Министерство образования Республики Беларусь
Учреждение образования «Витебский государственный университет имени П.М. Машерова»
Кафедра прикладного и системного программирования

ОПЕРАЦИОННЫЕ СИСТЕМЫ. ПРОЦЕССЫ И ПОТОКИ

Методические рекомендации

Витебск
ВГУ имени П.М. Машерова
2022

УДК 004.451(075.8)
ББК 32.972.112я73
О-60

Печатается по решению научно-методического совета учреждения образования «Витебский государственный университет имени П.М. Машерова». Протокол № 2 от 05.01.2022.

Составитель: старший преподаватель кафедры прикладного и системного программирования ВГУ имени П.М. Машерова **В.В. Новый**

Р е ц е н з е н т :
заведующий кафедрой информационных систем и технологий УО «ВГТУ», кандидат технических наук, доцент *В.Е. Казаков*

О-60 **Операционные системы. Процессы и потоки :** методические рекомендации / сост. В.В. Новый. – Витебск : ВГУ имени П.М. Машерова, 2022. – 47 с.

В методических рекомендациях изложены основные вопросы, связанные с исполнением программ в операционных системах семейств Windows и GNU/Linux. Рассмотрены эволюция и классификация ОС, понятия процесса, потока, синхронизации.

Издание предназначается для студентов специальностей «Управление информационными ресурсами», «Прикладная информатика (по направлениям)», «Информационные системы и технологии (в здравоохранении)», «Компьютерная безопасность» (дисциплина «Операционные системы»), «Программное обеспечение информационных технологий» (дисциплина «Системное программирование»), «Прикладная математика (научно-педагогическая деятельность)» (дисциплина «Операционные системы Windows»).

УДК 004.451(075.8)
ББК 32.972.112я73

СОДЕРЖАНИЕ

ВВЕДЕНИЕ

Операционные системы являются неотъемлемой частью программного обеспечения современных вычислительных систем и компьютерной техники. Одной из важнейших задач операционной системы является выполнение приложений пользователя. Это требует от операционной системы организации большого количества информации по управлению различными аппаратными и программными ресурсами: памятью, процессорным временем, устройствами ввода-вывода, объектами ядра и т.д. Это в свою очередь приводит к понятию процесса. Другой важной концепцией, перекликающейся с процессами, является концепция потоков исполнения и многопоточного программирования. Владение этими понятиями и умение применять их на практике является необходимой характеристикой современного программиста.

В данных методических рекомендациях приведены краткие теоретические сведения по различным вопросам, связанным с использованием процессов и потоков в прикладном программном обеспечении, примеры их использования и предложены задания для лабораторных работ.

Все примеры и задания ориентированы на использование языков C/C++ и распространенных операционных систем персональных компьютеров, таких как Microsoft Windows и GNU/Linux. Данный подход позволяет применять отдельные модули этих методических рекомендаций вне зависимости от привязки к конкретной операционной системе. Рассматриваются вопросы, связанные с запуском и завершением процессов, созданием потоков исполнения и управлением ими, а также ряд вопросов, связанных с синхронизацией процессов и потоков.

Материал соответствует отдельным темам учебных программ курсов: «Операционные системы» (специальностей «Управление информационными ресурсами», «Прикладная информатика (по направлениям)», «Информационные системы и технологии (в здравоохранении)», «Компьютерная безопасность»), «Операционные системы Windows» (специальность «Прикладная математика (научно-педагогическая деятельность)»), «Системное программирование» (специальностей «Программное обеспечение информационных технологий», «Программное обеспечение информационных систем»).

§ 1. ВВЕДЕНИЕ В ОПЕРАЦИОННЫЕ СИСТЕМЫ

Операционную систему чаще всего определяют, как программу, которая управляет выполнением прикладных программ и выступает в роли интерфейса между приложениями и устройствами компьютера (рисунок 1).

Рисунок 1 – Место операционной системы в структуре ПО компьютера

Функции операционной системы. Существуют различные подходы к определению функции, выполняемых операционной системой. В большинстве случаев, операционная система выполняет следующие функции:

- Организация пользовательского интерфейса (прием от пользователя заданий или команд, сформулированных на соответствующем языке и их обработка);
- Загрузка в ОЗУ и запуск программ, подлежащих исполнению;
- Распределение памяти и организация виртуальной памяти;
- Прием и исполнение запросов от приложений, включая обслуживание всех операции ввода-вывода (I/O-операции);
- Обеспечение работы систем управления файлами;
- Обеспечение режима **мультипрограммирования** (параллельного выполнения двух и более программ, создающего видимость их одновременного выполнения);
- Планирование и диспетчеризация задач;
- Организация механизмов обмена сообщениями и данными между выполняющимися программами;
- Обнаружение и обработка ошибок аппаратного и программного обеспечения;
- Защита программ от взаимного влияния, обеспечивающая сохранность данных и защита самой ОС от исполняющихся приложений;
- Аутентификация и авторизация пользователей, поддержка функции безопасности;

- Упрощение разработки программного обеспечения.

Согласно Эндрю Таненбауму [Современные операционные системы] операционные системы выполняют две основных роли:

- Предоставление **расширенной машины**, более простой для программирования, чем реальный компьютер;
- Предоставление механизмов для **управления ресурсами** вычислительной системы – процессорами, памятью, устройствами I/O и информацией (файлы, структуры ОС).

ОС как расширенная машина. Для получения сервисов ОС используют специальные инструкции, называемые **системными вызовами**.

Совокупность системных вызовов и правил, по которым их следует использовать определяют **интерфейс прикладного программирования** (**API**).

В целом, типичная компьютерная система обладает тремя ключевыми интерфейсами:

- Архитектурой набора команд (ISA, Instruction Set Architecture), который определяет набор инструкции программируемой части микропроцессора и является границей между аппаратурой и программным обеспечением (см. лекцию, посвященную ядру операционной системы);
- Двоичным интерфейсом приложений (ABI, Application Binary Interface), который определяет каким форматам файлов и интерфейсам с операционной системой должны подчиняться прикладные программы для некоторых микропроцессоров и какие аппаратные ресурсы и службы должны предоставляться. Отвечает за переносимость программного обеспечения в двоичном виде между платформами, на которых ABI реализован;
- Интерфейсом прикладного программирования (API, Application Programming Interface), предоставляющий программам доступ к аппаратным ресурсам и возможностям операционной системы через набор вызовов на языке высокого уровня (ЯВУ).

Для обхода несовместимости API различных ОС создаются т.н. **программные (системные) среды** – некоторое системное программное окружение, позволяющее выполнять все системные запросы от прикладной программы.

Системная программная среда, которая непосредственно образуется кодом ОС, называется **основной**, **естественной**, или **нативной** (native).

Кроме нативной путем эмуляции могут создаваться дополнительные программные среды (например, подсистема OS/2 для приложений IBM OS/2 и ntvdm для исполнения dos-приложений в ранних версиях Microsoft Windows NT).

Основные ресурсы ОС. С точки зрения управления ресурсами операционная система предоставляет возможности для распределения аппаратных и программных ресурсов между конкурирующими за них исполняющимися программами. К таким ресурсам можно отнести:

- центральные процессоры;
- оперативную память;

- дисковые накопители;
- различные устройства I/O;
- файлы;
- системные объекты и структуры.

При этом используются два основных подхода к распределению подобных ресурсов между выполняющимися программами (рисунок 1).

В первом случае, как правило, ресурс является неделимым с точки зрения распределения и предоставляется каждой программе в монопольное использование на ограниченное время. Во втором случае ресурс поддерживает возможность разделения на отдельные части, каждая из которых может быть предоставлена отдельной программе.

Эволюция ОС. Развитие операционных систем тесно связано с развитием вычислительной техники и увеличением её сложности и производительности.

Первое поколение (1945–1955)

Элементная база: реле, в последствии вакуумные лампы и коммутационные панели

Особенности: одна и та же группа людей разрабатывала, строила, программировала и использовала ЭВМ.

Языки программирования пока неизвестны – программирование велось через установку перемычек на коммутационной панели. В начале 50-х добавились перфокарты.

ОС: отсутствуют

Второе поколение (1955–1965)

Элементная база: транзисторы

Особенности: разделение между разработчиками, конструкторами, программистами (алгоритмистами и кодерами), операторами и обслуживающим персоналом.

Вычислительные системы представлены мэйнфреймами. Производители поддерживают две различных линии продуктов: для организации ввода-вывода данных и для выполнения вычислений.

ЯП – Фортран и ассемблер.

ОС: пакетные системы. Главная идея: использование программы – **монитора** (monitor) для автоматизации рутинных задач по загрузке библиотек языка программирования и загрузки и запуска программ с магнитной ленты. По завершении задачи управление переходит к монитору, который автоматически загружает следующую задачу. В каждое задание включаются простые команды языка управления заданиями (job control language – JCL). Типичные ОС:

- **FMS (**Fortran Monitor System**)**
- **IBSYS (**для IBM 7094**)**

Третье поколение (1965–1980).

Элементная база: полупроводниковые интегральные схемы

Особенности: создание совместимых семейств ЭВМ (IBM System/360). Миникомпьютеры. Оборудование защиты памяти.

К этому этапу относится появление основных технологии ОС: многозадачности, защиты памяти, подкачки (spooling), режима разделения времени, файловых систем, диалогового режима работы.

Типичные ОС:

- **M.I.T. CTSS** (Compatible Time Sharing System) (1963 г. ~5000 36-битовых слов);
- **IBM OS/360** для System/360 (1964 г. ~1 млн. машинных команд);
- **MULTICS** (MULTIplexed Information and Computer Service) (M.I.T., Bell Labs и General Electric) (1975 г. ~20 млн. команд).

Четвертое поколение (1980-настоящее).

Элементная база: СБИС (сверхбольшие интегральные схемы)

Особенности: микропроцессоры и персональные компьютеры.

Идеи ОС: концепция дружественного к пользователю интерфейса, GUI, сетевые и распределенные ОС.

Типичные операционные системы начала этого периода:

- Intel i8080 (1974) и **CP/M** (Control Program for Microcomputers) (Гэри Килдэлл и Digital Research) ~4Кб
- Motorola 6800 и 6502 – **Apple II** (Apple DOS 3.1 – июнь 1978 г.)
- Intel i8086 – **DOS** (выкупленная у Seattle Computer Products) (Microsoft).

CP/M, MS-DOS и Apple DOS – это системы командной строки (часто обозначается как **CLI - Command Line Interface**, интерфейс командной строки).

В это же время Дуглас Энгелбарт из Стенфордского исследовательского института предложил концепцию GUI (**Graphical User Interface**, графический интерфейс пользователя или ГИП) с окнами, пиктограммами, меню и мышью. Идея была реализована на компьютере Xerox Alto в 1973 году, но не получила коммерческого распространения.

В 1983 годы было представлено коммерческое решение – компьютер Apple Lisa с графическим интерфейсом пользователя. В 1984 Стив Джобс (Apple) использовал идею **дружественного к пользователю** ПК в другом успешном проекте компании Apple - Apple Macintosh (CPU Motorola 16 бит 68000, 64Kb ROM) + Mac System Software (System 1.0) (рисунок 2).

В 2001 году произошла смена ОС Apple на **Mac OS X** (новая версия Macintosh GUI поверх проекта Darwin OS основанного на Berkeley UNIX). В настоящее время носит имя **Mac OS**.

В ответ на разработку компании Apple, компания Microsoft предложила свою графическую оболочку поверх 16-битной операционной системы MS DOS – Windows (1985). Со временем эта оболочка эволюционировала до операционной системы – Microsoft Windows 95. Эта система и её потомки объединяют в семейство Windows9x, просуществовавшее до начала 21 века

и включавшее в себя помимо версии Windows 95 операционные системы Windows 98 и Windows ME (Millenium Edition). Параллельно с этими пользовательскими системами Microsoft разрабатывала полностью 32-битную систему без унаследованного 16-битного кода для корпоративных пользователей. Проект получил наименование Windows NT (Net Technology). К развитию этого проекта относятся и современные пользовательские версии Windows 10, Windows 11 и серверные Windows 2019 Server, Windows 2022 Server.

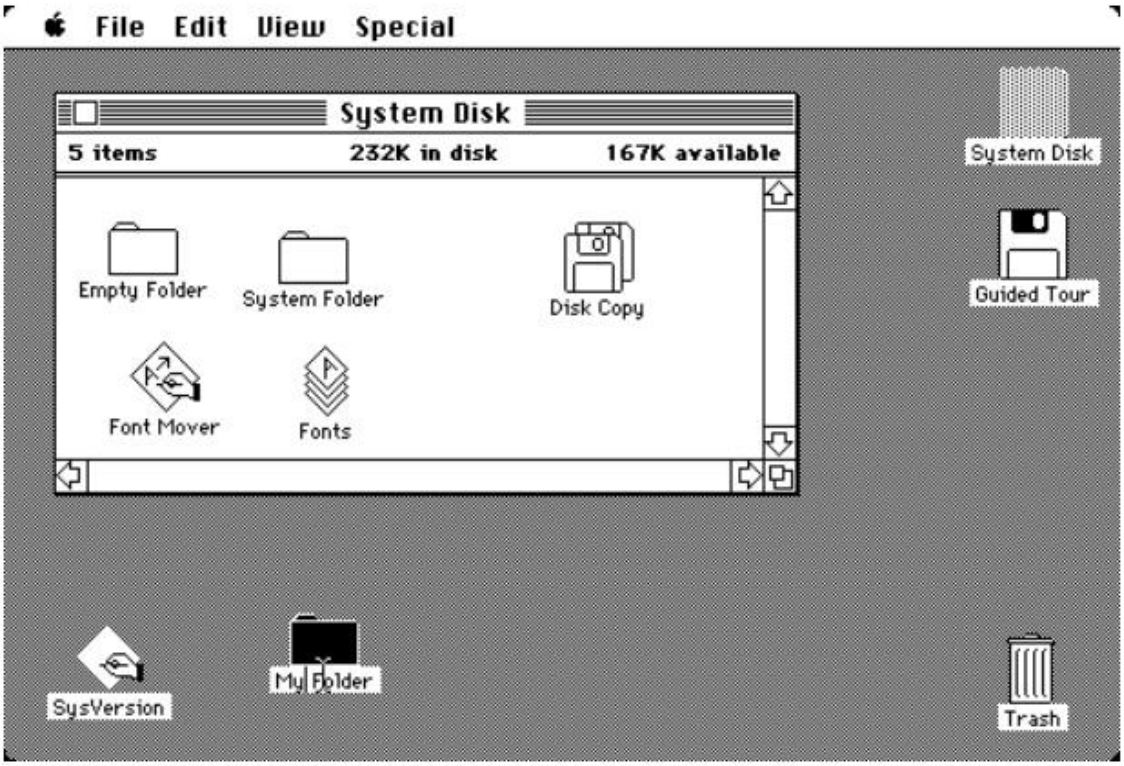

Рисунок 2 – Интерфейс MacOS 1.0 (1984)

Еще одним значимым игроком является компания Novell с OS-Net (1983 год). Основной идеей являлась разработка операционной системы с интегрированными сетевыми функциями. В дальнейшем разработки были использованы в семействе Novell NetWare – серверной версии для Intel-совместимых процессоров. В настоящее время компания осуществляет поддержку ОС семейства SuSE Linux.

Еще одним участником рынка операционных систем, разработки которого внесли значительный след в развитие операционных систем является компания IBM. К её разработкам кроме упомянутой ранее OS/360 относятся передовая на момент разработки система OS/2 (выпущена компаниями Microsoft и IBM в 1987 году и является первой многозадачной операционной системой для персональных компьютеров с процессорами 80286 использующей возможности защищенного режима), современные системы IBM System i, z/OS и z/VM для серверов и мейнфреймов, а также целый ряд разработанных технологии в области хранения данных и сетей.

Отдельно стоит упомянуть семейство операционных систем UNIX и семейство GNU/Linux. История операционной системы UNIX начинается в начале 1970-ых годов в Bell Laboratories. UNIX-системы оказали огромное влияние на развитие операционных систем и их технологии. Упомянем основные ветви этого семейства:

- Семейство System V от AT&T;
- BSD (Berkeley Software Distribution) от Калифорнийского университета Беркли;
- Учебная операционная система MINIX, разработанная Эндрю Таненбаумом (под её влиянием Линус Торвальдс начал разработку Linux);
- GNU/Linux.

Пятое поколение (1990 – настоящее).

Особенности: Мобильные компьютеры.

В 1996 Nokia выпускает коммуникатор N9000 – устройство объединяющее телефон и PDA (Personal Digital Assistant).

В 1997 г. Ericsson вводит понятие «smartphone» для GS88 «Penelope». Для подобных устройств также потребовалось системное программное обеспечение. Наиболее популярная мобильная ОС начального этапа – Symbian OS (консорциум Sony Ericsson, Samsung, Motorola, Nokia).

Другие наиболее известные представители операционных систем для мобильных устройств:

- 2002 – RIM Blackberry OS
- 2007 – Apple iOS для iPhone
- 2008 – Google Android
- 2011 – отказ Nokia от Symbian в пользу Windows Phone (поддержка Windows Phone прекращена 10 декабря 2019 года).

Развитие операционных систем для мобильных платформ продолжается и появляются новые представители этого семейства: Tizen, Harmony OS (Huawei), Aurora (ООО «Открытая мобильная платформа») и др.

Классификация ОС. Выше была упомянута только малая часть существовавших и используемых в настоящее время операционных систем. В виду такого большого их разнообразия необходимо классифицировать операционные системы по ряду критериев. Рассмотрим некоторые из подходов к классификации ОС.

Чаще всего операционные системы классифицируют **по сфере применения** и делят на:

- **Операционные системы общего назначения.** Такие системы могут использоваться для решения широкого спектра задач. К ним можно отнести ОС Microsoft Windows 10, Microsoft Windows 11, GNU/Linux (в редакциях Desktop).
- **Операционные системы специального назначения**. Предназначены для решения отдельного класса задач, например, системы для носимых устройств и встроенных систем, для организации и ведения БД,

для решения задач реального времени и т.д. Например, WatchOS, FreeNAS, Huawei LiteOS.

По **типу вычислительной системы** операционные системы принято делить на:

- ОС мэйнфреймов. Как правило отличаются возможностью работы в трёх основных режимах:
  - Режиме пакетной обработки (выполнение заданий в неинтерактивном режиме при отсутствии непосредственного взаимодействия с пользователем);
  - Режиме обработки транзакций (обработка большого количества небольших запросов в единицу времени, например, банковские платежные системы);
  - Режиме разделения времени (позволяют одновременно большому количеству пользователей выполнять свои задания на одной и той же вычислительной системе; очень часто такие системы называются терминальными серверами);

Примером подобных систем могут служить системы IBM z/OS и z/VM.

- Серверные ОС. Характеризуются возможностью работы на серверном оборудовании и обслуживанием множества одновременно подключившихся клиентов. Позволяют организовать совместную работу и совместное использование устройств и данных. К типичным представителям этого типа операционных систем можно отнести семейство BSD (FreeBSD, NetBSD), серверные варианты GNU/Linux, операционные системы семейства Microsoft Windows Server (Server 2019, Server 2022).
- Многопроцессорные ОС. Отдельная группа серверных операционных систем, предназначенная для использования на суперкомпьютерной технике. К таким системам можно отнести также операционные системы кластерных архитектур. Традиционно представлены кастомизированными вариантами на базе UNIX или GNU/Linux.
- ОС персональных компьютеров (Windows 10, Windows 11, MacOS, Linux в Desktop редакциях).
- Встроенные ОС (Embedded operating system). Предназначаются для встроенных компьютерных систем, работают как правило в условиях дефицита ресурсов. Например, OpenWRT, LiteOS, VW.OS.
- ОС смарт-карт. (Multos, SLCOS, TrustSec OS)

Операционные системы реального времени. Операционные системы, предназначенные для обеспечения требуемого уровня сервиса за заданный промежуток времени. Примеры: VxWorks, QNX.

Главные параметры время и реактивность системы.

ОС с жесткой системой реального времени – некоторое действие должно произойти в конкретный момент времени или внутри заданного диапазона.

ОС с гибкой системой реального времени – допустимы время от времени пропуски сроков выполнения операции.

По режиму обработки задач:

- Однопрограммные (классические варианты DOS);
- Мультипрограммные (многозадачные) (большинство современных операционных систем).

По способу взаимодействия с компьютером:

- Диалоговые системы (интерактивные системы);
- Пакетные системы

По организации работы в диалоговом режиме с пользователями:

- Однопользовательские ОС;
- Мультитерминальные ОС

По основному архитектурному принципу:

- **Макроядерные** ОС (**монолитные**) (WindowsNT, Linux)
- **ОС модели клиент-сервер** (**микроядерные** ОС) (Mach, QNX)
- **Многоуровневые** операционные системы (THE - Technische Hogeschool Eindhoven, MULTICS)
- **Виртуальные машины** (VM/370(CP/CMS))
- **Экзоядро** (exokernel)

Перспективными подходами к операционным системам являются следующие:

- Архитектура микроядра
- Многопоточность
- Симметричная многопроцессорность
- Распределенные ОС
- Объектно-ориентированный и компоненто-ориентированный дизайн

Контрольные вопросы:

1. Перечислите основные функции операционной системы.
2. Какой тип интерфейса операционных систем возник раньше: графический интерфейс пользователя или интерфейс командной строки?
3. К какому поколению развития вычислительной техники относят появление графического интерфейса пользователя?
4. К какому поколению развития вычислительной техники относят появление основных технологии операционных систем: подкачки, режима разделения времени, защиты памяти?
5. На основании приведенных критериев классификации операционных систем определите к каким пунктам будет относится операционная система Вашего компьютера.

§ 2. ПРОЦЕССЫ

Процесс (англ. **process**) – ключевое понятие любой современной многозадачной ОС, описывающее выполнение программы.

Напомним кратко терминологию: после появления идеи разработки какой-либо программы или приложения, автор продумывает *алгоритм* её работы, после чего кодирует его на выбранном языке программирования, получая *файлы исходного кода*, которые после этого переводятся транслятором в *машинный код*. Полученный машинный код компонуется и на выходе получается *исполнимый файл*, соответствующий применяемому в используемой операционной системе формату (например, ELF в GNU/Linux или PE в Microsoft Windows). При запуске этого файла на исполнение, операционная система создаёт ряд вспомогательных структур, необходимых для учёта выделяемых исполняемой программе ресурсов (памяти, открытых файлов, устройств и т.д.) и настроек её выполнения (переменных окружения, приоритета, лимитов потребляемого времени центрального процессора и т.д.). В этом случае мы можем говорить о создании в операционной системе *процесса*.

По-другому, процесс – объект, обладающий собственным независимым **виртуальным адресным пространством**, в котором могут размещаться код и данные, защищенные от других процессов.

Адресное пространство – список адресов в памяти, которые процесс может прочесть и в которые он может писать. Адресное пространство содержит код программы, данные и ее стек.

К ресурсам, например, принадлежащим процессу можно отнести:

- Виртуальное адресное пространство (включая код загруженных DLL);
- Один или несколько потоков исполнения;
- Строки, содержащие информацию об окружении;
- дескрипторы открытых файлов и др.

Одна из ключевых целей процессов – обеспечение необходимой информации для реализации режима многозадачности.

Режим выполнения процессов, при котором процессор, переключаясь с одного процесса на другой, обеспечивает их параллельное (псевдопараллельное) выполнение, называется **многозадачностью** или **мультипрограммным режимом работы** (англ. **multiprogramming).**

Следует отметить, что в операционных системах, не использующих режим многозадачности (т.н. однозадачных), понятие процесса как правило не применяется.

Общим подходом для реализации хранения необходимой информации и управления ресурсами в операционных системах является применение соответствующих таблиц: для распределения памяти, устройств ввода-вывода, файлов и т.д. (Следует понимать, что термин таблица не обязательно обозначает именно двумерный массив: это может быть любая удобная для хранения структура данных, например, односвязный список).

Для процессов необходимая информация храниться в **таблицах процессов** (англ. **process tables**). Таблица процессов состоит из структур – **блоков управления процессами** (англ. process control block, PCB; иногда используются термины дескриптор задачи или дескриптор процесса) – по одной на каждый существующий в данный момент процесс.

PCB содержит различные атрибуты процесса. Список атрибутов процесса может включать в различных операционных системах следующие:

- идентификатор процесса (Process ID, PID);
- идентификатор родительского процесса (Parent Process ID, PPID);
- идентификаторы пользователя (UID) и группы (GID), которым принадлежит этот процесс;
- Состояние процесса (выполняющийся, заблокированный и т.д.);
- Приоритет;
- Информация о планировании;
- Данные межпроцессного взаимодействия (IPC);
- Маркер доступа;
- Информация о расположении в памяти (таблицы сегментов/страниц);
- Данные управления ресурсами (открытые файлы и др.).

Множество, в которое входит код программы, её данные, стек и атрибуты часто обозначают термином **образ процесса** (англ. process image).

Иерархия процессов. Процесс может запускать другие процессы. Когда один процесс порождает другой, то порождающий процесс называют **родительским**, или **предком** (parent), а порождаемый процесс – **дочерним**, или **потомком** (child).

В некоторых операционных системах родительский и дочерний процессы в ходе исполнения остаются связанными между собой определенным образом. Дочерний процесс также может, в свою очередь, создавать процессы, формируя **иерархию** процессов. Например, в ОС UNIX/Linux процесс и его процессы-потомки образуют группу процессов. Это может использоваться для более удобного управления процессами. Например, сигналы доставляются всем членам группы. Примером группы процессов могут служить процессы потомки init/systemd.

В Microsoft Windows все процессы относительно иерархии равноправны (контроль с помощью описателя дочернего процесса, который может быть произвольно передан другому процессу), но могут быть объединены в объекты задач (job objects), что позволяет управлять ими как группой и устанавливать лимиты ресурсов.

Классификация процессов. По наличию интерфейса пользователя могут быть разделены на:

Интерактивные процессы (**приложения**) – выполняют взаимодействие с пользователем (имеют графический интерфейс или интерфейс командной строки).

Фоновые процессы (не взаимодействуют с конкретными пользователями напрямую). В различных операционных системах такие процессы обозначаются различными терминами. Например, в ОС UNIX/Linux они называются демонами (daemon), в ОС семейства Microsoft Windows – службами (service). В последнее время, также, можно встретить в терминологии прямую кальку с английского – сервис.

Еще одна классификация процессов разделяет их на:

Системные процессы – процессы, которые выполняют код операционной системы и относятся к операционной системе в целом. Такие процессы занимаются решением служебных задач: управлением устройствами, распределением памяти, свопингом. Как правило создаются ядром операционной системы.

Пользовательские процессы – процессы, которые выполняют некоторый исполнимый файл, запущенный конкретным пользователем. Срок выполнения таких процессов ограничен временем существования сеанса этого пользователя в системе. Как будет показано в разделе посвященном ядру операционной системы пользовательские процессы могут исполняться как в режиме пользователя, так и в режиме ядра (при выполнении системного вызова).

Состояния процессов. В ходе исполнения процессы могут находиться в различных состояниях. Существуют различные подходы к выделению таких состояний. Некоторые авторы выделяют более десятка различных состоянии. К основным состояниям можно отнести следующие:

1 **Работающий** (в этот конкретный момент использующий процессор).

2 **Готовый** к работе (процесс временно приостановлен, чтобы позволить выполняться другому процессу).

3 **Заблокированный** (процесс не может быть запущен прежде, чем произойдет некое внешнее событие).

Переходы между этими состояниями представлены на рисунке ниже (рисунок 3).

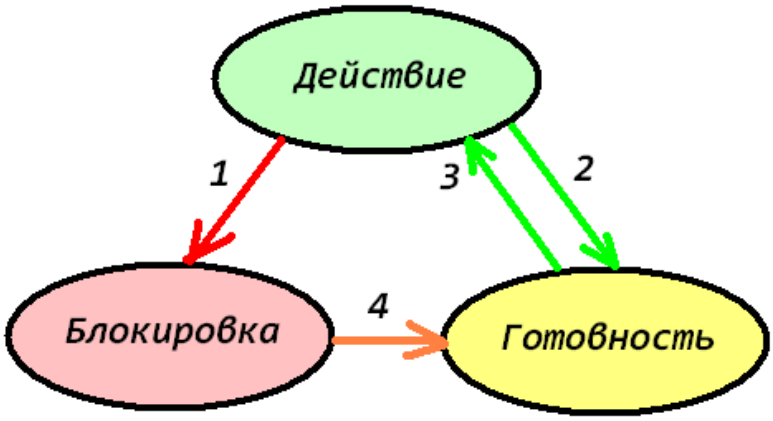

Рисунок 3 – Диаграмма переходов состояний

1 Процесс блокируется (например, в ожидании завершения операции чтения данных).

2 Процесс вытесняется с центрального процессора, и планировщик выбирает следующий процесс для исполнения.

3 Процесс вновь загружается на исполнение.

4 Процесс разблокируется (например, ожидаемая операция завершилась и доступны прочитанные данные).

Ситуации создания процессов. Процессы могут создаваться в следующих основных случаях:

– Инициализация системы (подробнее см. лекцию по управлению устройствами);

– Выполнение изданного работающим процессом системного запроса на создание процесса;

– Запрос пользователя на создание процесса;

– Инициирование пакетного задания.

Системные вызовы для создания процесса. Существуют различные подходы к реализации интерфейса создания процессов в операционных системах. Рассмотрим два варианта создания процессов: в операционных системах семейства Microsoft Windows и в операционных системах UNIX/Linux.

Для ОС семейства Microsoft Windows основным средством создания процессов является функция CreateProcess. Также доступны такие варианты как CreateProcessAsUser, CreateProcessWithLogonW, CreateProcessWithTokenW, ShellExecute, ShellExecuteEx и обозначенная, как используемая для совместимости со старым 16-битным кодом функция WinExec.

Приведенные функции берут на себя всю работу по созданию процесса: проверку приемлемости параметров вызова, открытие файла образа для загрузки кода и данных процесса, создание объекта процесса исполняющей системы Windows, выделение адресного пространства процессу, создание первичного потока исполнения для процесса и начало его исполнения.

Рассмотрим использование CreateProcess:

```
BOOL CreateProcess(
  [in, optional] LPCTSTR lpApplicationName,
  [in, out, optional] LPTSTR lpCommandLine,
  [in, optional] LPSECURITY_ATTRIBUTES lpProcessAttributes,
  [in, optional] LPSECURITY_ATTRIBUTES lpThreadAttributes,
  [in] BOOL bInheritHandles,
  [in] DWORD dwCreationFlags,
  [in, optional] LPVOID lpEnvironment,
  [in, optional] LPCTSTR lpCurrentDirectory,
  [in] LPSTARTUPINFO lpStartupInfo,
  [out] LPPROCESS_INFORMATION lpProcessInformation
);
```

Здесь параметры LPCTSTR lpApplicationName и LPTSTR lpCommandLine – указывают на исполняемую программу и список аргументов командной строки.

Если lpApplicationName не равен NULL, то параметр задает полный или краткий путь к файлу исполняемого модуля вместе с расширением, при этом поиск не выполняется. Обратите внимание на тип данных этого параметра – LP**C**TSTR, – он позволяет задавать литерал в вызове функции.

Если параметр lpApplicationName задан как NULL, то имя исполняемого файла задается первой из лексем, переданных в параметре lpCommanLine. При наличии в этом имени пробелов, имя берётся в двойные кавычки.

Если в параметре lpCommandLine полный путь к исполняемому файлу не указан, то производится его поиск:

1 Каталог модуля текущего процесса.
2 Текущий каталог процесса-родителя.
3 Системный каталог Windows, возвращаемый GetSystemDirectory.
4 Каталог Windows, возвращаемый GetWindowsDirectory.
5 Каталоги, перечисленные в переменной окружения PATH.

Следует заметить, что тип этого параметра LPTSTR, что указывает на то, что он может быть изменён функцией.

Параметры LPSECURITY_ATTRIBUTES lpProcessAttributes и LPSECURITY_ATTRIBUTES lpThreadAttributes задают указатели на структуры атрибутов защиты процесса и потока. (по умолчанию - NULL);

BOOL bInheritHandles – наследует ли новый процесс открытые наследуемые дескрипторы (файлов, отображений файлов и др.) из вызывающего процесса;

DWORD dwCreationFlags – задает одно или несколько флаговых значений, например:

CREATE_SUSPENDED – основной поток будет создан в приостановленном состоянии;

DETACHED_PROCESS и CREATE_NEW_CONSOLE – взаимоисключающие значения – создание процесса у которого отсутствует консоль или будет своя консоль соответственно. Если ни один из флагов не указан – новый процесс наследует консоль родительского процесса;

CREATE_NO_WINDOW – для консольного приложения не будет создаваться окно при его запуске.

Также в этом параметре может быть указан класс приоритета запускаемого процесса.

LPVOID lpEnvironment – блок параметров настройки окружения нового процесса. Значение по умолчанию – NULL (окружение родительского процесса).

LPCTSTR lpCurrentDirectory – указатель на строку, содержащую путь к текущему каталогу процесса. Значение NULL по умолчанию задает рабочий каталог и рабочий диск родительского процесса.

LPSTARTUPINFO lpStartupInfo – указатель на структуру, которая описывает внешний вид основного окна и содержит дескрипторы стандартных устройств нового процесса. Для ее задания можно использовать два варианта:

– Использовать информацию родительского процесса, извлеченную при помощи GetStartupInfo;

– Заполнить структуру STARTUPINFO перед вызовом CreateProcess;

Важно: даже не используя STARTUPINFO следует обнулить эту структуру и задать в поле cb ее размер, например, следующим способом:

```
…
STARTUPINFO si;
ZeroMemory(&si, sizeof(si));
si.cb = sizeof(si);
…
```

или используя синтаксис инициализаторов для структуры:

```
…
STARTUPINFO si = {0};
si.cb = sizeof(si);
…
```

LPPROCESS_INFORMATION lpProcessInformation – указатель на структуру, возвращающую значения дескрипторов и глобальных идентификаторов процесса и его первичного потока. В операционной системе эта структура задана следующим образом:

```
typedef struct _PROCESS_INFORMATION {
    HANDLE hProcess;
    HANDLE hThread;
    DWORD dwProcessId;
    DWORD dwThreadId;
} PROCESS_INFORMATION;
```

Как было указано выше, каждый процесс имеет уникальный идентификатор – Process Identifier или сокращенно PID. Помимо этого, процесс, как объект ядра, задается дескриптором – `HANDLE`. Количество дескрипторов, в отличие от ProcessID может быть произвольным, причем с различными правами доступа к процессу. Можно получить дескриптор процесса различными способами:

– Для дочернего процесса его дескриптор можно получить из структуры `PROCESS_INFORMATION`;

– Для текущего процесса можно получить его псевдодескриптор (имеет смысл только в текущем процессе) с помощью `GetCurrentProcess` (и его PID с помощью `GetCurrentProcessId`);

– Для произвольного процесса дескриптор можно получить по PID этого процесса с помощью функции `OpenProcess`:

```
HANDLE OpenProcess(
  DWORD dwDesiredAccess,
```

```
  BOOL bInheritHandle,
  DWORD dwProcessId
);
```

Здесь dwDesiredAccess задает желаемые права доступа к процессу, например:

SYNCHRONIZE – разрешает использование дескриптора в функциях ожидания завершения процесса;

PROCESS_TERMINATE – разрешает принудительное завершение процесса;

PROCESS_QUERY_INFORMATION – разрешает получать информацию о процессе с помощью GetExitCodeProcess и GetPriorityClass;

PROCESS_ALL_ACCESS – все флаги доступа к процессу.

Приведём простейший пример создания процесса – запуск текстового редактора Блокнот:

```
#include <windows.h>
#include <iostream>

int main()
{
     STARTUPINFO si = { 0 };
     si.cb = sizeof(si);
     PROCESS_INFORMATION pi;
     if (!CreateProcess(L"C:\\Windows\\notepad.exe", NULL,
NULL, NULL, FALSE, 0, NULL, NULL, &si, &pi))
     {
     std::cerr << "process creation error: " << GetLastEr-
ror() << std::endl;
     }
     system("pause");
     return 0;
}
```

Запуск приложения, ассоциированного с расширением указанного файла в Windows, выполняется при помощи функции ShellExecute или ShellExecuteEx:

```
HINSTANCE ShellExecute(
      HWND hwnd,
    LPCTSTR lpOperation,
    LPCTSTR lpFile,
    LPCTSTR lpParameters,
    LPCTSTR lpDirectory,
    INT nShowCmd
);
```

Здесь,

HWND hwnd – дескриптор окна родительского приложения (можно задать равным NULL);

LPCTSTR lpOperation – операционная строка, задающая действие выполняемое с файлом (каталогом): открытие в приложении, печать или открытие каталога ("open" = NULL, "print", "explore");

LPCTSTR lpFile – указатель на строку, содержащую имя файла, по ассоциации с которым будет запущено приложение;

LPCTSTR lpParameters – указатель на строку, содержащую параметры командной строки (обычно NULL);

LPCTSTR lpDirectory – указатель на строку, содержащую текущий каталог для запускаемого приложения;

INT nShowCmd – способ отображения окна, совпадающий с константой для ShowWindow.

Существует аналог данной функции: ShellExecuteEx, отличающийся от ShellExecute способом задания параметров. Для данного системного вызова они задаются через структуру SHELLEXECUTEINFO.

Функция WinExec в настоящее время объявлена Microsoft устаревшей и, согласно документации MSDN, должна использоваться только для совместимости с 16-битным кодом. Тем не менее, так как она в конечном итоге обращается к функции CreateProcess, в некоторых случаях допускается ее использование:

```
UINT WinExec(
  LPCSTR lpCmdLine,
  UINT uCmdShow
);
```

Здесь,

LPCSTR lpCmdLine – имя программы (полный путь или короткое имя);

UINT uCmdShow – способ отображения окна, совпадающий с параметром функции ShowWindow, изученной в предшествующих дисциплинах.

Несколько иначе реализовано создание новых процессов в операционных системах UNIX/Linux. В этих системах процесс создается как точная копия родительского процесса системным вызовом fork. Обычно, (но не обязательно) после выполнения fork**()**, дочерний процесс выполняет один из exec-вызовов (execl или подобный) для изменения своего *образа памяти* и запуска новой программы (рисунок 4).

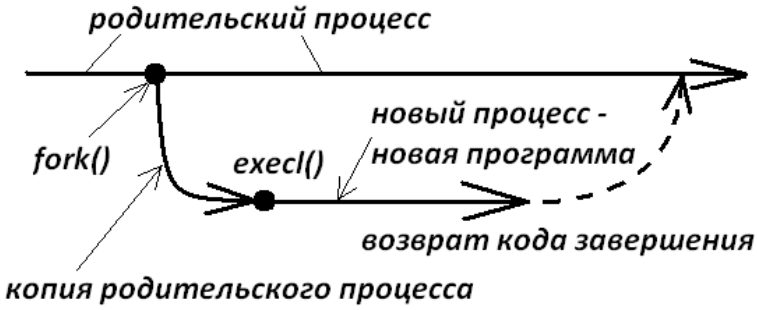

Рисунок 4 – диаграмма запуска процесса в Linux

Первый этап работы по созданию процесса выполняет функция fork():

```
#include <unistd.h>
      pid_t fork(void);
```

(«ветвление», «вилка») – создает дубликат вызываемого процесса. При этом fork возвращает управление в оба процесса – и в родительский процесс, и в дочерний процесс. Родительскому процессу возвращается идентификатор дочернего процесса, а дочернему процессу – значение 0. Более подробно об этой функции можно почитать во встроенной справочной системе по команде:

```
man 2 fork
```

После этого выполняется системный вызов execve или одна из библиотечных функций семейства exec, приведенная ниже. Вызов execve описан следующим образом:

```
#include <unistd.h>
int execve(const char* PATH, const char** ARGV, const char** ENVP);
```

Здесь PATH – путь к исполняемому файлу;

ARGV – массив аргументов программы (ARGV [0] – имя программы или что-либо другое, но не фактический аргумент);

ENVP – массив задающий окружение программы.

Кроме execve могут быть использованы:

```
#include <unistd.h>
int execl (const char* PATH, const char* ARG, …);
int execle (const char* PATH, const char* ARG, …, const char** ENVP);
int execlp (const char* PATH, const char* ARG, …);
int execv (const char* PATH, const char** ARGV);
int execvp (const char* PATH, const char** ARGV);
```

Общий принцип формирования имён этих функции можно описать следующим образом:

execX[Y] (…);

Где X задается из множества {l,v}, а Y - из {e,p}. Расшифровка указанных символов в имени:

l – list – означает, что список аргументов передается не в массиве, а в виде отдельных аргументов вызова и заканчивается NULL;

```
execl("/usr/bin/mc", "/usr/bin/mc", "/home/user", NULL);
```

v – vector – аргументы программы передаются в едином массиве ARGV, последним элементом которого является NULL;

```
char* argv[] = {"/usr/bin/mc", "/home/user", NULL};
execv("/usr/bin/mc", argv);
```

e – environment – передается окружение программы (в формате строк вида <имя_пер._среды>=<значение>);

p – path – поиск имени исполняемого файла будет проводится в соответствии с содержимым переменной окружения PATH, если имя файла не начинается с «/».

Для идентификации запущенного процесса в UNIX/Linux используется уникальный идентификатор процесса PID (Process IDentifier, тип данных pid_t). Также к процессу привязан атрибут PPID (Parent Process IDentifier) – идентификатор родительского процесса.

Получить информацию о текущем процессе можно с помощью функций:

```
#include <unistd.h>
#include <sys/types.h>  // для типа pid_t
pid_t getpid(void);
pid_t getppid(void);
```

Подведём итоги по отличию моделей создания процессов:

- В Windows отсутствует аналог fork: CreateProcess = fork + exec;
- Пути доступа в UNIX/Linux определяются ТОЛЬКО переменной среды PATH;
- В Windows процессы идентифицируются дескрипторами и PID, в UNIX/Linux – дескрипторы отсутствуют;
- В UNIX/Linux процесс и его потомки образуют **группу процессов** (например, группа процессов с родителем init);

В Windows все процессы относительно **равноправны** (контроль с помощью дескриптора дочернего процесса), но могут быть объединены в **объекты задач** (job objects), что позволяет управлять ими как группой и устанавливать лимиты ресурсов.

Завершение процессов. Можно выделить 4 ситуации завершения процесса:

- Обычный выход (преднамеренно);
- Выход по ошибке (преднамеренно);
- Выход по неисправимой ошибке (непреднамеренно);
- Уничтожение другим процессом (непреднамеренно).

Завершение процесса в Windows. Нормальное завершение работы процесса – завершение выполнения функции main (wmain, _tmain) для консольного приложения или функции WinMain (wWinMain) – для GUI-приложения. По окончании работы процесс (поток) может вызвать функцию ExitProcess указав в качестве параметра **код возврата** (exit code):

```
VOID ExitProcess(UINT uExitCode);
```

Это равносильно выполнению в программе оператора return uExitCode.
Другой процесс может определить этот код возврата:

```
BOOL GetExitCodeProcess(
  HANDLE hProcess,
  LPDWORD lpExitCode
);
```

Здесь дескриптор hProcess должен иметь права доступа PROCESS_QUERY_INFORMATION, а lpExitCode указывать на переменную, которая принимает код завершения. В случае, если целевой процесс еще не завершил работу возвращается значение соответствующее константе STILL_ACTIVE.

Принудительное завершение другого процесса:

```
BOOL TerminateProcess(
  HANDLE hProcess,
  UINT uExitCode
);
```

Завершающий процесс должен получить дескриптор hProcess с правами доступа PROCESS_TERMINATE.

Завершение процессов в UNIX/Linux. Программа может вызвать:

- оператор return exit_code;
- функцию из библиотеки C

```
void exit(int exit_code);
```

- системный вызов:

```
void _exit(int exit_code);
```

Так же, как и в Windows, по соглашению нулевой код статуса завершения указывает, что процесс завершился успешно, ненулевой – неудачно.

Принудительное завершение указанного процесса выполняется отправкой определенного **сигнала**:

```
#include <sys/types.h>
#include <signal.h>
int kill (pid_t pid, int signum);
```

Здесь pid – идентификатор процесса, который нужно завершить;

signum – номер сигнала, указывающий что необходимо сделать. Для завершения процесса чаще всего применяются варианты:

- SIGKILL – безусловное завершение,
- SIGTERM - «просьба» завершиться (действие может быть отложено или проигнорировано).

Если в качестве signum передать 0, то kill проверяет имеет ли вызывающий процесс необходимые полномочия (0 = да, не 0 – нет).

Код статуса завершения программы сообщается родительскому процессу с помощью:

```
#include <sys/wait.h>
pid_t wait (int* exit_status);
pid_t waitpid (pid_t child_pid, int* exit_status, int options);
```

Возвращаемое значение - PID дочернего процесса. Если процесс еще не завершен – выполнение родительского процесса будет **приостановлено.**

wait ожидает завершения **любого** дочернего процесса.

waitpid – позволяет **указать какой** из порожденных процессов надо ожидать (child_pid=-1 – любой, >0 – дочерний процесс с заданным PID).

Зомби

Для передачи кода возврата родительскому процессу дочерний процесс после завершения переводится в состояние, обозначаемое как «зомби» – все структуры освобождаются, но описатель процесса не закрывается, пока родительский процесс не вызовет wait или не завершится.

Если родительский процесс завершается раньше, то код возврата принимает процесс №1 в системе.

Лабораторная работа. Процессы в Linux

Задание 1 (разминка). Разработайте приложение, выводящее на экран свой PID и PPID. Определите, какое из приложений выполняется процессом с идентификатором равным PPID. Продемонстрируйте (например, при помощи команды ps в режиме отображения дерева процессов), что Ваша программа выводит корректную информацию (процесс с идентификатором PPID действительно является родительским процессом для Вашего приложения).

Задание 2. Разработайте приложение, запускающее новый процесс, исполняющий другую программу (например, Leafpad). При запуске программы используйте функцию, соответствующую Вашему варианту:

1) execl;
2) execv;
3) execle;
4) execve;
5) execlp;
6) execvp;
7) fexecve;

При запуске программы используйте функцию, соответствующую Вашему варианту и передайте ей в качестве параметра имя файла, который запускаемая программа может открыть (например, для Leafpad – имя текстового файла):

8) execl;
9) execv;
10) execle;
11) execve;
12) execlp;
13) execvp;
14) fexecve.

Указание: предусмотрите в программе использование возможностей используемой функции exec.

Задание 3. Разработайте приложение, передающее при помощи системного вызова kill, указанный для Вашего варианта сигнал некоторому процессу, заданному его PID.

1) Аварийное завершение процесса;

2) Возобновление работы приостановленного ранее процесса;
3) Завершение;
4) Приостановку исполнения процесса;
5) Изменение размеров окна терминала;
6) Ввод с терминала символа прерывания;
7) Аппаратная ошибка, не связанная с памятью;
8) Арифметическая ошибка;
9) Падение напряжения питания/перезапуск;
10) Ошибка доступа к памяти;
11) Сигнал, определяемый пользователем;
12) Истекло время таймера.
Продемонстрируйте работу Вашей программы.

Указания: **1)** при написании программ предусмотрите обработку возможных ошибок.

2) Для получения информации о системном вызове или команде используйте команду man с именем вызова или команды в качестве параметра. Например:

```
man exec
```

В случае, если существует несколько одноименных объектов команде man можно указать требуемый раздел руководства, использовав следующий формат:

```
man n <вызов>,
```

где n – задает секцию справочного руководства и может принимать следующие значения:

1 – исполнимые программы или команды оболочки (shell);
2 – системные вызовы;
3 – библиотечные вызовы (например, printf);
4 – специальные файлы (например, файлы устройств);
5 – форматы файлов и соглашения;
и т.д.

Например, для вывода справки по системному вызову fork:

```
man 2 fork
```

3) Для получения списка процессов и другой информации о них могут быть использованы команды ps (консольная команда), top (или ее графическая версия gtop).

Лабораторная работа. Процессы в Windows

Задание 1. Разработайте приложение, демонстрирующее создание одного дочернего процесса. В качестве запускаемых приложений используйте приложение из списка: MS Word, MS Excel, MS Access, MS Paint, 7-zip.

Задание 2. Модифицируйте проект из задания 1 таким образом, чтобы запуск приложения происходил с использованием командной строки и в приложении открывался указанный Вами документ.

Задание 3. Дан некоторый файл с данными (документ MS Word, рабочая книга MS Excel, точечный рисунок, и т.д.). Разработайте приложение, демонстрирующее создание процесса на основе зарегистрированной в системе ассоциации с расширением заданного файла данных.

Задание 4. Модифицируйте проект из задания 3 таким образом, чтобы при его запуске открывалось окно Explorer'а с открытым каталогом C:\Windows\.

Задание 5. Напишите приложение, считывающее указанный пользователем PID некоторого процесса, а затем принудительно завершающее этот процесс (упрощенный аналог команды taskkill).

Задание 6* (для самостоятельной работы). Разработка графической оболочки для программы UPX. Разработайте графическое приложение – управляющую оболочку для программы UPX, – упаковщика исполнимых файлов. Приложение должно позволять: выбрать сжимаемый файл, выбрать режим работы – компрессию или декомпрессию файла, выбрать степень сжатия файла, показать результат выполнения операции – успех (код возврата UPX = 0), неудача (код возврата UPX=1), предупреждение (код возврата UPX = 2).

§ 3. ПОТОКИ

В классических операционных системах каждый процесс имеет адресное пространство и один поток управления. Такая модель процесса базируется на двух независимых концепциях:

- группирование ресурсов
- выполнение программы.

С точки зрения исполнения программы, процесс – поток исполняемых команд или просто **программный поток**.

С точки зрения группирования ресурсов процесс – набор файловых ресурсов, данных, дескрипторов используемых объектов ядра, адресного пространства и т.д.

Если эти понятия разделить, появляется понятие потока (thread). Соответственно, можно считать, что **процессы** – это контейнеры ресурсов, а **потоки** – это объекты, поочередно выполняющиеся на центральном процессоре.

Потоки **разделяют ресурсы процесса** и выполняются **только в рамках** какого-либо процесса. При этом различные потоки могут выполнять **один и тот же** код (одну и ту же функцию).

При запуске процесса создается один поток – **первичный поток процесса**. В дальнейшем процессы могут создавать дополнительные потоки, обращаясь к соответствующим системным вызовам операционной системы.

Как только завершается **последний** поток – процесс считается завершенным.

Каждый поток характеризуется следующими атрибутами:

- значением счетчика команд процессора;

- содержимым регистров общего назначения;
- собственным стеком вызовов;
- состоянием.

В то же время, все потоки процесса разделяют следующие ресурсы:

- адресное пространство процесса, включая глобальные переменные;
- открытые файлы;
- дочерние процессы;
- сигналы и их обработчики;
- информацию об использовании ресурсов (квоты).

Многопоточностью называется способность операционной системы поддерживать в рамках одного процесса выполнение нескольких потоков.

Подход, при котором каждый процесс представляет собой единый поток выполнения, называется **однопоточным** подходом.

Примеры:

- Один однопоточный процесс: MS-DOS
- Множество однопоточных процессов: классические разновидности UNIX
- Множество многопоточных процессов: OS/2, Linux, Solaris, Mach, Windows NT.

Причины использования дополнительных потоков в приложении:

1) Для сохранения отзывчивости основного потока к командам пользователя, вводу данных и управлению окнами в процессах с графическим интерфейсом пользователя;

2) Для масштабирования производительности приложения в многопроцессорных и многоядерных ЭВМ;

3) Для обеспечения работы приложения, во время приостановки при выполнении операций ввода/вывода.

Порождение и завершение потоков. Вне зависимости от операционной системы, для создания потока исполнения необходимо реализовать два шага:

1) Оформить код, который должен выполняться в отдельном потоке в виде функции, с соответствующей сигнатурой;

2) Обратиться к соответствующей функции прикладного интерфейса программирования операционной системы с указанием адреса запускаемого потока исполнения.

Рассмотрим эти шаги в операционной системе семейства Microsoft Windows:

Точка входа в поток определяется согласно следующему шаблону:

```
DWORD WINAPI ThreadFunc(LPVOID lpThreadParam)
{
     // Добавьте код потока сюда
     return 0;
}
```

При этом имя функции поток исполнения выбирается программистом (в данном случае – ThreadFunc). Парамерт lpThreadParam является указателем на передаваемые в поток в качестве аргумента данные.

Для создания потока применяется следующая функция:

```
HANDLE CreateThread (
     LPSECURITY_ATTRIBUTES lpsa,
     DWORD dwStackSize,
     LPTHREAD_START_ROUTINE lpStartAddress,
     LPVOID lpThreadParam,
     DWORD dwCreationFlags,
     LPDWORD lpThreadId );
```

Здесь *lpsa* – указатель на структуру атрибутов защиты (чаще всего имеет значение по умолчанию – NULL);

dwStackSize – размер стека нового потока в байтах (обычно 1Мб). Значению 0 соответствует размер стека основного потока;

lpStartAddress – указатель на функцию – точку входа в поток (следует напомнить, что в языках C/C++ имя функции задает указатель на эту функцию);

lpThreadParam – указатель на аргумент функции потока;

dwCreationFlags – задает состояние потока после запуска. Может принимать значения:

- 0 – поток запускается сразу же после создания;
- CREATE_SUSPENDED – создание потока выполняется в приостановленном состоянии, для его запуска требуется вызов функции ResumeThread;

lpThreadId – указатель на переменную, получающую идентификатор потока.

В случае успеха CreateThread возвращает дескриптор созданного потока, иначе – NULL.

Приведем пример:

```
// функция - точка входа в поток
DWORD WINAPI MyThread(LPVOID param)
{
     cout << "hello from new thread #" << *((int *) param);
     return 0;
}
int main()
{
     DWORD ThreadID;
     // передаваемый в поток параметр
     int ThreadNumber = 1;
     HANDLE  hThread  =  CreateThread(NULL,  0,  MyThread,
(LPVOID)&ThreadNumber, 0, &ThreadID);
     if (hThread != NULL) {   /* выполняем обработку ошибки
*/   }
     system("pause");
     return 0;
}
```

Исполнение потока завершается в следующих ситуациях:

1. Функция потока возвращает управление вызовом оператора return (лучший вариант);
2. Поток завершает свое выполнение с помощью ExitThread;
3. Сторонний поток завершает текущий поток с помощью TerminateThread;
4. Завершается процесс, запустивший данный поток (при этом все потоки этого процесса завершаются).

Функции для работы с потоками практически полностью дублируют API работы с процессами:

Завершение текущего потока:

```
void ExitThread( DWORD dwExitCode );
```

Принудительное завершение указанного потока:

```
BOOL TerminateThread( HANDLE hThread, DWORD dwExitCode );
```

Получение кода возврата указанного потока:

```
BOOL GetExitCodeThread( HANDLE hThread, LPDWORD lpExitCode
);
```

При этом для идентификации потоков, аналогично процессам в Windows, используется две величины: идентификатор потока (TID, Thread Identifier) и дескриптор потока (HANDLE). Для работы с ними доступны следующие функции:

GetCurrentThread – возвращает дескриптор текущего потока;

GetCurrentThreadId – возвращает идентификатор текущего потока;

GetThreadId – возвращает идентификатор потока по известному дескриптору;

OpenThread – создает дескриптор потока по известному идентификатору.

Потоки в GNU/Linux. Современная реализация потоков в Linux (ядра 2.6-6.x) базируется на библиотеке Native POSIX Thread Library (NPTL), соответствующей стандарту POSIX.

Основные вызовы, относящиеся к этой библиотеке описаны в заголовочном файле <pthread.h>. В ней также определен основной тип, идентифицирующий поток – pthread_t.

Принципы работы с потоками соответствуют, рассмотренным выше для операционных систем Windows. Точка входа в поток в Linux задается функцией со следующим прототипом:

```
void* ThreadFunc (void* arg);
```

Создание потока выполняется при помощи функции pthread_create:

```
#include <pthread.h>
int pthread_create (
     pthread_t* thread_id,
     const pthread_attr_t* attr,
     void* (*start_fn)(void*),
     void* arg );
```

Здесь thread_id – параметр, принимающий идентификатор создаваемого потока;

attr – атрибуты потока или NULL, если используются атрибуты по умолчанию;

start_fn – адрес функции, реализующей поток;

arg – аргумент, передаваемый функции потока.

В случае успешного выполнения функция pthread_create возвращает 0, иначе – ненулевое значение.

Рассмотрим пример запуска потока:

```
// функция, содержащая код потока
void* my_thread(void* arg)
{
    printf("поток № %d работает\n", *((int*)arg) );
}
int main()
{
     pthread_t tid;
     int thread_num = 1;
if(pthread_create(&tid,          NULL,          my_thread,
(void*)&thread_num)) {
     /* выполняем обработку ошибки */
}
     // если всё хорошо – ждём завершения работы потока
     pthread_join (tid, NULL);
     return 0;
}
```

При сборке многопоточного приложения, использующего библиотеку POSIX Threads (pthread) ее следует явно подключить к программе, указав компоновщику опцию –lpthread (или просто –pthread). Например,

```
g++  myProgram.cpp  –lpthread  –o  Run.Me
```

Завершение потока может быть выполнено следующими способами:

1. Возвратом управления из функции потока (оператор return);
2. Поток сам завершает свое выполнение вызовом pthread_exit;
3. Другой поток может завершить текущий поток вызовом pthread_cancel;
4. Завершение работы процесса;

Завершение текущего потока:

```
void pthread_exit(void* status);
```

Здесь status – указывает на код возврата из потока;

Отмена указанного потока:

```
int pthread_cancel(pthread_t tid);
```

При удачном завершении возвращает 0, иначе – не нулевое значение, при этом код возврата завершенного потока задается константой PTHREAD_CANCELED

Ожидание завершения потока и получение кода возврата:

```
int pthread_join(pthread_t tid, void** status);
```

Действие: блокирует вызывающий поток до тех пор, пока не завершится поток с идентификатором tid и помещает его код возврата в переменную status. При удачном завершении возвращает 0.

Идентификация потоков. Получение идентификатора текущего потока может быть выполнено следующим образом:

```
pthread_t pthread_self (void);
```

Сравнение идентификаторов двух потоков:

```
int pthread_equal (pthread_t TID1, pthread_t TID2);
```

Если TID1 и TID2 относятся к одному и тому же потоку, то функция возвращает ненулевое значение, если к разным потокам – 0.

Состояния потоков. В многопоточных операционных системах рассматриваются следующие основные состояния потоков (рисунок 5):

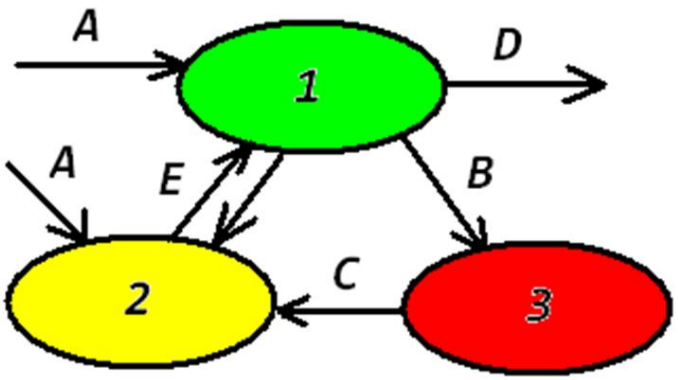

Рисунок 5 – диаграмма состоянии потоков

- Состояние выполнения потока (running state) – поток выполняется процессором;
- Состояние готовности (ready state) – поток может выполняться и ожидает освобождения процессора;
- Состояние блокировки (wait state) (так же говорят о блокированных (blocked) или спящих (sleeping) потоках) – поток выполняет функцию ожидания несигнализирующих объектов или объектов синхронизации, завершения операции ввода-вывода и т.д.

Основные действия с потоками, приведенные на диаграмме:

A. Порождение;

B. Блокирование;

C. Разблокирование – в таком случае говорят, что поток «пробуждается» (wakes);

D. Завершение.

E. Выбор планировщиком следующего потока – по истечению кванта времени или при вызове Sleep(0).

Рассмотрим программные методы перевода потоков между состояниями. Для перевода потока в приостановленное состояние в операционной системе Windows используется следующая функция:

```
DWORD SuspendThread (HANDLE hThread);
```

Для возобновления выполнения используется:

```
DWORD ResumeThread (HANDLE hThread);
```

Оба вызова принимают дескриптор управляемого потока. Возвращают предыдущее значение счетчика приостановок в случае, если завершаются успешно, или иначе - 0xFFFFFFFF.

Указанная ниже функция

```
VOID Sleep (DWORD dwMilliseconds);
```

позволяет потоку отказаться от полученного процессорного времени и перейти из состояния выполнения в состояние ожидания, которое будет длится в течение заданного промежутка времени. По истечении периода ожидания планировщик переводит поток в состояние готовности.

Здесь, dwMilliseconds – интервал ожидания в миллисекундах, или константа INFINITE – бесконечный промежуток ожидания или 0 – отказ потока от оставшейся части отведенного кванта времени.

Плюсы использования потоков

- Создание нового потока в уже существующем процессе занимает **намного меньше времени**, чем создание совершенно нового процесса.
- Поток можно завершить **намного быстрее**, чем процесс.
- Переключение потоков в рамках одного и того же процесса происходит **намного быстрее**.
- При использовании потоков **повышается эффективность обмена информацией** между двумя выполняющимися программами.

Преимущества применения потоков в программировании:

- Одновременная работа в приоритетном и фоновом режимах в приложении;
- Асинхронная обработка;
- Навязанная модульная структура программы.

Недостатки потоков:

- Потоки усложняют программную модель;
- Программные потоки совместно используют большое количество структур данных, что может приводить к ошибкам синхронизации, таким как ситуация гонки (race condition);
- Усложняется обработка ошибок;
- Более сложно контролируется проблема с переполнением стеков.

Лабораторная работа. Потоки в Linux

Задание 1. Разработайте приложение, демонстрирующее многопоточность ОС Linux: запустите несколько потоков исполнения. В коде потоков реализуйте бесконечный цикл (например, `while(1);`). Продемонстрируйте их параллельное выполнение (например, `ps -T ax`). Предусмотрите в приложении **корректное** завершение дополнительных потоков исполнения перед выходом из программы.

Задание 2. По вариантам

1) Разработайте многопоточное приложение для поиска максимального элемента в двумерном массиве из MxN элементов (0<M<10000, 0<N<10000) в K потоков, выполняющихся параллельно (1<K<1000). Значение K вводиться пользователем.

В коде программы предусмотрите возможность автоматического заполнения массива тестовыми данными и проверки результата последовательным поиском.

2) Разработайте многопоточное приложение для сложения двух матриц из MxN элементов (0<M<10000, 0<N<10000) в M потоков, выполняющихся параллельно. Значения M, N вводятся пользователем.

В коде программы предусмотрите возможность автоматического заполнения матриц тестовыми данными и проверки результата однопоточным сложением.

3) Разработайте многопоточное приложение для подсчета в двумерном целочисленном массиве из MxN элементов (0<M<10000, 0<N<10000) количества отрицательных чисел в N потоков, выполняющихся параллельно. Значения M, N вводятся пользователем.

В коде программы предусмотрите возможность автоматического заполнения массива тестовыми данными и проверки результата однопоточным поиском.

4) Разработайте многопоточное приложение для умножения двух матриц из NxN элементов (0<N<10000) в N^2 потоков, выполняющихся параллельно. Значение N вводится пользователем.

В коде программы предусмотрите возможность автоматического заполнения матриц тестовыми данными и проверки результата однопоточным алгоритмом.

5) Разработайте многопоточное приложение для подсчета количества совпадающих элементов для двух матриц MxN (0<M<10000, 0<N<10000) из вещественных чисел в K потоков, выполняющихся параллельно (0<K<1000). Совпадающими следует считать те элементы, которые находятся на соответствующих позициях и равны по значению. Значение K вводится пользователем.

В коде программы предусмотрите возможность автоматического заполнения матриц тестовыми данным и проверки результата однопоточным алгоритмом.

6) Разработайте многопоточное приложение для умножения двух целочисленных матриц из NxN элементов (0<N<10000) в N потоков, выполняющихся параллельно. Значение N вводится пользователем.

В коде программы предусмотрите возможность автоматического заполнения матриц тестовыми данными и проверки результата однопоточным алгоритмом.

7) Разработайте многопоточное приложение для печати минимальных элементов по строкам в двумерном массиве из MxN элементов (0<M<10000, 0<N<10000) в M потоков, выполняющихся параллельно. Значения M и N задаются пользователем.

В коде программы предусмотрите возможность автоматического заполнения массива тестовыми данными и проверки результата последовательным поиском.

8) Разработайте многопоточное приложение для вычисления среднего значения элементов двумерного массива вещественных чисел из MxN элементов (0<M<10000, 0<N<10000) в K потоков, выполняющихся параллельно. Значения M, N и K задаются пользователем.

В коде программы предусмотрите возможность автоматической генерации тестовых данных и проверки результата последовательным вычислением.

9) Разработайте многопоточное приложение для подсчета частоты появления символов в заданной текстовой строке в K потоков, где K – количество искомых символов. Входными данными для программы являются текстовая строка из N символов (0<N<100000) и набор символов для подсчета. Результатом работы программы должна быть таблица частот появления этих символов в тексте.

В коде программы предусмотрите возможность автоматической генерации тестовой строки и проверки результата последовательным подсчетом.

10) Разработайте многопоточное приложение для вычисления разности двух квадратных матриц MxN (0<M<10000, 0<N<10000) в K потоков, выполняющихся параллельно. Значение M, N и K вводятся пользователем.

В коде программы предусмотрите возможность автоматического заполнения матриц тестовыми данными и проверки результата однопоточным вычитанием.

11) Разработайте многопоточное приложение для параллельного поиска локальных экстремумов в заданном двумерном массиве из MxN (0<M<10000, 0<N<10000) вещественных чисел в K потоков. M, N и K вводятся пользователем. Результатом работы программы должен быть набор локальных экстремумов с указанием их расположения в массиве. Локальным экстремумом будем считать число, которое больше или меньше всех соседних с ним элементов массива. Для простоты граничными элементами массива (0 и M-1 строки, а также 0 и N-1 столбцы) можно пренебречь.

В коде программы предусмотрите возможность автоматического заполнения массива тестовыми данными и проверки результата однопоточным алгоритмом.

12) Разработайте многопоточное приложение для подсчета количества вхождений заданного элемента в двумерный целочисленный массив из MxN элементов (0<M<10000, 0<N<10000) в K потоков. Параметры M, N и K вводятся пользователем. В коде программы предусмотрите возможность автоматического заполнения массива тестовыми данными и проверки результата однопоточным алгоритмом.

Лабораторная работа. Потоки в Windows

Задание 1. Создайте на основе приведенных выше материалов в Microsoft Visual Studio консольное приложение с 2-3 потоками исполнения. Соберите проект и проверьте с помощью Диспетчера задач Windows, что потоки успешно запущены.

```
#include <windows.h>
#include <iostream>

using namespace std;

DWORD WINAPI WorkerThread(LPVOID param)
{
	while(true)
	{
		cout << *((int*)param) << " ";
	}
}

int main()
{
	HANDLE hThread;
	DWORD ThreadID;
	DWORD Num = 2;
	hThread = CreateThread(NULL, 0, WorkerThread,
(void*)&Num, 0, &ThreadID);
	if(hThread == NULL)
	{
		cout << "ошибка " << GetLastError() << endl;
		system("pause");
		return -1;
	}
	while(true)
	{
		cout << 1 << " ";
	}
	return 0;
}
```

Задание 2. Разработайте приложение, которое будет запускать 2-3 дополнительных потока исполнения и затем по команде пользователя приостанавливать и возобновлять их работу. Отследите с помощью Диспетчера задач Windows загрузку центрального процессора.

Задание 3. Разработайте приложение, решающее задачу из задания 2 работы «Потоки в Linux».

§ 4. СИНХРОНИЗАЦИЯ ПРОЦЕССОВ И ПОТОКОВ

В ходе исполнения процессы могут взаимодействовать друг с другом. Такое взаимодействие можно разделить на два вида:

- Синхронизация;
- Обмен данными между процессами.

С точки зрения синхронизации нас будут интересовать параллельные взаимодействующие процессы – процессы, которые одновременно находятся в каком-либо активном состоянии. Такие процессы можно разделить на два вида:

• **Независимые** (independent processes) – множества переменных которых (файлы, области оперативной памяти) не пересекаются;

• **Взаимодействующие** (cooperating processes) – совместно используют некоторые (общие) переменные.

Ресурсы таких процессов можно разделить на допускающие одновременное использование и на не допускающие одновременного использования – критические. В случае критических ресурсов может проявиться несколько проблем. Одна из наиболее важных – состояние состязания или состояние гонки (race condition).

Состояние состязания – ситуация, в которой два (и более) процесса (или потока) считывают или записывают данные одновременно и конечный результат зависит от того, в каком порядке процессы получают доступ к данным.

Критической областью или **критической секцией (critical section)** называется часть программы, в которой есть обращение к совместно используемому неразделяемому ресурсу.

Основным способом решения проблемы состязания является запрет одновременного обращения к критическому ресурсу более чем одним процессом (**взаимное исключение, mutual exclusion**), т.е. запрет на одновременное нахождение двух или более процессов в критической секции (рисунок 6).

Для написания корректного кода были сформулированы условия правильного использования общих данных:

1. Два процесса не должны одновременно находиться в критических областях.

2. Не должно быть предположений об относительных скоростях или количестве процессов.

3. Процессы, находящиеся вне критической области, не должны блокировать другие процессы.

4. Невозможна ситуация, в которой процесс вечно ждет попадания в критическую область.

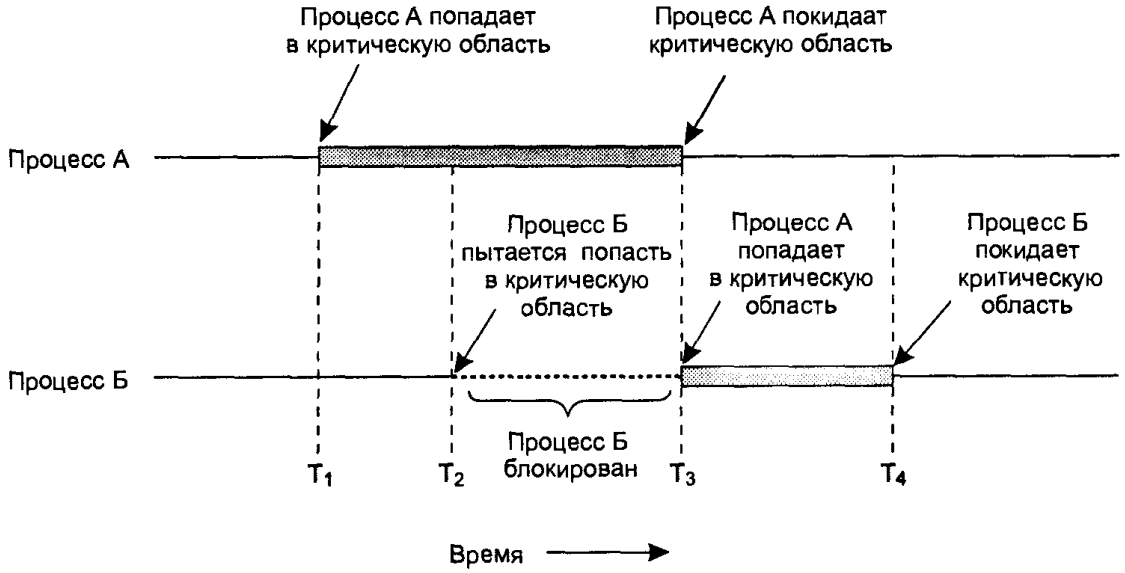

Рисунок 6 – диаграмма реализации взаимного исключения

Рассмотрим некоторые из способов реализации взаимного исключения. К первой группе отнесем алгоритмы активного ожидания.

Постоянная проверка значения переменной в ожидании некоторого значения называется **активным ожиданием**, а алгоритмы, её использующие – **алгоритмами активного ожидания (busy waiting algorithms).**

Блокировка, использующая активное ожидание, называется **спин-блокировкой (spinlock)**.

Классическими алгоритмами активного ожидания являются:

а) **Алгоритм Деккера** – первое известное корректное программное решение проблемы взаимного исключения.

б) **Алгоритм Петерсона** (1981 год) – более простой алгоритм программного взаимного исключения.

в) **Алгоритм пекарни Лампорта** (также упоминается как алгоритм Лампорта) – решение для случая n-процессов.

Приведем алгоритм Петерсона:

```
#define FALSE 0
#define TRUE  1
int turn;                          /* Чья сейчас очередь? */
int interested[2];                 /* Все переменные изначально
*/                                 /*  равны 0 (FALSE) */
void enter_region(int process) /* Процесс 0 или 1 */
{  int other;           /* Номер второго процесса */
   other = 1 - process; /* Противоположный процесс */
```

```
    interested[process] = TRUE; /* Индикатор интереса */
    turn = process;                    /* Установка флага */
    while (turn == process && interested[other] == TRUE);
  }
  void leave_region(int process) /* process-процесс, покидающий
  кр.с.*/
  {
    interested[process] = FALSE; /* индикатор выхода из кр. с.
  */
  }
```

Для применения этого алгоритма двум процессам (или потокам) условно назначаются номера «0» и «1», после чего критические секции размечаются операторными скобками enter_region() и leave_region() с соответствующими номерами. Исследование работы алгоритма предлагаем выполнить читателям самостоятельно.

Вторую группу способов реализации взаимного исключения составляют примитивы синхронизации, использование которых предполагает, что ожидающий процесс или поток находятся в неактивном состоянии.

Одними из наиболее известных примитивов синхронизации являются семафоры. Семафоры представляют собой новый специальный тип целочисленных переменных, предложенный Дейкстрой в 1965 году. Значение семафора может быть 0 или некоторым положительным числом. Над семафорами определены две операции:

- закрытие (down, wait, P(S));
- открытие (up, post, signal, V(S)).

Операция **down (P(S))** – сравнивает значение семафора с 0, и если семафор больше 0, уменьшает его на 1 и возвращает управление, если = 0, то переводит вызывающий процесс в состояние пассивного ожидания.

Операция **up (V(S))** – увеличивает значение семафора на 1 и переводит один или несколько ожидающих процессов, которые не могли завершить операцию down в состояние готовности.

Операции down и up – являются атомарными (т.е. все действия выполняются как неделимые).

Семафоры по способу реализации можно классифицировать на

- Двоичные семафоры – могут принимать значения только 0 или 1;
- N-ичные или счетные семафоры – могут иметь значения от 0 до N.

Кроме операции down и up зачастую рассматривается дополнительная операция – инициализация семафора InitSemaphore(имя, значение).

Для выполнения синхронизации критических секции семафоры могут быть использованы согласно следующему псевдокоду:

```
semaphore S;
InitSemaphore(S, 1);
```

```
// 1-ый процесс
while (true) {
     P(S);
     // критическая секция
     V(S);
}
```

```
// 2-ой процесс
while (true) {
     P(S);
     // критическая секция
     V(S);
}
```

Другим популярным примитивом синхронизации являются мьютексы (mutex – mutual exclusion semaphore – семафор взаимного исключения). Представляют собой некоторый аналог специальной двоичной версии семафора. Позволяют только управлять доступом к совместно используемым ресурсам или коду.

Мьютекс может находится в одном из двух состоянии:

- неблокированном (открыт, свободен, 1)
- блокированном (закрыт, захвачен, 0).

Поддерживает операции:

- *mutex_lock* – закрыть, захватить мьютекс;
- *mutex_unlock* – открыть, освободить мьютекс;
- *mutex_trylock* – неблокирующая попытка закрыть (захватить) мьютекс, возвращающая управление сразу же.

К дополнительным свойствам мьютекса можно отнести:

- Запоминание владельца (освободить мьютекс может только захвативший его);
- Рекурсивность (возможность многократного захвата мьютекса).

В качестве примера использования мьютексов рассмотрим одни из классических задач взаимодействия процессов – задачу производители-потребители (англ. producers-consumers). Эта задача также известна как задача ограниченного буфера:

- Дан буфер заранее фиксированного размера
- Имеется один (или более) процесс-производитель, который помещает данные в буфер (буферный пул). Если буфер полностью заполняется – производители «засыпают»
- Имеется один (или более) процесс-потребитель, который считывает данные из буфера. При опустошении буфера потребители блокируются.

Приведем псевдокод, демонстрирующий решение этой задачи:

```
semaphore S_free, S_full, S_excl ;
InitSemaphore(S_free, N);
InitSemaphore(S_full, 0);
InitSemaphore(S_excl, 1);
// производитель
while (true) {
// приготовить сообщение
     P(S_free);
     P(S_excl);
     // послать сообщение
     Put(message);
     V(S_full);
     V(S_excl);
}
```

```
// потребитель                          V(S_free);
while (true) {                          V(S_excl);
     P(S_full);                    // обработать сообщение
     P(S_excl);                    }
     // получить сообщение
```

Здесь S_free – счетный семафор, S_full, S_excl – мьютексы.

В Windows решение задачи синхронизации представлено несколькими способами. Один из них использование специальных объектов ядра – объеков синхронизации. **Объектами синхронизации** называются объекты ядра, которые могут находится в одном из двух состояний: сигнальном (signaled) и несигнальном (nonsignaled).

Объекты синхронизации могут быть разделены на 4 категории:

1) Объекты синхронизации для параллельных потоков:

- Мьютекс;
- Событие;
- Семафор.

2) Объект синхронизации, переходящий в сигнальное состояние по истечении заданного интервала времени:

- Ожидающий таймер.

3) Объекты, переходящие в сигнальное состояние по завершению:

- Задание (job);
- Процесс (process);
- Поток (thread).

4) Объекты, переходящие в сигнальное состояние после получения сообщения об изменении содержимого объекта:

- Изменение состояния каталога (change notification);
- Консольный ввод.

Работа с объектами синхронизации реализуется функциями ожидания перехода объекта ядра в сигнальное состояние:

```
DWORD WaitForSingleObject (
     HANDLE hObject,
     DWORD dwMilliseconds);
DWORD WaitForMultipleObjects (
     DWORD nCount,
     CONST HANDLE *lpHandles,
     BOOL fWaitAll,
     DWORD dwMilliseconds);
```

Первая из функции рассчитана на работу с одним объектом синхронизации, вторая – с группой до 64 объектов. Соответственно, в качестве аргумента первая из приведенных функции принимает дескриптор объекта синхронизации `hObject`, а вторая функция принимает целочисленный параметр `nCount`, задающий количество дескрипторов объектов в массиве `lpHandles` и адрес массива дескрипторов `lpHandles`. Также обе функции принимают

параметр `dwMilliseconds` задающий интервал ожидания перехода объекта синхронизации в сигнальное состояние. Дополнительно, вторая функция принимает параметр флаг `fWaitAll`, задающий ожидание перехода в сигнальное состояние всех дескрипторов из массива или хотя бы одного.

Возвращаемое значение указывает причину завершения ожидания и может быть равно:

- `WAIT_OBJECT_0` – указанный объект перешел в сигнальное состояние;
- `WAIT_TIMEOUT` – в течение отведенного времени объект не перешел в сигнальное состояние;

`WAIT_FAILED` – неудачное завершение функции.

Мьютексы в Windows. Работа с мьютексами в Windows осуществляется на основе концепции владения: мьютекс рассматривается как объект, который может принадлежать одновременно только одному потоку или не принадлежать никому. Захват мьютекса в собственность потока может быть выполнен двумя способами:

- В момент создания мьютекса при установке флага bInitialOwner в TRUE;
- Как побочное действие функций WaitForSingleObject или WaitForMultipleObjects.

За создание мьютекса отвечает функция CreateMutex:

HANDLE **CreateMutex**(LPSECURITY_ATTRIBUTES *lpMutexAttributes*, BOOL *bInitialOwner*, LPCTSTR *lpName*);

Здесь lpMutexAttributes задает адрес структуры атрибутов безопасности для мьютекса (при использовании для синхронизации потоков одного процесса как правило задается как NULL);

bInitialOwner задает флаг начального владения мьютексом: в случае TRUE – поток-создатель становится его владельцем, иначе мьютекс остаётся не захваченным;

lpName задает символьное имя для мьютекса, которое используется для доступа к мьютексу из другого процесса при помощи функции OpenMutex:

HANDLE **OpenMutex**(DWORD *dwDesiredAccess*, BOOL *bInheritHandle*, LPCTSTR *lpName*);

Действия по освобождению и захвату мьютекса выполняют функции ReleaseMutex (освобождение) и WaitForSingleObject (захват). По действию, функция WaitForSingleObject в случае если мьютекс находится в сигнальном состоянии (т.е. свободен) захватывает его и возвращает управление. Если мьютекс находится в несигнальном состоянии, то функция останавливает выполнение текущего потока до тех пор, пока мьютекс не освободиться и не перейдет в сигнальное состояние.

BOOL **ReleaseMutex**(HANDLE *hMutex*);

Пример:

```
// объявляем мьютекс
HANDLE hMutex;
// создаем свободный, «ничей» мьютекс
hMutex = CreateMutex( NULL, FALSE, "MutexForGuardResource");
// обрабатываем ошибки
if (hMutex = = NULL)
{ … }
// ожидаем захвата мьютекса
if (WaitForSingleObject(hMutex, INFINITE) = = WAIT_OBJECT_0)
{ // мьютекс захвачен – реализуем код
  // критической секции
   …
}
// «освобождаем» мьютекс
ReleaseMutex(hMutex);
```

Мьютексы в Linux. Все функции и типы данных, относящиеся к мьютексам библиотеки потоков POSIX, определены в заголовочном файле `pthread.h`. Также как и в Windows, мьютекс реализует интерфейс взаимного исключения, то есть, представляет собой блокировку, которая устанавливается перед доступом к разделяемому ресурсу и снимается после выполнения необходимых действий. Мьютексы описываются типом `pthread_mutex_t`. Перед использованием мьютекс должен быть инициализирован вызовом функции `pthread_mutex_init()`:

```
int pthread_mutex_init(pthread_mutex_t* mutex, pthread_mutexattr_t* attr);
```

Первый аргумент – указатель на переменную `pthread_mutex_t` (идентификатор мьютекса), второй – указатель на переменную типа `pthread_mutexattr_t` (дополнительные атрибуты мьютекса, по умолчанию = `NULL`).

Для получения исключительного доступа к ресурсу поток «запирает» мьютекс вызывая функцию `pthread_mutex_lock()` (захватывает мьютекс). Единственный параметр функции – идентификатор мьютекса.

Закончив работу с ресурсом поток «открывает» мьютекс, вызывая `pthread_mutex_unlock()`, в который передается идентификатор освобождаемого мьютекса.

Если поток вызывает `pthread_mutex_lock()` для мьютекса захваченного другим потоком, эта функция не вернет управление, пока другой поток не освободит мьютекс с помощью вызова `pthread_mutex_unlock()`.

Если поток не должен блокироваться при попытке захватить мьютекс он может использовать функцию `pthread_mutex_trylock()`, которая захватывает свободный мьютекс или возвращает код ошибки (EBUSY), если мьютекс уже захвачен другим потоком.

По окончании работы с мьютексом он удаляется функцией `pthread_mutex_destroy()`.

Все перечисленные функции, кроме `pthread_mutex_init` принимают ровно один аргумент – указатель на переменную типа `pthread_mutext_t`.

Семафоры В Windows

Создание или открытие семафора:

```
HANDLE CreateSemaphore(
  LPSECURITY_ATTRIBUTES lpSemaphoreAttributes,
  LONG lInitialCount,
  LONG lMaximumCount,
  LPCTSTR lpName
);
```

Здесь:

`lpSemaphoreAttributes` – указатель на структуру, определяющую параметры безопасности и наследования получаемого дескриптора;

`lInitialCount` – начальное значение семафора (`0≤lInitialCount≤lMaximumCount`);

`lMaximumCount` – максимальное значение семафора;

`lpName` – имя семафора (опционально, используется при разделении объекта между несколькими процессами). Если значение задано как `NULL` – семафор создается без имени и может быть использован при помощи своего дескриптора.

В случае успешного завершения функция возвращает дескриптор семафора, в противном случае – `NULL`.

Системный вызов `OpenSemaphore()` может быть использован для открытия семафора в другом процессе по имени.

Значение семафора уменьшается на 1 при его использовании в функции ожидания (соответствует операции DOWN над семафором).

Увеличить значение семафора можно посредством вызова функции `ReleaseSemaphore()` (соответствует функции UP над семафором), которая имеет следующий прототип:

```
BOOL ReleaseSemaphore(
  HANDLE hSemaphore,
  LONG lReleaseCount,
  LPLONG lpPreviousCount
);
```

Здесь:

`hSemaphore` – дескриптор семафора;

`lReleaseCount` – положительное число на которое увеличивается значение семафора;

- `lpPreviousCount` – указатель на переменную, в которую помещается предыдущее значение семафора. Может быть равно NULL.

Семафоры в Linux. Для работы с семафорами существует два набора системных вызовов. Самый старый из них называется System V IPC (читается «систем пять»), более современный – POSIX IPC. Все объявления функций и типов, относящихся к семафорам, можно найти в файле /usr/include/nptl/semaphore.h. (заголовочный файл <semaphore.h> следует подключить к проекту для использования семафоров). Семафоры POSIX описываются типом `sem_t`. Перед использованием неименованный семафор должен быть проинициализирован функцией `sem_init()`. Первый параметр `sem_init()` – указатель на семафор, второй параметр – pshared – не будет использоваться (задайте=0), в третьем параметре передается значение, которым инициализируется семафор.

Дальнейшая работа с семафором осуществляется с помощью функции `sem_wait()` и `sem_post()`. Единственным аргументом `sem_wait()` и `sem_post()` служит указатель на семафор. Вызов `sem_post()` увеличивает значение семафора на единицу, а `sem_wait()` – приостанавливает выполнение вызвавшего ее процесса (потока) до тех пор, пока значение семафора не станет большим нуля, после чего функция уменьшает значение семафора на единицу и возвращает управление, разблокировав процесс или поток.

После завершения работы с неименованным семафором следует вызвать `sem_destroy()` для удаления семафора.

Прототипы функции для работы с неименованными семафорами POSIX:

```
#include <semaphore.h>
int sem_init(sem_t *sem, int pshared, unsigned value);
int sem_wait(sem_t *sem);
int sem_post(sem_t *sem);
int sem_trywait(sem_t *sem);
int sem_destroy(sem_t *sem);
```

Лабораторная работа. Синхронизация в Linux

Задание 1. Разминка: Дана консольная программа, дважды печатающая на экране упорядоченную последовательность из глобального массива:

```
#include <pthread.h>
#include <iostream>
using namespace std;
#define N 25
int a[N];
// функция потока
void* ThreadFunc(void*)
{
  int pause;
  for(int i=0;i<N;i++)
  {
    cout << " " << a[i] << " ";
// замедляем вывод для наглядности
    for(long j=0;j<1000000;j++) pause++;
  }
  cout << endl;
```

```
  return 0;
}
int main(void)
{
  pthread_t Thread;
  int pause;
  int res;
// заполняем глобальный массив
  for(int i=0;i<N;i++) a[i]=i;
  res = pthread_create(&Thread, NULL, ThreadFunc, NULL);
  if(res != 0)
  {
    cerr << "Ошибка при создании потока" << endl;
     cin.get();
     return 0;
  }
  for(int i=0;i<N;i++)
   {
    cout << " " << a[i] << " ";
    for(long j=0;j<1000000;j++) pause++;
   }
  cout << endl;
// дожидаемся завершения работы потока
  pthread_join(Thread, NULL);

  return 0;
}
```

Выполните заданную выше программу. Объясните полученный результат. Определите в коде критический ресурс и критическую секцию.

Задание 2. Синхронизируйте потоки в программе из задания 1 используя мьютексы POSIX.

Задание 3. Синхронизируйте потоки в программе из задания 1 используя семафоры POSIX.

Задание 4. По вариантам:

Вариант 1) Задача о Винни Пухе и правильных пчелах.

В одном лесу живут N пчел и один медведь, которые используют один горшок меда, вместимостью M глотков. Сначала горшок пустой. Пока горшок не наполнится, медведь спит. Как только горшок заполняется, медведь просыпается и съедает весь мед, после чего снова засыпает. Каждая пчела многократно собирает по одному глотку меда и кладет его в горшок. Пчела, которая приносит последнюю порцию меда, будит медведя.

Создайте многопоточное приложение, моделирующее поведение пчел и медведя.

Вариант 2) Задача о каннибалах.

Племя из N дикарей ест вместе из большого горшка, который вмещает m кусков тушеного миссионера. Когда дикарь хочет обедать, он ест из горшка один кусок, если только горшок не пуст, иначе дикарь будит повара и ждет, пока тот не наполнит горшок. Повар, сварив обед, засыпает.

Создайте многопоточное приложение, моделирующее обед дикарей. При решении задачи используйте семафоры.

СПИСОК ЛИТЕРАТУРЫ

1. Лав, Р. Linux. Системное программирование / Р. Лав. – СПб.: Питер, 2008. – 416 с.

2. Побегайло, А.П. Системное программирование в Windows / А.П. Побегайло. – СПб.: БХВ Петербург, 2006. – 1056 с.

3. Рихтер, Д. Windows via C/C++. Программирование на языке Visual C++ [Электронный ресурс]: пер. с англ. – Электрон. текстовые дан. – СПб. [и др.]: Питер, 2009. – 878 с. – (Мастер-класс). – Режим доступа: lib.vsu.by.

4. Стивенс, Р. UNIX. Профессиональное программирование / Р. Стивенс, С. Раго. – СПб.: Символ-Плюс, 2010. – 1040 с.

5. Таненбаум, Э. Современные операционные системы / Э. Таненбаум. – СПб.: Питер, 2014. – 1120 с.

6. Чан, Т. Системное программирование на C++ для Unix / Т. Чан; пер. с англ. – К.: Издательский отдел BHV, 1997.

7. Programming reference for the Win32 API [Электронный ресурс]. – Режим доступа: https://learn.microsoft.com/en-us/windows/win32/api/.

Учебное издание

ОПЕРАЦИОННЫЕ СИСТЕМЫ. ПРОЦЕССЫ И ПОТОКИ

Методические рекомендации

Составитель
НОВЫЙ Вадим Владимирович

Технический редактор *Г.В. Разбоева*
Компьютерный дизайн *В.Л. Пугач*

Подписано в печать .2022. Формат 60x84 $^{1}/_{16}$. Бумага офсетная.
Усл. печ. л. 2,73. Уч.-изд. л. 2,21. Тираж экз. Заказ .

Издатель и полиграфическое исполнение – учреждение образования
«Витебский государственный университет имени П.М. Машерова».

Свидетельство о государственной регистрации в качестве издателя,
изготовителя, распространителя печатных изданий
№ 1/255 от 31.03.2014.

Отпечатано на ризографе учреждения образования
«Витебский государственный университет имени П.М. Машерова».
210038, г. Витебск, Московский проспект, 33.