**Основные определения на защиту 2 и 3 лр**

**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.

**Процесс (process) -**пользовательская *программа* при ее исполнении в компьютерной системе.

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. Эдсгер Дейкстра предложил новый способ *синхронизации процессов*, ставший классическим, - семафоры.

**Двоичный семафор (binary semaphore)**– *системная переменная*, над которой определены *операции* открытия и закрытия, обеспечивающая в закрытом состоянии *прерывание* процесса, пытающегося ее закрыть, и добавление его к очереди к закрытому семафору; используется для *синхронизации процессов* по общим ресурсам.

**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.

**Монитор (как средство синхронизации)**– многовходовый *модуль* в котором определены общие для параллельных процесов данные и набор операций (в виде процедур) над ними, таких, что в каждый момент времени не более чем один из *параллельных процессов* может выполнять какую-либо операцию монитора.

**Race condition**- несогласованный *доступ* из *параллельных процессов* к общим данным.

**Атомарная (atomic) операция**– операция, такая, что, если один из *параллельных процессов* начал ее выполнять, никакой другой процесс до ее завершения не может также начать выполнять эту же операцию над теми же данными.

**Взаимное исключение (mutual exclusion)**– режим выполнения **критической секции**, в котором в каждый момент времени ее может выполнять не более чем один из *параллельных процессов*.

**Критическая секция**- код, который может выполняться несколькими процессами параллельно и осуществляет *доступ* к некоторому общему для всех процессов ресурсу – например, глобальной области памяти или общему файлу.

**Тупик (взаимная блокировка, deadlock** ) – ситуация, при которой образуется циклическая цепочка *блокированных процессов*, бесконечно ждущих друг друга.

**Мью́текс** ([англ.](https://ru.wikipedia.org/wiki/%D0%90%D0%BD%D0%B3%D0%BB%D0%B8%D0%B9%D1%81%D0%BA%D0%B8%D0%B9_%D1%8F%D0%B7%D1%8B%D0%BA) *mutex*, от *mutual exclusion* — «взаимное исключение») — примитив синхронизации, обеспечивающий взаимное исключение исполнения [критических участков](https://ru.wikipedia.org/wiki/%D0%9A%D1%80%D0%B8%D1%82%D0%B8%D1%87%D0%B5%D1%81%D0%BA%D0%B0%D1%8F_%D1%81%D0%B5%D0%BA%D1%86%D0%B8%D1%8F) кода[[1]](https://ru.wikipedia.org/wiki/%D0%9C%D1%8C%D1%8E%D1%82%D0%B5%D0%BA%D1%81#cite_note-_6a648ca94df68812-1). Классический мьютекс отличается от [двоичного семафора](https://ru.wikipedia.org/wiki/%D0%A1%D0%B5%D0%BC%D0%B0%D1%84%D0%BE%D1%80_(%D0%B8%D0%BD%D1%84%D0%BE%D1%80%D0%BC%D0%B0%D1%82%D0%B8%D0%BA%D0%B0)) наличием эксклюзивного владельца, который и должен его освобождать (т.е. переводить в незаблокированное состояние)[[2]](https://ru.wikipedia.org/wiki/%D0%9C%D1%8C%D1%8E%D1%82%D0%B5%D0%BA%D1%81#cite_note-:02-2). От [спинлока](https://ru.wikipedia.org/wiki/%D0%A1%D0%BF%D0%B8%D0%BD%D0%BB%D0%BE%D0%BA) мьютекс отличается передачей управления планировщику для переключения потоков при невозможности захвата мьютекса[[3]](https://ru.wikipedia.org/wiki/%D0%9C%D1%8C%D1%8E%D1%82%D0%B5%D0%BA%D1%81#cite_note-3). Встречаются также [блокировки чтения-записи](https://ru.wikipedia.org/wiki/%D0%91%D0%BB%D0%BE%D0%BA%D0%B8%D1%80%D0%BE%D0%B2%D0%BA%D0%B8_%D1%87%D1%82%D0%B5%D0%BD%D0%B8%D1%8F-%D0%B7%D0%B0%D0%BF%D0%B8%D1%81%D0%B8), именуемые разделяемыми мьютексами и предоставляющие помимо эксклюзивной блокировки общую, позволяющую совместно владеть мьютексом, если нет эксклюзивного владельца[[4]](https://ru.wikipedia.org/wiki/%D0%9C%D1%8C%D1%8E%D1%82%D0%B5%D0%BA%D1%81#cite_note-_0224f52bcb5f40d0-4).

Условно классический мьютекс можно представить в виде переменной, которая может находиться в двух состояниях: в заблокированном и в незаблокированном. При входе в свою критическую секцию поток вызывает функцию перевода мьютекса в заблокированное состояние, при этом поток блокируется до освобождения мьютекса, если другой поток уже владеет им. При выходе из критической секции поток вызывает функцию перевода мьютекса в незаблокированное состояние. В случае наличия нескольких заблокированных по мьютексу потоков во время разблокировки планировщик выбирает поток для возобновления выполнения (в зависимости от реализации это может быть, как случайный, так и детерминированный по некоторым критериям поток)[[5]](https://ru.wikipedia.org/wiki/%D0%9C%D1%8C%D1%8E%D1%82%D0%B5%D0%BA%D1%81#cite_note-_4dd9f51a54d39f79-5).

Задачей мьютекса является защита объекта от доступа к нему других потоков, отличных от того, который завладел мьютексом. В каждый конкретный момент только один поток может владеть объектом, защищённым мьютексом. Если другому потоку будет нужен доступ к данным, защищённым мьютексом, то этот поток [блокируется](https://ru.wikipedia.org/wiki/%D0%91%D0%BB%D0%BE%D0%BA%D0%B8%D1%80%D0%BE%D0%B2%D0%BA%D0%B0_(%D0%BF%D1%80%D0%BE%D0%B3%D1%80%D0%B0%D0%BC%D0%BC%D0%B8%D1%80%D0%BE%D0%B2%D0%B0%D0%BD%D0%B8%D0%B5)) до тех пор, пока мьютекс не будет освобождён. Мьютекс защищает данные от повреждения в результате асинхронных изменений ([состояние гонки](https://ru.wikipedia.org/wiki/%D0%A1%D0%BE%D1%81%D1%82%D0%BE%D1%8F%D0%BD%D0%B8%D0%B5_%D0%B3%D0%BE%D0%BD%D0%BA%D0%B8)), однако при неправильном использовании могут порождаться другие проблемы, например, [взаимная блокировка](https://ru.wikipedia.org/wiki/%D0%92%D0%B7%D0%B0%D0%B8%D0%BC%D0%BD%D0%B0%D1%8F_%D0%B1%D0%BB%D0%BE%D0%BA%D0%B8%D1%80%D0%BE%D0%B2%D0%BA%D0%B0) или двойной захват.

По типу реализации мьютекс может быть быстрым, [рекурсивным](https://en.wikipedia.org/wiki/Reentrant_mutex)  (англ.)[рус.](https://ru.wikipedia.org/w/index.php?title=%D0%A0%D0%B5%D0%BA%D1%83%D1%80%D1%81%D0%B8%D0%B2%D0%BD%D1%8B%D0%B9_%D0%BC%D1%8C%D1%8E%D1%82%D0%B5%D0%BA%D1%81&action=edit&redlink=1) или с контролем ошибок.

Еще более подробное изложнение определений мьюткса , монитора и семафора :

https://javarush.ru/groups/posts/2174-v-chem-raznica-mezhdu-mjhjuteksom-monitorom-i-semaforom