
| lprm | Удаление задания из очереди печати |
| chgrp | Изменение группы файла |
| chown | Изменение владельца файла |
| chmod | Изменение прав доступа к файлу |
| zip | Сжатие файлов в архив ZIP |
| gzip | Сжатие файлов с использованием GZIP |
| gunzip | Разархивирование файлов, сжатых с помощью GZIP |
| bzip2 | Сжатие файлов с использованием BZIP2 |
| bunzip2 | Разархивирование файлов, сжатых с помощью BZIP2 |
| tar | Архивирование и распаковка файлов в формате TAR |
| zcat | Просмотр содержимого файлов, сжатых с GZIP |
| bzcat | Просмотр содержимого файлов, сжатых с BZIP2 |
| locate | Поиск файлов по имени |
| grep | Поиск текста по шаблону в файлах |
| find | Поиск файлов по критериям |
| history | Просмотр истории команд |
| alias | Создание псевдонимов для команд |
| unalias | Удаление псевдонима для команды |
| ps | Просмотр списка процессов |
| top | Мониторинг процессов в реальном времени |
| lsof | Просмотр открытых файлов и сокетов |
| free | Отображение состояния памяти |
| df | Отображение свободного пространства на диске |
| du | Отображение размера каталогов и файлов |
| yum | Пакетный менеджер для дистрибутивов на основе Red Hat |
| apt | Пакетный менеджер для дистрибутивов на основе Debian |
| ifconfig | Настройка сетевых интерфейсов |
| ping | Проверка доступности хоста через сеть |
| traceroute | Просмотр маршрута пакетов до хоста |
| host | Информация о DNS имени или IP-адресе |
| iwconfig | Настройка беспроводных сетевых интерфейсов |
| dhclient | Запрос IP-адреса через DHCP |
| ifup | Включение сетевого интерфейса |
| ifdown | Отключение сетевого интерфейса |
| route | Просмотр или изменение таблицы маршрутизации |
| dig | Инструмент для работы с DNS |
| ss | Отображение сокетов и подключений |
| ssh | Подключение к удаленному серверу через SSH |
| sftp | Передача файлов через SSH с использованием SFTP |
| scp | Копирование файлов между хостами через SSH |
| rsync | Синхронизация файлов и каталогов |
| wget | Загрузка файлов из сети |
| curl | Отправка HTTP-запросов и загрузка данных из сети |

**Задание 2**

|  |  |
| --- | --- |
| Переменная окружения | Краткое описание |
| $HOME | Хранит путь к домашнему каталогу текущего пользователя  cd $HOME |
| $PATH | Содержит список директорий, в которых оболочка ищет исполняемые файлы, когда вы вводите команду  echo $PATH |
| $PS1 | Определяет основной вид приглашения командной строки (prompt) в оболочке.  export PS1='\u@\h:\w\$ ' — настройка приглашения для bash, которое отображает имя пользователя (\u), имя хоста (\h), текущий рабочий каталог (\w) и символ $ или #, если вы работаете от имени суперпользователя |
| $PS2 | Определяет дополнительное приглашение командной строки, которое появляется, когда команда продолжается (например, при многострочных командах).  В bash, по умолчанию $PS2 установлен на >, что указывает, что команда продолжается и ожидает дальнейшего ввода |
| $IFS | Содержит разделители полей, используемые для разбиения строки на части при чтении ввода или при работе с переменными. По умолчанию $IFS содержит пробел, табуляцию и новую строку. Если изменить $IFS на запятую, IFS=,, команда read будет использовать запятые как разделители при чтении данных. |

**3 Задание**

Nano

Vim

**4 Задание**

**ls –la**

конвейер (pipe)

cut

grep

tr

sort

sed

|  |  |  |
| --- | --- | --- |
| pk | Bla bla | datatype |

**Задание 5**

Указание интерпретатора обеспечивает совместимость с различными системами. Скрипт, начинающийся с #!/bin/bash, будет выполнен с Bash, независимо от того, какой интерпретатор используется по умолчанию в системе.

**Задание 6**

1. Приведите примеры ситуации, когда удобно использовать опцию –f в команде tail

Опция -f в команде tail используется для отображения новых строк, добавляемых к файлу в реальном времени. Это особенно полезно в различных ситуациях, когда необходимо следить за изменениями в файле по мере их появления. Например вы управляете сервером и хотите наблюдать за логами в реальном времени, чтобы отслеживать работу приложений или искать ошибки.

1. Как выполнять поиск в утилите less?
2. Чем отличается less от more?
3. В чем разница между man, apropos и whatis?
4. Какая утилита обеспечивает лучшее сжатие текстовых  
   файлов: gzip или bzip2?
5. Что такое shebang?
6. Какой будет эффект от удаления французского языка из  
   системы командой **rm –fr /**? Вопрос теоретический, на  
   реальных системах команду не выполнять.

**Что такое процесс?**

**Процесс** — это просто экземпляр выполняемой программы, включая текущие значения счетчика команд, регистров и переменных.

**Что такое контекст процесса?**

Контекст процесса — это совокупность всей информации, необходимой операционной системе для корректного управления процессом, его выполнения и последующего возобновления с того места, где оно было приостановлено.

Информация, хранимая в блоке управления процессом, делится на две части: регистровый контекст и системный контекст.

* Регистровым контекстом процесса называется содержимое всех регистров общего назначения процессора (включая значение программного счетчика).
* Содержимое остальных регистров процессора называется системным контекстом процесса.
* Пользовательским контекстом называются данные, находящиеся в адресном пространстве процесса

**Что такое родительский и дочерний процесс?**

**Родительский процесс** — это процесс, который создает другой процесс. В операционных системах каждый процесс может порождать новые процессы, и тот процесс, который создает другой, называется "родительским".

**Дочерний процесс** — это процесс, который был создан (или порожден) другим процессом. Он является "потомком" родительского процесса.

Процессы организованы иерархически. Каждый процесс имеет родительский процесс. Процессы, созданные данным родителем называются дочерними процессами. Дочерний наследует многие из атрибутов родительского процесса. Новые процессы создаются вызовом int pid=fork(), который создаёт точную копию вызвавшего его процесса. Пара процессов называются "родительский" и "дочерний" и отличаются друг от друга тремя значениями:

* уникальный идентификатор процесса *PID*
* идентификатор родительского процесса *PPID*
* значение, возвращаемое вызовом fork(). В родительском это *PID* дочернего процесса или ошибка (-1), в дочернем fork() всегда возвращает 0.

После создания, дочерний процесс может загрузить в свою память новую программу (код и данные) из исполняемого файла вызовом execve(const char \*filename, char \*const argv [], char \*const envp[]);

Дочерний процесс связан с родительским значением *PPID*. В случае завершения родительского процесса *PPID* меняется на особое значение 1 - *PID* процесса *init*.

**Что такое процесс инициализации OS?**

**Процесс инициализации операционной системы** — это особый процесс, который запускается сразу после загрузки ядра операционной системы и является основным для управления другими процессами в системе. Он играет ключевую роль в старте системы и управлении жизненным циклом других процессов.

В зависимости от операционной системы процесс инициализации может называться по-разному: В системах на основе UNIX/Linux этот процесс называется init.

**Основные функции процесса инициализации:**

* После завершения загрузки ядра, процесс инициализации запускает все важные системные процессы и службы, такие как демоны (фоновые процессы) и пользовательские приложения.
* Он следит за дочерними процессами, порожденными системой, и при необходимости управляет их завершением. Если какой-либо процесс завершается, процесс инициализации может перехватывать информацию о его завершении (особенно для "зомби-процессов").
* Процесс инициализации запускает основные системные службы и демоны, которые выполняются в фоновом режиме, такие как серверы баз данных, веб-серверы, сетевые службы и другие компоненты.
* В случае необходимости процесс инициализации может выполнять перезапуск или остановку определенных служб.
* Он также управляет процессами завершения работы системы, например, правильной остановкой всех служб и безопасным завершением ОС.

**Перечислите области памяти процесса и поясните их назначение.**

**Сегмент кода** : Содержит машинные инструкции, которые выполняются процессором. Это сама программа в скомпилированном виде.Может быть общей для нескольких процессов, если они выполняют одну и ту же программу, что позволяет экономить память (например, библиотеки).

**Сегмент данных** : Хранит глобальные и статические переменные программы, которые инициализируются в начале работы программы.Разделяется на **инициализированные данные** (данные, которым присвоены значения) и **неинициализированные данные (BSS)** (глобальные переменные, объявленные, но не инициализированные).

**Куча (Heap):** Используется для динамического выделения памяти во время выполнения программы. Программист может вручную выделять и освобождать память в этой области (например, с помощью функций malloc() и free() в C или оператора new в C++).Гибкая по размеру, может увеличиваться или уменьшаться в зависимости от необходимости.

**Стек (Stack):** Используется для хранения данных вызовов функций и локальных переменных. Каждый раз, когда вызывается новая функция, в стек добавляется новый фрейм (стековый фрейм) с информацией о параметрах функции, локальных переменных и адресом возврата.Управляется автоматически: когда функция завершает работу, её фрейм стека удаляется.

**Сегмент загрузки динамических библиотек** : Хранит загруженные динамически подключаемые библиотеки (например, .dll или .so файлы). Библиотеки могут загружаться по требованию во время выполнения программы, а не в момент её старта. Может быть общей между процессами для экономии памяти, если одна и та же библиотека используется несколькими процессами.

**Сегмент командной строки и переменных окружения:** Эта область хранит переданные процессу параметры командной строки (аргументы) и переменные окружения, которые могут влиять на поведение программы.

**Пример взаимодействия областей памяти:**

Когда запускается программа:

* Инструкции программы загружаются в **сегмент кода**.
* Глобальные переменные попадают в **сегмент данных**.
* Когда функция вызывается, её локальные переменные размещаются в **стеке**.
* Если программа выделяет память динамически (например, для создания объекта или массива), эта память берётся из **кучи**.

**Чем отличаются системные процессы от пользовательских?**

Процессы, которые выполняют системный код, называются **системными** и применяются к системе в целом. Они занимаются выполнением таких служебных задач, как распределение памяти, обмен страницами между внутренним и вспомогательным запоминающими устройствами, контроль устройств и т.п. Они также выполняют некоторые задачи «по поручению» пользовательских процессов, например, делают запросы на ввод-вывод данных, выделяют память и т.д.

**Пользовательские** процессы выполняют собственный код и иногда обращаются к системным функциям. Выполняя собственный код, пользовательский процесс пребывает в пользовательском режиме (user mode). В пользовательском режиме процесс не может выполнять определенные привилегированные машинные команды. При вызове системных функций (например, read(), write() или open()) пользовательский процесс выполняет инструкции операционной системы. При этом пользовательский процесс «удерживает» процессор до тех пор, пока не будет выполнен системный вызов. Для выполнения системного вызова процессор обращается к ядру операционной системы. В это время о пользовательском процессе говорят, что он пребывает в привилегированном режиме, или режиме ядра (kernel mode), и не может быть выгружен никаким другим пользовательским процессом.

 **Назначение**:

* **Системные процессы**: Управляют важными функциями ОС (ядро, драйверы, службы).
* **Пользовательские процессы**: Выполняют пользовательские программы и приложения.

 **Уровень привилегий**:

* **Системные**: Имеют высокий уровень привилегий (доступ к аппаратуре и системным ресурсам).
* **Пользовательские**: Ограничены в доступе к критическим системным ресурсам.

 **Инициация**:

* **Системные**: Запускаются при старте ОС.
* **Пользовательские**: Запускаются пользователем или по его запросу.

 **Прерывания**:

* **Системные**: Могут выполнять задачи с высоким приоритетом и прерывать пользовательские процессы.
* **Пользовательские**: Могут быть прерваны системными для выполнения критических задач.

**Что такое Windows-сервисы, Linux-демоны?**

**Windows-сервисы -** Это специализированные **фоновые процессы**, работающие в Windows.

* Запускаются автоматически при старте системы или по требованию.
* Могут работать в фоновом режиме без необходимости входа пользователя в систему.
* Управляются через диспетчер служб (Services.msc), где можно их запускать, останавливать или перезагружать.
* Примеры: службы обновлений, службы печати, сетевые службы.

**Linux-демоны -** Это **фоновые процессы** в Linux и UNIX-подобных системах.

* Обычно запускаются при старте системы и работают непрерывно.
* Их имена часто оканчиваются на "d" (например, sshd, httpd), что указывает на их фоновую природу.
* Управляются через системы инициализации, такие как **systemd**, **Upstart**, или **SysV init**.
* Примеры: демоны SSH (sshd), веб-серверы (httpd), планировщики заданий (cron).

**С помощью каких системных вызовов можно создать дочерний процесс в Windows? Поясните разницу.**

В Windows для создания дочернего процесса используются следующие основные системные вызовы:

**1. CreateProcess()**

* Это основной и наиболее гибкий системный вызов для создания нового процесса в Windows.
* **Функция**: CreateProcess() создает новый процесс и его начальный поток. Родительский процесс может передать дочернему процессу начальные аргументы, задать атрибуты среды и указать, будет ли дочерний процесс унаследовать дескрипторы ресурсов.
* **Ключевые параметры**:
  + Путь к исполняемому файлу или командной строке.
  + Наследуемая среда и параметры безопасности.
  + Можно указать, будет ли дочерний процесс запускаться в фоновом режиме или на переднем плане.
* **Пример**:
  + CreateProcess("C:\\Program Files\\myapp.exe", NULL, ...).
* **Особенность**: Этот вызов предоставляет полный контроль над процессом и его контекстом. Родительский процесс может настроить дескрипторы ввода-вывода и назначить дочернему процессу различные параметры.

**2. CreateProcessAsUser()**

* Позволяет создавать процесс от имени другого пользователя, обычно используется для запуска процессов с повышенными правами.
* **Функция**: Этот вызов требует указания токена пользователя, от имени которого нужно запустить процесс.
* **Пример**:
  + Используется в случае, когда необходимо запустить процесс с привилегиями другого пользователя (например, от имени администратора).
* **Особенность**: Требует токен доступа пользователя, что делает его подходящим для создания процессов с особыми правами.

**3. CreateProcessWithLogonW()**

* Позволяет создать процесс, указывая имя пользователя, пароль и домен, под которыми процесс должен быть запущен.
* **Функция**: Предназначена для создания процессов от имени другого пользователя с явной передачей его учетных данных (логин и пароль).
* **Пример**:
  + Может использоваться для создания процессов с правами администратора при вводе учетных данных.
* **Особенность**: Это удобный способ для запуска процессов с другой учетной записью без предварительного получения токена, как в случае с CreateProcessAsUser().

**Основные различия:**

* **CreateProcess()** — базовый и наиболее универсальный вызов, используется для создания процесса в текущем контексте пользователя.
* **CreateProcessAsUser()** — позволяет запустить процесс с привилегиями другого пользователя при наличии его токена.
* **CreateProcessWithLogonW()** — позволяет запускать процесс под другим пользователем с вводом учетных данных, не требуя токена.

Все три вызова обеспечивают гибкие возможности для управления процессами в Windows в зависимости от уровня привилегий и необходимости работы с учетными записями пользователей.

**С помощью каких системных вызовов можно создать дочерний процесс в Linux? Поясните разницу.**

Linux для создания дочернего процесса используются несколько системных вызовов, каждый из которых имеет свои особенности. Основные вызовы для создания дочерних процессов:

**1. fork()** : Создает точную копию текущего процесса (родителя), включая его память, дескрипторы файлов, и т.д. Новый процесс (дочерний) получает уникальный идентификатор процесса (PID), но наследует код, данные и состояние родительского процесса.

* + Возвращает 0 в дочернем процессе.
  + Возвращает PID дочернего процесса в родительском процессе.
* **Особенность**: fork() создает **полную копию** адресного пространства родительского процесса. Это может быть неэффективно для процессов с большим объемом данных. Однако в современных системах часто используется механизм **copy-on-write**, что делает процесс копирования более оптимизированным (копируются только измененные страницы памяти).

**2. vfork()** : Похож на fork(), но предназначен для оптимизации производительности. Он создаёт дочерний процесс **без копирования** адресного пространства родителя. Дочерний процесс использует адресное пространство родителя до вызова exec() или завершения.

* **Как работает**:
  + Дочерний процесс работает в пространстве родителя, и до тех пор, пока он не вызовет exec() или не завершится, родительский процесс **приостанавливается**.
* **Особенность**: Быстрее, чем fork(), но требует осторожности, так как любые изменения в адресном пространстве дочернего процесса изменяют память родительского.

**3. clone()** : Более гибкий системный вызов для создания дочернего процесса с возможностью указания, какие ресурсы процесса будут общими между родителем и дочерним процессом (например, память, файловые дескрипторы, сигналы).

* **Как работает**:
  + Параметры вызова позволяют настроить, что будет разделено (например, CLONE\_VM — общая память, CLONE\_FS — общая файловая система).
* **Особенность**: Используется для создания потоков и более тонкого контроля над поведением дочернего процесса. Именно на основе clone() реализованы потоки (т.н. pthread).

**4. exec() (и его семейство функций execve, execl, execp, и т.д.)**

* **Функция**: Не создает процесс, но используется для **замены** текущего процесса (после вызова fork() или vfork()) новым процессом. Процесс продолжает работу с новыми инструкциями программы, заменяя свое содержимое новой программой.
* **Как работает**: Выполняет новый исполняемый файл в текущем процессе, заменяя его код, данные и сегмент памяти.
* **Особенность**: Обычно вызывается после fork() для того, чтобы новый дочерний процесс начал выполнение другой программы.
* **system()** выполняет команду, переданную в качестве аргумента, через оболочку (например, /bin/sh), как если бы вы её ввели в командной строке. Вызов system() запускает команду в **новом процессе**, ожидает её завершения, а затем возвращает управление родительскому процессу.
* Внутренне system() использует комбинацию системных вызовов fork() и exec(). То есть он сначала создаёт новый процесс с помощью fork(), а затем заменяет содержимое этого процесса на выполнение команды с помощью exec().

**Основные отличия:**

* **fork()**: Создает полную копию процесса с копированием всех ресурсов (но с оптимизацией через copy-on-write).
* **vfork()**: Быстрее, чем fork(), так как дочерний процесс использует адресное пространство родителя до вызова exec() или завершения.
* **clone()**: Позволяет более тонко настроить разделяемые ресурсы между родителем и дочерним процессом. Используется для создания потоков.
* **exec()**: Не создает процесс, а заменяет текущий процесс новым, обычно используется после fork().
* **system()** автоматически использует **fork()** и **exec()** внутри себя, но не предоставляет такого контроля. Он предназначен для простого выполнения команды в новом процессе с ожиданием завершения.

**Какие потоки данных доступны любому процессу автоматически?**

В каждом процессе в операционных системах обычно доступны несколько стандартных потоков данных, которые автоматически создаются при запуске процесса. В Linux и Windows эти потоки включают:

**1. Стандартный ввод (stdin)** Этот поток используется для получения ввода от пользователя или других источников (например, текстовые файлы).

* **Файловый дескриптор**:
  + В Linux: дескриптор 0.
  + В Windows: стандартный ввод также связан с дескриптором 0.

**2. Стандартный вывод (stdout)** Этот поток используется для вывода данных, например, текстовых сообщений, на экран или в файл.

* **Файловый дескриптор**:
  + В Linux: дескриптор 1.
  + В Windows: стандартный вывод связан с дескриптором 1.

**3. Стандартный вывод ошибок (stderr)** Этот поток используется для вывода сообщений об ошибках, позволяя отличать ошибки от обычного вывода.

* **Файловый дескриптор**:
  + В Linux: дескриптор 2.
  + В Windows: стандартный вывод ошибок связан с дескриптором 2.

**Поясните назначение системного вызов WaitForSingleObject в Windows-приложении.**

* **Ожидание завершения потока или процесса**: При вызове WaitForSingleObject с дескриптором потока или процесса текущий поток приостанавливается, пока ожидаемый объект не завершится. Это важно для синхронизации работы между потоками или процессами.
* **Синхронизация**: Позволяет обеспечить согласованность данных и корректное завершение работы. Например, один поток может выполнить некоторую работу и затем ожидать, когда другой поток завершит выполнение.
* **Ожидание события**: Также может использоваться для ожидания пользовательских объектов, таких как события, мьютексы и семафоры, что позволяет управлять доступом к общим ресурсам.

**Поясните назначение системного вызова wait в Linux- приложении.**

Системный вызов **wait** в Linux используется для управления дочерними процессами. Он позволяет родительскому процессу ожидать завершения одного или нескольких своих дочерних процессов. Вот основные назначения и особенности этого вызова:

* **Ожидание завершения дочернего процесса**: wait блокирует выполнение родительского процесса до тех пор, пока один из его дочерних процессов не завершится. Это помогает синхронизировать работу процессов.
* **Получение информации о завершении**: После завершения дочернего процесса wait возвращает его статус завершения, который позволяет родительскому процессу узнать, успешно ли завершился дочерний процесс или произошла ошибка.
* **Избежание зомби-процессов**: Когда дочерний процесс завершает работу, он не исчезает сразу, а становится зомби-процессом, пока родительский процесс не вызовет wait. Это позволяет избежать накопления зомби-процессов в системе.

**waitpid**: Более гибкий вариант, который позволяет ожидать конкретный дочерний процесс или устанавливать параметры ожидания, например, не блокировать выполнение.

**Заключение:**

Системный вызов wait является важным инструментом для управления процессами в Linux, позволяя эффективно обрабатывать завершение дочерних процессов, обеспечивать синхронизацию и предотвращать проблемы с зомби-процессами.

**Лаб4**

**1**

DWORD — это тип данных, определённый в Windows API. Он представляет собой 32-битное беззнаковое целое число (аналог типа unsigned long в C/C++).

processId и threadId — это переменные типа DWORD, которые хранят 32-битные значения, представляющие уникальные идентификаторы процесса и потока, полученные с помощью функций Windows API.

this\_thread::sleep\_for, которая принимает в качестве аргумента временной интервал из библиотеки chrono.

$A = Get-process name

$A.Threads

**2**

DWORD WINAPI OS04\_02\_T1(LPVOID) — это определение функции для потока в Windows. WINAPI указывает, что используется соглашение о вызовах stdcall, при котором параметры передаются в функцию через стек, и функция сама очищает стек после завершения.

(LPVOID) — это аргумент функции, который используется для передачи данных в поток. В данном случае это универсальный указатель, который может указывать на произвольные данные (например, структуру, массив, целое число и т.д.).

HANDLE CreateThread(

LPSECURITY\_ATTRIBUTES lpThreadAttributes, // атрибуты безопасности

SIZE\_T dwStackSize, // размер стека потока

LPTHREAD\_START\_ROUTINE lpStartAddress, // указатель на функцию, которая будет выполняться в потоке

LPVOID lpParameter, // параметр, который передается в поток

DWORD dwCreationFlags, // флаги создания потока

LPDWORD lpThreadId // указатель на переменную, которая получит идентификатор потока

);

HANDLE hThread1 = CreateThread(...) и HANDLE hThread2 = CreateThread(...): функции Windows API, которые создают два потока. В первом аргументе указывается атрибут безопасности (NULL — по умолчанию), второй аргумент — размер стека (0 — по умолчанию), третий — указатель на функцию, которая будет выполняться в потоке (в данном случае OS04\_02\_T1 и OS04\_02\_T2), и четвертый — параметры для этой функции (здесь NULL). Поток запускается сразу же, так как флаг (пятый аргумент) установлен в 0.

**WaitForSingleObject(hThread1, INFINITE);** и **WaitForSingleObject(hThread2, INFINITE);**: эти функции заставляют основной поток ждать завершения каждого из потоков (hThread1 и hThread2). Флаг INFINITE означает, что программа будет ждать бесконечно долго, пока поток не завершится.

**CloseHandle(hThread1);** и **CloseHandle(hThread2);**: эти функции закрывают дескрипторы потоков, освобождая системные ресурсы.

**3**

После создания потока в приостановленном состоянии, ResumeThread(hThread) используется для запуска его выполнения.

**4**

IsBackground = false

 **Передний поток** продолжает работать, даже если основной поток завершил выполнение.

 **Фоновый поток** завершится, как только основной поток (главный поток) завершит выполнение программы.

ThreadWithParam: Этот метод принимает параметр (строку), который передается в качестве аргумента при запуске потока.

(\*-1) — это вывод из главного потока, который выполняет свои итерации в течение 5 секунд.

(Z-3) — это вывод из потока t1, который выполняет свои итерации в течение 10 секунд.

(Анастасия-4) — это вывод из потока t1a, который работает 20 секунд.

(Немкович-5) — вывод из потока t1b, который фоновый и завершится, когда завершится основной поток, не дождавшись окончания всех итераций.

Thread.CurrentThread.ManagedThreadId — это свойство в C#, которое возвращает уникальный **идентификатор** для текущего потока, в котором выполняется код.

**6**

Thread[] t: Здесь мы объявляем массив t, который может содержать до 20 объектов типа Thread. Каждый элемент этого массива будет представлять отдельный поток.

new Thread[20]: Создается новый массив потоков размером 20. В этом массиве мы будем хранить ссылки на потоки, которые мы создадим позже.

2. Запуск потоков

for (int i = 0; i < 20; ++i)

{

t[i] = new Thread(WorkThread); // Создание нового потока

t[i].Start(); // Запуск потока

}

Подробное объяснение:

for (int i = 0; i < 20; ++i): Это цикл, который повторяется 20 раз (от 0 до 19), и на каждой итерации мы создаем новый поток.

new Thread(WorkThread):

Thread: Это класс из пространства имен System.Threading, который представляет поток выполнения.

WorkThread: Это метод, который будет выполняться в новом потоке. Он должен соответствовать делегату ThreadStart, то есть не принимать никаких параметров и не возвращать значение. В данном случае WorkThread — это статический метод, который будет выполняться в каждом потоке.

t[i] = new Thread(WorkThread);: Создается новый объект типа Thread, который будет выполнять метод WorkThread, и ссылка на этот поток сохраняется в массиве t на позиции i.

t[i].Start();:

Start(): Этот метод запускает поток, то есть вызывает метод WorkThread, который будет выполняться в этом потоке. После вызова Start() поток начинает свое выполнение, и управление возвращается в основной поток (или поток, который его запустил).

3. Ожидание завершения потоков

for (int i = 0; i < 20; ++i)

t[i].Join();

Подробное объяснение:

for (int i = 0; i < 20; ++i): Еще один цикл, который снова выполняется 20 раз (от 0 до 19).

t[i].Join();:

Join(): Этот метод блокирует текущий поток (в данном случае основной поток) до тех пор, пока поток t[i] не завершит свое выполнение. Это означает, что основной поток будет ждать, пока все потоки, которые были запущены, закончат свою работу, прежде чем продолжить выполнение следующего кода (в данном случае, вывода значения Count).

**7**

В цикле поток выполняет следующие действия:

* Определяет текущее время и вычисляет, сколько секунд прошло с момента старта.
* Если прошло меньше ObservationTime, то в соответствующей ячейке матрицы (Matrix[id, ElapsedSeconds]) увеличивается значение на 50.
* Затем поток "спит" 50 миллисекунд, используя метод MySleep.

В Main создается массив потоков t с размером ThreadCount.

В цикле запускаются потоки, каждый из которых выполняет метод WorkThread. Передается идентификатор потока i как параметр o.

После запуска всех потоков основной поток ждет завершения всех с помощью метода Join().

**8**

 Запускается цикл от 0 до ThreadCount (количество потоков, заданное ранее).

 **Передача идентификатора потока**:

csharp

Копировать код

object o = i; // Передаем идентификатор потока

В этой строке создается объект o, который будет использоваться для передачи идентификатора текущего потока в метод WorkThread.

 **Добавление задачи в пул потоков**:

csharp

Копировать код

ThreadPool.QueueUserWorkItem(WorkThread, o); // Добавляем задачу в пул

Метод QueueUserWorkItem добавляет задачу в пул потоков. Он принимает два параметра:

* **WorkThread**: Метод, который будет выполняться в потоке из пула.
* **o**: Аргумент, который будет передан в метод WorkThread.

Пул потоков: Пул потоков — это механизм, который управляет набором потоков, которые могут быть переиспользованы для выполнения задач. Это позволяет избежать накладных расходов на создание и уничтожение потоков для каждой задачи, что делает его более эффективным для работы с большим количеством короткоживущих задач.

QueueUserWorkItem: Этот метод добавляет задачу в пул потоков. Пул выделяет потоки для выполнения добавленных задач, когда они становятся доступны. Этот подход позволяет более эффективно использовать ресурсы системы.

**9**

Создание массива задач:

csharp

Копировать код

Task[] tasks = new Task[TaskCount]; // Массив задач

Здесь создается массив задач, где TaskCount — это константа, определяющая количество задач, которые будут созданы и запущены.

Вывод сообщения о создании задач:

csharp

Копировать код

Console.WriteLine("Создание задач...");

Этот вывод информирует пользователя о начале процесса создания задач.

Запуск задач:

csharp

Копировать код

tasks[0] = Task.Run(() => Work(0));

tasks[1] = Task.Run(() => Work(1));

// ...

tasks[19] = Task.Run(() => Work(19));

В этом блоке кода задачи запускаются с помощью Task.Run(). Каждый вызов Task.Run() создает новую задачу, которая будет выполнена асинхронно. Давайте подробнее разберем, что здесь происходит:

Task.Run(): Этот метод используется для запуска задачи асинхронно. Он принимает делегат (в данном случае — лямбда-выражение), который будет выполнен в отдельном потоке.

Лямбда-выражение: Внутри Task.Run() используется лямбда-выражение () => Work(taskId), которое представляет метод Work, который будет выполнен. Здесь taskId — это параметр, передаваемый в метод Work, который идентифицирует, какая задача выполняется.

Захват переменной: Так как taskId объявляется в цикле, необходимо передать его значение. В данном случае, каждая задача получает уникальный идентификатор (от 0 до 19), который соответствует индексу в массиве tasks.

Метод Task.WaitAll(tasks) приостанавливает выполнение текущего потока (в данном случае, основного потока), пока все задачи в массиве tasks не завершатся. Это позволяет убедиться, что программа не завершится, пока все задачи не будут выполнены.

Задачи (или Tasks) в .NET представляют собой механизм для управления асинхронным и параллельным выполнением кода. Они позволяют создавать и выполнять операции, которые могут выполняться в фоновом режиме, не блокируя основной поток выполнения. Давайте рассмотрим, как работают задачи в .NET, их основные характеристики и преимущества.

1. Основы задач (Task)

Задачи в .NET реализованы через класс Task, который находится в пространстве имен System.Threading.Tasks. Они представляют собой асинхронные операции, которые могут выполняться в любом потоке пула потоков.

2. Создание задач

Задачи создаются с помощью статического метода Task.Run или конструктора Task. Например:

Задачи могут быть запущены сразу после их создания или позже. Для запуска задачи можно использовать метод Start(), если вы создали задачу с помощью конструктора, или использовать метод Task.Run, который сразу же запускает задачу.

4. Ожидание завершения задач

После запуска задачи вы можете дождаться её завершения с помощью метода Wait():

Задачи позволяют эффективно использовать ресурсы системы, распределяя нагрузку между несколькими потоками. Это особенно полезно для вычислительно сложных операций или операций ввода-вывода.

Задачи управляются пулом потоков, что означает, что система сама управляет созданием и уничтожением потоков. Это значительно упрощает управление потоками и улучшает производительность приложения, поскольку не требуется создавать новый поток для каждой задачи.

10

Первая задача запускается с помощью метода Task.Run(), который принимает лямбда-выражение, вызывающее метод Work(0). Этот метод выполняется в фоновом режиме, и его выполнение не блокирует основной поток.

ContinueWith позволяет вам указать действие, которое будет выполнено после завершения текущей задачи. В данном случае, задача tasks[1] запускает Work(1) только после завершения tasks[0].

Метод Task.WaitAll() блокирует выполнение основного потока до тех пор, пока все задачи в массиве tasks не завершатся.

**Что такое поток управления OS?**

Поток выполнения (англ. thread — нить) — наименьшая единица обработки, исполнение которой может быть назначено ядром операционной системы. Несколько потоков выполнения могут существовать в рамках одного и того же процесса и совместно использовать ресурсы, такие как память, тогда как процессы не разделяют этих ресурсов.

В частности, потоки выполнения разделяют:

* Инструкции процесса (его код);
* Контекст процесса (значения переменных, которые они имеют в любой момент времени).

**С помощью каких системных вызовов создаются потоки в Windows и Linux?**

Новый поток создается с помощью вызова функции **pthread\_create**. В качестве значения функции возвращается идентификатор только что созданного потока. Поток создается путем выполнения start\_routine с arg в качестве единственного аргумента. Если start\_routine возвращает значение, эффект такой же, как если бы был неявный вызов pthread\_exit() с использованием возвращаемого значения start\_routine в качестве статуса выхода.

Функция **CreateThread** создает новый поток для процесса. Создаваемый поток должен указать начальный адрес кода, который должен выполнить новый поток. Как правило, начальный адрес — это имя функции, определенной в коде программы. Эта функция принимает один параметр и возвращает значение DWORD.

**Что такое системные и пользовательские потоки?**

При первом подходе (на уровне пользовательского пространства) процесс ведет собственную таблицу потоков, а ядро считает этот процесс однопоточным. Переключение потоков в этом случае требует меньше затрат процессорного времени. Однако возникают проблемы с планированием — если поток не возвращает управление добровольно, другие потоки не смогут выполняться. Также системные вызовы или отсутствие страницы в памяти могут заблокировать весь процесс.

При реализации потоков на уровне ядра не требуется новых неблокирующих системных вызовов. Более того, если один из потоков сталкивается с отсутствующей страницей, ядро может проверить, есть ли у процесса другие готовые к выполнению потоки, и запустить один из них, пока ожидается загрузка страницы с диска. Главным недостатком этого подхода являются высокие затраты на системные вызовы, что особенно ощутимо при частых операциях с потоками, таких как создание или удаление. Также могут возникнуть сложности, например, неопределенность при разветвлении многопоточного процесса или при обработке сигналов. Управляются операционной системой через системные вызовы (например, pthread\_create в Linux или CreateThread) Системный вызов — это механизм, с помощью которого программы в пользовательском пространстве взаимодействуют с ядром операционной системы для выполнения различных привилегированных операций

**Что такое многопоточность?**

Многопоточность позволяет процессу, созданному в операционной системе, состоять из нескольких потоков, которые выполняются «параллельно», то есть без строго заданного порядка. В .NET Framework начиная с версии 4 рекомендуется использовать библиотеку параллельных задач (TPL) и Parallel LINQ (PLINQ) для организации многопоточности.

TPL и PLINQ опираются на потоки пула потоков (ThreadPool). Класс System.Threading.ThreadPool предоставляет .NET-приложениям пул рабочих потоков, которые можно использовать для выполнения задач. Также можно напрямую работать с классом System.Threading.Thread, который представляет управляемый поток.

Когда несколько потоков требуют доступ к общему ресурсу, необходимо синхронизировать доступ, чтобы избежать состояния гонки и сохранить ресурс в целостном состоянии. Для этого .NET предоставляет различные типы для синхронизации и координации потоков. Важным аспектом работы с потоками является обработка исключений, так как необработанные исключения в потоках могут привести к завершению всего процесса.

**Что такое контекст потока и для чего он нужен?**

Контекст потока содержит все сведения, позволяющие потоку безболезненно возобновить выполнение, в том числе набор регистров процессора и стек потока. Несколько потоков могут выполняться в контексте процесса. Все потоки процесса используют общий диапазон виртуальных адресов. Поток может исполнять любую часть программного кода, включая части, выполняемые в данный момент другим потоком

* Основные компоненты контекста потока:
* Регистры процессора
* Стек
* Состояние потока
* Идентификатор потока
* Приоритет потока
* Используемые ресурсы

**Перечислите состояния, в которых может быть поток и поясните их назначение.**

Каждый созданный в системе поток может находиться в одном из трех состояний:

* Состояние выполнения: в этом состоянии код потока выполняется процессором. На однопроцессорных платформах только один поток может находиться в этом состоянии в любой момент времени.
* Состояние готовности к выполнению: поток готов продолжить свою работу и ожидает освобождения центрального процессора (ЦП).
* Состояние ожидания: поток не требует времени ЦП и ждет наступления определенного события, например, завершения операции ввода/вывода или освобождения необходимого ресурса. Потоки в этом состоянии часто называют блокированными.

Изменение состояния потока происходит в результате различных действий, которые можно описать следующим образом:

* «Готовность» → «Выполнение»: система выбирает поток для выполнения в соответствии с алгоритмом планирования и предоставляет ему ЦП.
* «Выполнение» → «Готовность»: поток готов продолжать работу, но система решает прервать его выполнение. Это может произойти по двум основным причинам:
* Завершается выделенное потоку время владения процессором.
* Появляется более приоритетный поток среди готовых к выполнению.
* «Выполнение» → «Ожидание»: если дальнейшее выполнение кода текущего потока невозможно без наступления какого-либо события, поток прерывает выполнение и переводится в состояние ожидания (блокируется).
* «Ожидание» → «Готовность»: когда происходит событие, которое ожидал один из блокированных потоков, система переводит его в состояние готовности (разблокирует). После этого поток будет учитываться системой при планировании использования ЦП.
* Поток может завершить свое выполнение нормально или аварийно. После этого система удаляет его дескриптор из своей внутренней структуры, и поток перестает существовать.

В состояниях готовности и ожидания может находиться несколько потоков, поэтому система создает отдельные списковые структуры для хранения их дескрипторов. Организация этих списков зависит от принципов, заложенных в систему планирования потоков для данной операционной системы.

**Что такое LWP?**

Многие системы, реализующие модель «многие ко многим» или двухуровневую модель, размещают промежуточную структуру данных между пользовательскими потоками и потоками ядра. Эта структура данных, обычно называемая облегченным процессом или LWP. LWP выглядит как виртуальный процессор, на котором приложение может запланировать запуск пользовательского потока в библиотеке пользовательских потоков. Каждый облегченный процесс прикреплен к потоку ядра, и именно потоки ядра операционная система планирует запустить на физических процессорах. LWP также блокируется, если блокируется поток ядра (например, при ожидании завершения операции ввода-вывода). Поток пользовательского уровня, прикрепленный к LWP выше по цепочке, также блокируется.

Для эффективного запуска любого количества LWP может потребоваться приложение. Рассмотрим приложение, привязанное к ЦП и работающее на одном процессоре. В этом сценарии одновременно может работать только один поток, поэтому достаточно одного LWP. Однако приложению с интенсивным вводом-выводом может потребоваться выполнение нескольких LWP. Обычно LWP требуется для каждого параллельного системного вызова блокировки. Например, четыре разных запроса на чтение файлов происходят одновременно. Требуется четыре LWP, поскольку все они могут ожидать завершения ввода-вывода в ядре. Если у процесса всего три LWP, то четвертый запрос должен ждать, пока один из LWP вернется из ядра.

В некоторых операционных системах нет отдельного слоя LWP между потоками ядра и пользовательскими потоками. Это означает, что пользовательские потоки реализуются непосредственно поверх потоков ядра. В этих контекстах термин «легкий процесс» обычно относится к потокам ядра, а термин «потоки» может относиться к пользовательским потокам. В Linux пользовательские потоки реализуются путем разрешения определенным процессам совместно использовать ресурсы, что иногда приводит к тому, что эти процессы называют «легкими процессами».

Каждый LWP является ресурсом ядра в пуле ядра и выделяется (присоединяется) и отсоединяется (отсоединяется) для потока на основе каждого потока. Это происходит по мере планирования или создания и

**Что такое потокобезопасность программного кода?**

Потокобезопасность — это предотвращение гонок данных — ситуаций, в которых данные устанавливаются в правильные или неправильные значения в зависимости от порядка, в котором несколько потоков получают доступ к данным и изменяют их.

Процедура потокобезопасна, когда она логически корректна при одновременном выполнении несколькими потоками.

Не потокобезопасно:

* Существуют области памяти (например, глобальные или статические переменные), доступные более чем одному потоку. Каждый поток имеет свои локальные переменные, которые находятся в его собственном стеке и недоступны другим потокам.
* Есть определённое свойство (инвариант), которое должно быть истинным для правильной работы программы. Например, переменная totalRequests должна отражать общее количество выполненных операций увеличения потоками.
* Во время обновления значения инвариант может временно нарушаться. Например, при чтении totalRequests и перед записью нового значения инвариант может быть нарушен.
* Другие потоки могут получить доступ к общей памяти в момент, когда инвариант нарушен. Это может привести к непредсказуемому или некорректному поведению программы.

**Что такое реентерабельность кода?**

Компьютерная программа в целом или её отдельная процедура называется реентерабельной, если она разработана таким образом, что одна и та же копия инструкций программы в памяти может быть совместно использована несколькими пользователями или процессами [3]. При этом второй пользователь может вызвать реентерабельный код до того, как с ним завершит работу первый пользователь и это как минимум не должно привести к ошибке, а при корректной реализации не должно вызвать потери вычислений (то есть не должно появиться необходимости выполнять уже выполненные фрагменты кода). Реентерабельность тесно связана с безопасностью функции в многопоточной среде (thread-safety), тем не менее, это разные понятия. Обеспечение реентерабельности является ключевым моментом при программировании многозадачных систем, в частности, операционных систем.

Для обеспечения реентерабельности необходимо выполнение нескольких условий:

* никакая часть вызываемого кода не должна модифицироваться;
* вызываемая процедура не должна сохранять информацию между вызовами;
* если процедура изменяет какие-либо данные, то они должны быть уникальными для каждого пользователя;
* процедура не должна возвращать указатели на объекты, общие для разных пользователей.

**Что такое Fiber?**

Волокно — наименьшая последовательность запрограммированных инструкций, которой можно управлять в пользовательском режиме, внешнем по отношению к планировщику операционной системы. Волокно представляет память для стека выполнения и набор регистров ЦП, необходимых для выполнения инструкций. Волокна могут выполняться только явно потоками и никогда не будут запланированы планировщиком операционной системы. Запуск волокна является кооперативным, то есть поток должен явно выбрать волокно для запуска, а волокно должно сообщить потоку, в котором оно запущено, что оно завершено или его необходимо приостановить.

Волокно похоже на поток в том смысле, что у него есть свой стек, но в отличие от потока в том, что операционная система никогда не планирует волокно. Вместо этого рабочие потоки carb.tasking запускают волокна по мере их доступности.

Fiber (волокно) — это легковесный поток исполнения, который позволяет разработчикам реализовывать многопоточность на более высоком уровне, обеспечивая возможность управлять выполнением и переключением между волокнами более эффективно, чем это можно сделать с традиционными потоками операционной системы.

Основные характеристики Fiber:

* Волокна намного легче, чем традиционные потоки, так как они имеют меньшие накладные расходы на создание и управление.
* В отличие от потоков ядра, которые управляются операционной системой, волокна управляются на уровне приложения. Это позволяет разработчикам контролировать, когда и как происходит переключение между волокнами, что может привести к более эффективному использованию ресурсов.
* Поскольку волокна управляются на уровне пользователя, переключение между ними происходит быстрее, так как не требуется вмешательство ядра.
* Каждое волокно имеет своё собственное состояние выполнения, включая регистры процессора, стек и локальные переменные, что позволяет каждому волокну выполняться независимо от других.

**Лаб 6**

В операционных системах Windows объектами синхронизации называются объекты ядра, которые могут находиться в одном из двух состояний: сигнальном (signaled) и несигнальном (nonsignaled).

Группа команд BT, BTC, BTS и BTR используется для работы с отдельными битами своего операнда. Команда BT (Bit) позволяет получить состояние заданного бита своего операнда. Команды BTC (Bit Change), BTS (Bit Set), BTR (Bit Reset) также получают состояние заданного бита, но помимо этого еще выполняют некоторые действия над этим битом.

Объект **критическая секция** обеспечивает синхронизацию. Этим объектом может владеть только один поток, что и обеспечивает синхронизацию. Для работы с критическими секциями есть ряд функций API и тип данных CRITICAL\_SECTION. Для использования критической секции нужно создать переменную данного типа, и проинициализировать ее перед использованием с помощью функции InitializeCriticalSection(). Для того, чтобы войти в секцию нужно вызвать функцию EnterCriticalSection(), а после завершения работы LeaveCriticalSection(). Что будет, если поток обратится к секции, в которой сейчас другой поток ? Тот поток, который обратится будет блокирован пока критическая секция не будет освобождена. Саму критическую секцию можно удалить функцией DeleteCriticalSection(). Для того, чтобы обойти блокировку потока при обращении к занятой секции есть функция TryEnterCriticalSection(), которая позволяет проверить критическую секцию на занятость.

Флаг CREATE\_NEW\_CONSOLE, используемый в функции CreateProcessW, указывает системе Windows создать для нового процесса отдельное консольное окно.

**mutex** = CreateMutex(NULL, FALSE, L"OS06\_03");

1. NULL (lpMutexAttributes): Указывает на атрибуты безопасности объекта. Если задан NULL, то используются настройки по умолчанию, и объект мьютекса может быть доступен только текущему пользователю.
2. FALSE (bInitialOwner): Задает начальное состояние мьютекса.
   * Если значение FALSE, то мьютекс создается в неназначенном (освобожденном) состоянии.
   * Если TRUE, то текущий поток сразу захватывает мьютекс, и другие процессы или потоки должны ждать, пока он не будет освобожден.
3. L"OS06\_03" (lpName): Имя мьютекса.
   * Это имя позволяет другим процессам или потокам получить доступ к тому же самому объекту мьютекса, если они укажут то же самое имя.
   * Если мьютекс с таким именем уже существует в системе, CreateMutex вернет дескриптор существующего мьютекса, а не создаст новый.

WaitForMultipleObjects(

DWORD nCount, // Количество объектов

const HANDLE \*lpHandles, // Массив дескрипторов объектов

BOOL bWaitAll, // Указывает, нужно ли ждать завершения всех объектов

DWORD dwMilliseconds // Время ожидания

);

Функция OpenMutex используется для получения дескриптора уже существующего объекта мьютекса.

1. dwDesiredAccess (SYNCHRONIZE):
   * Этот параметр определяет, какой тип доступа необходим для работы с объектом мьютекса.
   * В данном случае используется флаг SYNCHRONIZE, который позволяет только ожидать мьютекс (с помощью функций типа WaitForSingleObject или WaitForMultipleObjects), но не изменять его состояние (не захватывать или освобождать). Это минимальный уровень доступа, который необходим, чтобы синхронизировать выполнение потоков и процессов.
   * Если нужно получить полный доступ (например, захватывать или освобождать мьютекс), можно использовать флаг MUTEX\_ALL\_ACCESS, но в данном контексте SYNCHRONIZE вполне достаточен.
2. bInheritHandle (FALSE):
   * Этот параметр указывает, будет ли дескриптор мьютекса унаследован дочерними процессами.
   * Если FALSE, дескриптор не будет унаследован.
   * Если TRUE, дескриптор может быть унаследован дочерними процессами (что полезно, если нужно передать управление мьютексом дочернему процессу).
3. lpName (L"OS06\_03"):
   * Это имя мьютекса, который вы хотите открыть. Мьютекс с таким именем должен быть создан ранее (например, с помощью CreateMutex), и другие процессы могут использовать это имя для обращения к тому же мьютексу.
   * В данном случае, если мьютекс с именем L"OS06\_03" существует, функция OpenMutex вернет дескриптор этого мьютекса.

Мьютекс захватывается потоком посредством любой функции ожидания, а освобождается функцией ReleaseMutex

**Семафор** — это синхронизирующий объект, который управляет доступом к ресурсу на основе счетчика разрешений. Этот счетчик указывает, сколько потоков или процессов могут одновременно использовать данный ресурс. Семафоры делятся на:

1. Двоичные семафоры (аналогичны мьютексу), где счетчик может быть только 0 или 1.
2. Счетные семафоры (собственно, семафоры), где счетчик может быть любым неотрицательным числом.

semaphore = CreateSemaphore(NULL, 2, 2, L"OS06\_04");

1. Первый параметр (NULL):
   * Этот параметр указывает на атрибуты безопасности. Если значение NULL, семафор создается с атрибутами безопасности по умолчанию, что обычно подходит для стандартного использования. Атрибуты безопасности могут быть указаны, чтобы настроить доступ к семафору другими процессами или потоками.
2. Второй параметр (2):
   * Это начальное значение счетчика семафора, то есть сколько потоков или процессов могут одновременно захватить (использовать) семафор сразу после его создания. В данном случае, начальное значение — 2, что означает, что сразу после создания семафора два потока могут одновременно получить доступ к ресурсу, на который указывает этот семафор.
3. Третий параметр (2):
   * Это максимальное значение счетчика семафора. Максимальное значение указывает, какое наибольшее количество потоков или процессов может одновременно владеть семафором. Если счетчик семафора достиг максимального значения, его больше нельзя увеличивать вызовами ReleaseSemaphore. В данном случае максимальное значение также равно 2, что ограничивает одновременный доступ к ресурсу двумя потоками или процессами.
4. Четвертый параметр (L"OS06\_04"):
   * Это имя семафора. Если указать имя, семафор можно использовать для синхронизации не только между потоками, но и между процессами. Если семафор с таким именем (L"OS06\_04") уже существует в системе, CreateSemaphore вернет дескриптор к уже существующему семафору, а не создаст новый. Если имя указано как NULL, семафор будет безымянным и доступен только внутри текущего процесса.

BOOL ReleaseSemaphore(

HANDLE hSemaphore, // Дескриптор семафора

LONG lReleaseCount, // Число увеличений счетчика семафора

LPLONG lpPreviousCount // Указатель для записи предыдущего значения счетчика (может быть NULL)

);

HANDLE OpenSemaphore(

DWORD dwDesiredAccess, // Уровень доступа к семафору

BOOL bInheritHandle, // Возможность наследования дескриптора семафора дочерними процессами

LPCWSTR lpName // Имя семафора, которое было присвоено при создании

);

* SEMAPHORE\_MODIFY\_STATE (0x0002): Позволяет изменять состояние семафора (например, вызывать ReleaseSemaphore для увеличения счётчика).
* SEMAPHORE\_ALL\_ACCESS (0x1F0003): Предоставляет полный доступ к семафору, включая возможность изменять его состояние и ожидать на нем. Это означает, что процесс может как вызывать WaitForSingleObject (ждать на семафоре), так и ReleaseSemaphore (освобождать семафор).

HANDLE **event** = CreateEvent(

NULL, // Атрибуты безопасности

FALSE, // Автоматический сброс (если TRUE — ручной сброс)

FALSE, // Начальное состояние (FALSE — не сигнал)

NULL // Имя события

);

1. Автоматическое событие (Auto-reset event)

Когда событие сигнализируется (становится в состоянии "сигнал"), оно автоматически сбрасывается в состояние "не сигнал" после того, как хотя бы один поток его обработает. То есть, после того как поток, ожидающий на этом событии, пробуждается, событие возвращается в состояние "не сигнал", и другие потоки не могут его обработать.

2. Ручное событие (Manual-reset event)

Когда событие сигнализируется, оно остается в состоянии "сигнал" до тех пор, пока не будет явно сброшено. Все потоки, которые будут ожидать на этом событии, будут пробуждаться до тех пор, пока событие не будет сброшено с помощью функции ResetEvent.

Сброс события:

* Для автоматических событий состояние события будет сброшено после того, как хотя бы один поток, ожидающий на этом событии, будет пробужден.
* Для ручных событий состояние события остаётся в "сигнал" до тех пор, пока оно не будет сброшено вручную с помощью ResetEvent.

Для освобождения нескольких потоков, ждущих сигнального состояния события с ручным сбросом, используется функция PulseEvent, которая имеет следующий прототип:

|  |  |  |
| --- | --- | --- |
| BOOL PulseEvent( |  |  |
| HANDLE | hEvent | // дескриптор события |
| ); |  |  |

При вызове этой функции все потоки, ждущие события с дескриптором hEvent, выводятся из состояния ожидания, а само событие сразу переходит в несигнальное состояние. Если функция PulseEvent вызывается для события с автоматическим сбросом, то из состояния ожидания выводится только один из ожидающих потоков. Если нет потоков, ожидающих сигнального состояния события из функции PulseEvent, то состояние этого события остается несигнальным.

EVENT\_ALL\_ACCESS

EVENT\_MODIFY\_STATE

SYNCHRONIZE

Флаг EVENT\_ALL\_ACCESS означает, что поток может выполнять над событием любые действия. Флаг EVENT\_MODIFY\_STATE означает, что поток может использовать функции SetEvent и ResetEvent для изменения состояния события. Флаг SYNCHRONIZE означает, что поток может использовать событие в функциях ожидания.