**Основные компоненты Операционной Системы**

*Операционная система* – весьма сложная по архитектуре программная система, в которой можно выделить следующие основные компоненты:

1. Управление процессами.
2. Управление основной памятью.
3. Управление файлами.
4. Управление системой ввода-вывода.
5. Управление внешней памятью.
6. Поддержка сетей (networking).
7. Система защиты (protection).
8. Система поддержки командного интерпретатора.
9. Графическая оболочка.

Рассмотрим эти компоненты подробнее.

**Управление процессами**. Процесс – это *программа* пользователя в ходе ее выполнения в компьютерной системе. ОС управляет работой процессов, их распределением по процессорам и ядрам системы, порядком их выполнения и размещения в памяти, их синхронизацией при параллельном решении частей одной и той же задачи разными процессами.

**Управление основной памятью**. Основная (оперативная) *память* может рассматриваться как большой *массив*. *Операционная система* распределяет ресурсы памяти между процессами, выделяет *память* по запросу, освобождает ее при явном запросе или по окончании процесса, хранит списки занятой и свободной памяти в системе.

**Управление файлами**. *Файл* – это логическая *единица* размещения информации на внешнем устройстве, например, на диске. ОС организует работу пользовательских программ с файлами, создает файлы, выполняет их открытие и закрытие и *операции* над ними (чтение и *запись*), хранит ссылки на файлы в директориях (папках) и обеспечивает их *поиск* по символьным именам.

**Управление системой ввода-вывода**. Как уже отмечалось, в компьютерной системе имеется большое число *внешних устройств* (принтеры, сканеры, *устройства управления* компакт-дисками и др.), управляемых специальными контроллерами (спецпроцессорами) и драйверами – низкоуровневыми программами управления устройствами, выполняемыми в привилегированном режиме. ОС управляет всеми этими аппаратными и *программными компонентами*, обеспечивая *надежность* работы *внешних устройств*, эффективность их использования, диагностику и *реконфигурацию* в случае их сбоев и отказов. Для этого ОС хранит и использует *таблицу состояния*

**Управление внешней памятью**. Как уже говорилось, внешняя (вторичная) *память* – это расширение оперативной памяти процессора более медленными, но более емкими и постоянно хранящими информацию видами памяти (диски, ленты и др.). При управлении внешней памятью ОС решает задачи, аналогичные задачам управления основной памятью, - выделение памяти по запросу, *освобождение памяти*, хранение списков свободной и занятой памяти и др. ОС поддерживает также использование ассоциативной памяти (кэш-памяти) для оптимизации обращения ко внешней памяти.

**Поддержка сетей**. Как неоднократно подчеркивалось, любая современная компьютерная система постоянно или временно находится в различных локальных и глобальных сетях. *Операционная система* обеспечивает использование сетевого оборудования (*сетевых карт*, или адаптеров), вызов соответствующих драйверов, поддержку удаленного взаимодействия с файловыми системами, находящимися на компьютерах сети, удаленный вход на другие компьютеры сети и использование их вычислительных ресурсов, отправку и получение сообщений по сети, защиту от *сетевых атак*.

**Система защиты**. Согласно современным принципам надежных и безопасных, при работе ОС должны быть обеспечены *надежность* и *безопасность*, т.е. защита от внешних атак, *конфиденциальность* личной и корпоративной информации, *диагностика* и исправления ошибок и неисправностей и др. ОС обеспечивает защиту *компонент* компьютерной системы, данных и программ, поддерживает фильтрацию *сетевых пакетов*, обнаружение и предотвращение внешних атак, хранит информацию обо всех действиях над системными структурами, полезную для анализа атак и борьбы с ними.

**Система поддержки командного интерпретатора**. Любая *операционная система* поддерживает *командный язык* (или набор *командных языков*), состоящих из пользовательских команд, выполняемых с пользовательского терминала (из пользовательской консоли). Типичные команды – это получение информации об окружении, установка и смена текущей рабочей директории, пересылка файлов, *компиляция* и выполнение программ, получение информации о *состоянии системы* и выполнении своих процессов и др. В системе *Windows* для выполнения команд по традиции используется окно пользовательской консоли MS *DOS* (MS *DOS* *Prompt*), в системе Linux – специальное окно "*Терминал*" (Start / *System* *Tools* / *Terminal*). Наиболее мощные *командные процессоры* имеются в *системах типа* *UNIX* (*UNIX*, Solaris, Linux и др.). Их командные языки позволяют писать **скрипты** – командные файлы, содержащие часто используемые последовательности команд ОС. В *UNIX* это наиболее удобно. Можно назвать такие командные языки *UNIX*, как **sh (Bourne Shell), csh (C shell), ksh (Korn shell), bash**.Каждый *UNIX*-программист имеет свой излюбленный *командный язык* и привыкает постоянно использовать скрипты и длинные нетривиальные последовательности команд, которые он выполняет с терминала. Что касается *Windows*, сравнительно недавно в ней появился мощный *командный интерпретатор* **PowerShell**,который и рекомендуется к использованию. Кроме того, для *Windows* имеется система **CygWin**,позволяющая выполнять команды и командные файлы *UNIX* в среде *Windows*. Типичная последовательность команд в стиле *UNIX*: *ps* –a | grep *saf*, которая выводит в *стандартный вывод* информацию об *активных процессах*, причем только принадлежащих пользователю **saf**. Вертикальная черта (p1 | p2) обозначает операцию **конвейер (pipe)**, позволяющую использовать *стандартный вывод* процесса p1 как *стандартный ввод* процесса p2, что и используется операцией grep (*фильтрация строк*, содержащих заданную последовательность). Подробнее о *UNIX* (Linux) можно прочитать в книге.

**Графическая оболочка** – подсистема ОС, реализующая графический пользовательский *интерфейс* пользователей и системных администраторов с операционной системой. Разумеется, использование одного лишь командного языка и системных вызовов неудобно, поэтому простой и наглядный графический пользовательский *интерфейс* с ОС необходим. Имеется много известных графических оболочек для операционных систем, причем их возможности очень похожи друг на друга - настолько, что подчас не вполне понятно, какая именно ОС используется. Среди графических оболочек, используемых в *системах типа* *UNIX*, можно назвать *CDE*, KDE, GNOME. ОС *Windows* и MacOS имеют собственные, весьма удобные графические оболочки.

**Управление процессами**

**Процесс (process) -** это пользовательская *программа* при ее исполнении в компьютерной системе. Для выполнения процесса требуется ряд **ресурсов**, включая время процессора, *память*, файлы, *устройства ввода-вывода*, сетевые устройства и др.

В классической схеме *UNIX*, при *создании процесса* для него создается новое *пространство* виртуальной памяти, т.е. *таблица страниц* для отображения *виртуальных адресов* в физические, своя для каждого нового процесса. При этом расходуются значительные ресурсы. Если учесть, что в *UNIX* каждая *команда* пользователя (например, ls – *вывод* содержимого *текущей директории*) запускается как **отдельный процесс**, то становится понятным, насколько "дорога" операция создания процесса в классическом смысле. Поэтому еще в 1980-х гг. появилась концепция **облегченного процесса** (lightweight process) – выполняемого в том же пространстве виртуальной памяти, что и процесс-родитель. При создании нового облегченного процесса ОС создает для него только **стек** – системный резидентный *массив* в памяти, предназначенный для поддержки выполнения процедур процесса и хранящий их локальные данные и связующую информацию между ними.

ОС отвечает за следующие действия, связанные с управлением процессами:

**Создание и удаление процессов**. При *создании процесса* необходимо создать в памяти соответствующие системные структуры (таблицу страниц, *стек* и др.). При удалении процесса *память*, занимаемая ими, освобождается, а также выполняется закрытие всех файлов и освобождение всех других ресурсов, которые использовал процесс, если последний не сделал этого явно.

**Приостановка и возобновление процессов**. Выполнение процесса приостанавливается при выполнении *синхронного ввода-вывода*, а также системного вызова или команды (типа **suspend** ). Сразу отметим, что использовать подобные *операции* явной *приостановки процессов* следует с осторожностью, так как приостанавливаемый процесс может находиться в своей **критической секции** – выполнять обработку общего ресурса, к которому каждому процессу предоставляется монопольный *доступ*, так что при его приостановке возникает ситуация **тупика (deadlock** ) – приостановленный процесс не может освободить *ресурс*, а конкурирующий процесс не может его получить. При приостановке процесса ОС сохраняет состояние его выполнения, а при возобновлении – восстанавливает.

**Синхронизация процессов**. Процессы работают параллельно и при этом конкурируют за общие ресурсы, а также должны в некоторые моменты вычислений ожидать наступления некоторых событий. Для предотвращения возможных конфликтов и несогласованностей, например, **race condition** - несогласованного доступа к общим данным, при котором один процесс читает старые данные, а другой их в этот же момент обновляет, - ОС предоставляет средства **синхронизации** (например, **семафоры** и **мониторы**, рассмотренные в следующем разделе).

**Взаимодействие процессов**. При своей параллельной работе процессам необходимо взаимодействие, с целью согласованного решения различных частей одной и той же задачи. Процессы могут взаимодействовать с помощью передачи **сообщений** друг другу, а также с помощью так называемых **условных переменных** и **рандеву** (все эти виды взаимодействия рассмотрены позже). ОС предоставляет все эти средства, в виде системных вызовов, для организации адекватного и удобного взаимодействия процессов.

**Семафоры**. В 1966 г. в работе ]проф. Эдсгер Дейкстра предложил новый способ *синхронизации процессов*, ставший классическим, - семафоры.

**Двоичный семафор (binary semaphore)** – *переменная* S, которая может находиться в двух состояниях: "открыт" и "закрыт"; над S определены две *операции* ( "семафорные скобки"): P(S) – закрыть, V(S) – открыть. При попытке закрыть уже закрытый семафор происходит *прерывание*, и ОС добавляет текущий процесс в *очередь* к закрытому семафору. Операция V(S) активизирует первый стоящий в очереди к S процесс, который успешно завершает операцию P(S). Если семафор S уже открыт, операция V(S) не имеет никакого эффекта.

Таким образом, если предположить, что аппаратура и ОС поддерживают подобную концепцию семафора, то она является удобным инструментом для синхронизации по ресурсам. Назовем **критической секцией** код, который может выполняться несколькими процессами параллельно и осуществляет *доступ* к некоторому общему для всех процессов ресурсу – глобальной области памяти, общему файлу и т.д. Обозначим код *критической секции* **critical\_section**.Если допустить, что данный код может выполняться параллельно в нескольких процессах напрямую, то может возникнуть уже известная нам ситуация **race condition (** конкуренция за общие данные): один процесс может изменять *ресурс*, а второй в этот момент считывать его (некорректное) состояние, либо два процесса одновременно будут пытаться изменять один и тот же *ресурс*, что приведет к нарушению его целостности. Таким образом, для критических секций необходимо решить задачу **взаимного исключения (mutual exclusion)** – в каждый момент времени не более чем один из *параллельных процессов* может выполнять *критическую секцию*. С помощью *семафоров Дейкстры* эта задача решается легко и изящно: код *критической секции* должен иметь вид

P(S); critical\_section; V(S);

В самом деле, предположим, что несколько процессов выполняют данный код. Первый из них, который начал выполнять операцию P(S), закрывает семафор S и получает *доступ* к *критической секции*. Все остальные процессы, которые пытаются выполнить операцию P(S) над закрытым семафором S, прерываются и попадают в *очередь* к закрытому семафору. Когда первый процесс закончил работу с ресурсом, он открывает семафор S операцией V(S) для первого процесса из очереди, который, выполнив P(S), вновь закрывает семафор, и т.д.

Очень важное свойство операций P и V в следующем: они **атомарны (atomic)** для других процессов, т.е. если процесс начал выполнять операцию P(S) или V(S), то никакой другой процесс до ее завершения не может также начать выполнять аналогичную операцию.

Подведем итог: для *синхронизации процессов* по общему ресурсу необходимы взаимное *исключение* выполнения критических секций и *атомарность операций* синхронизации.

Однако следует заметить, что использование семафоров – далеко не идеальный способ синхронизации, с точки зрения надежности. При их неаккуратном использовании возможна ситуация **тупика (взаимной блокировки, deadlock**), при которой образуется цепочка процессов, бесконечно ждущих друг друга. Простейший способ создать *deadlock* – использовать **два** семафора S1 и S2, так, что первый *параллельный процесс* пытается выполнить код **P(S1); P(S2)**,а второй – код **P(S2); P(S1)**.Очевидно, что при любом соотношении времен выполнения операций будут закрыты оба семафора, на которых и будут "висеть" оба процесса, не в состоянии двинуться дальше. Как же избежать подобных ситуаций? Ведь ни *компилятор*, ни *операционная система* не подскажут программисту правильный способ использования семафоров. Очень легко также "забыть" вызов V(S) и, тем самым, сделать общий *ресурс* "навеки" недоступным для других процессов. Один из способов решения этой задачи заключается в том, чтобы использовать специальные инструменты и технологии, автоматически обеспечивающие "правильную" последовательность применения операций над семафорами. Один из таких инструментов – **аспектно-ориентированное программирование**.

**Мониторы –** еще один, более надежный способ синхронизации, предложенный в 1974 г. одним из классиков компьютерных наук профессором Чарльзом Хоаром.

**Монитор** – многовходовый *модуль* M, в котором определены общие для процессов данные D (скрытые) и (абстрактные) *операции* P1, … PN над этими данными (в виде процедур).

В каждый момент не более чем один из *параллельных процессов* может вызвать какую-либо из операций: M.Pi (X, Y, …)

Вызов каждой *операции* монитора – атомарен (как и *операции* над семафором).

*Монитор* – еще один удобный механизм *синхронизации процессов* по ресурсам. Он более надежен, чем семафоры, поскольку вызов *операции* монитора автоматически обеспечивает разблокировку ресурса после завершения вызова.

Мониторы включены Ч. Хоаром в разработанный им язык *Concurrent* *Pascal* для *параллельного программирования* и разработки операционных систем.

Подробнее о семафорах и мониторах – в специальных разделах курса, посвященных управлению процессами и *синхронизации рооцессов.*