**Дисциплина:**

**«Операционные системы»**

Оглавление

[Введение 2](#_Toc136630436)

[1. Постановка задач 2](#_Toc136630437)

[2. Реализация первой задачи 2](#_Toc136630438)

[3. Реализация второй задачи 9](#_Toc136630439)

[4. Реализация третей задачи 14](#_Toc136630440)

[Заключение 19](#_Toc136630441)

[Список литературы 19](#_Toc136630442)

### Введение

*Виртуальная машина (ВМ) — компьютерная система, эмулирующая возможности каких-либо вычислительных комплексов гостевых платформ (guest) на аппаратно-программном обеспечении хост-платформы (host).*

*Упрощенно, ВМ — это программа, которая имитирует работу другой программы («компьютер внутри компьютера» или «машина внутри машины»). Поскольку имитируемая машина вне среды хост-платформы не существует, она получила название «виртуальной».*

*VirtualBox – программа виртуализации для операционных систем Windows, Linux и других.*

*Linux – это операционная система с открытым исходным кодом, который доступен для изменения любым пользователям.*

*Конкурентность — способность компьютера справляться с множеством разных задач одновременно.*

*Критическая секция — участок исполняемого кода программы, в котором производится доступ к общему ресурсу (данным или устройству), который не должен быть одновременно использован более чем одним*[*потоком выполнения*](https://ru.wikipedia.org/wiki/%D0%9F%D0%BE%D1%82%D0%BE%D0%BA_%D0%B2%D1%8B%D0%BF%D0%BE%D0%BB%D0%BD%D0%B5%D0%BD%D0%B8%D1%8F)*.*

## Постановка задач

1. Установка операционной системы на виртуальную машину.
2. Написание программы, которая будет имитировать обработку процессором потоков.
3. Решение задачи читатель-писатель с помощью многопоточности, а также защита всех уязвимых позиций программы.

## Реализация первой задачи

Установка виртуальной операционной системы состоит из 3 частей: установка virtualbox, создание и настройка виртуальной машины, установка линукса на виртуальную машину.

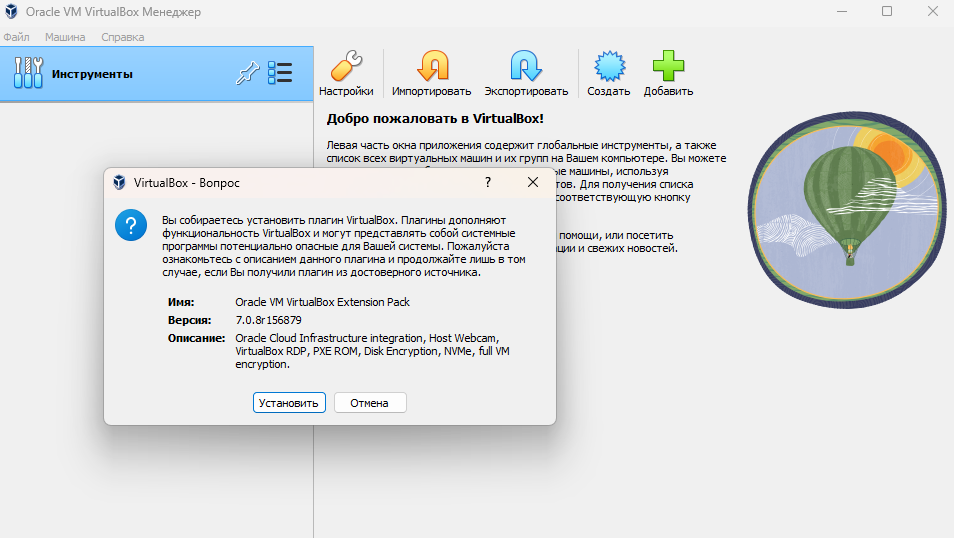
**Установка virtualbox.**



Установка данной программы проста. Нам необходимо скачать установочный файл для вашей операционной системы windows, запустить его и следовать простым шагам установщика.

Вдобавок можно установить Extension Pack, что разблокирует дополнительный функционал.

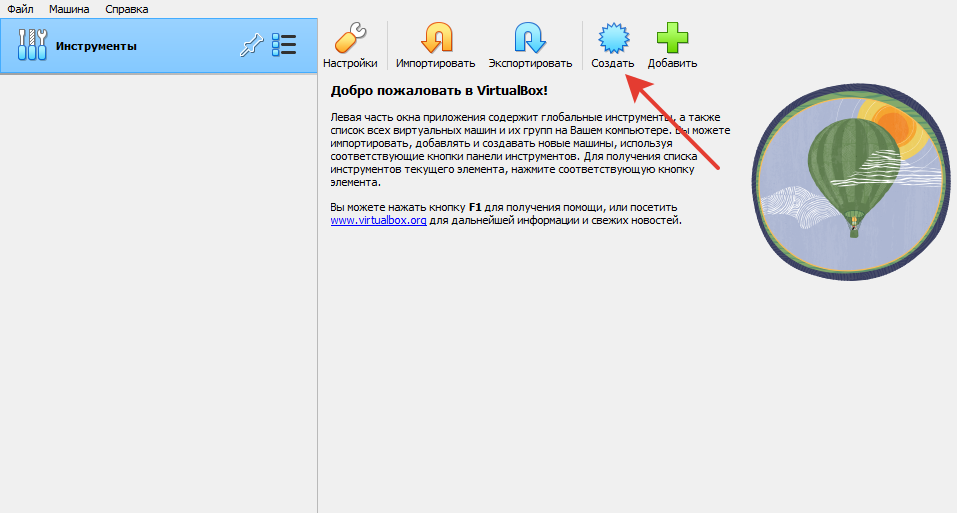
Установить Extension Pack еще проще. После установки *VirtualBox* запустите файл расширения и нажмите «Установить».



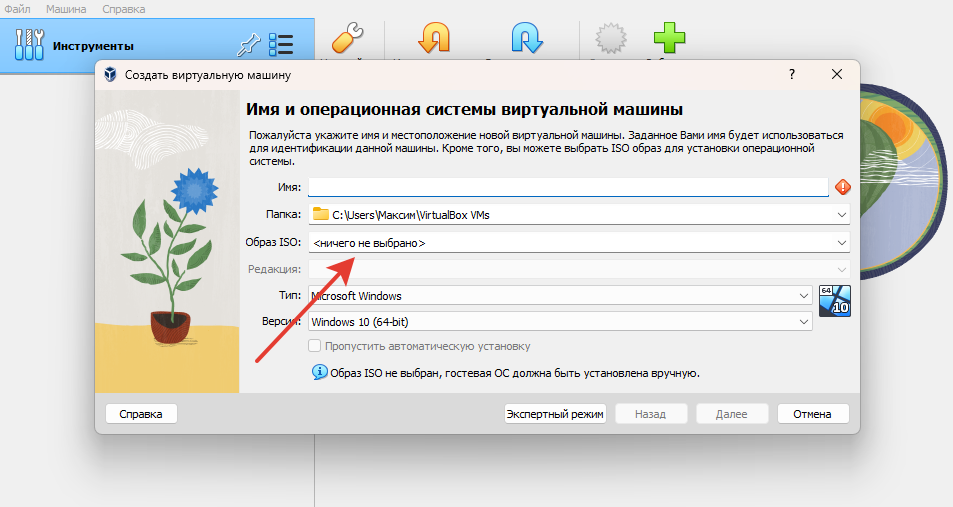
*Все настройки по умолчанию меня устраивают, настройка VirtualBox потребуется только после установки новой операционной системы.*

Создание виртуальной машины.

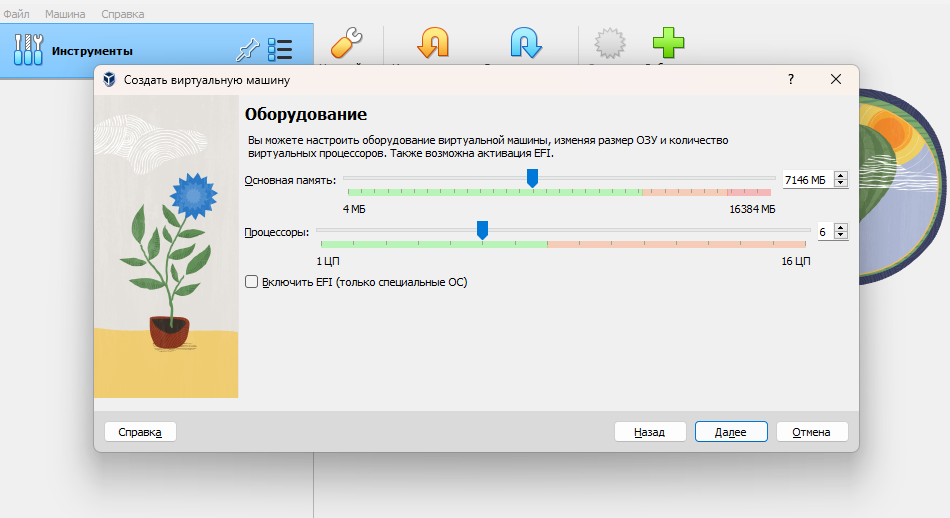
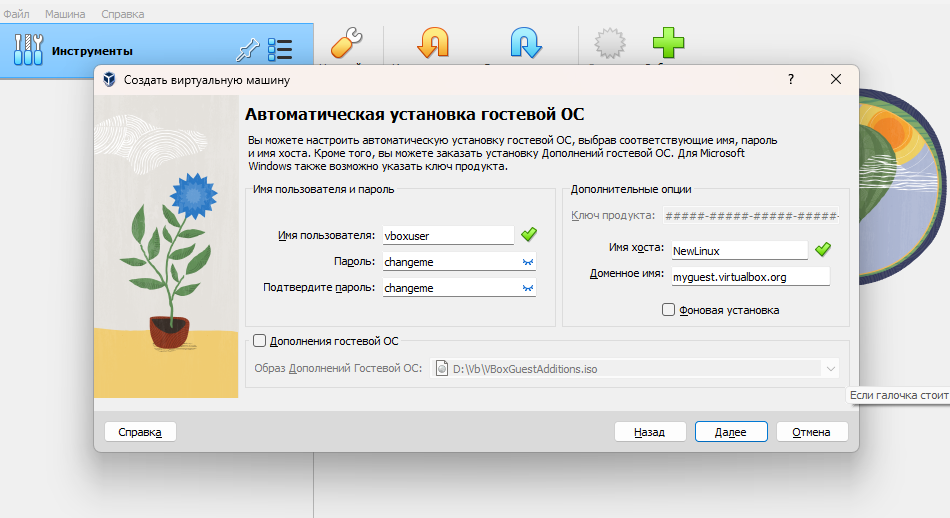
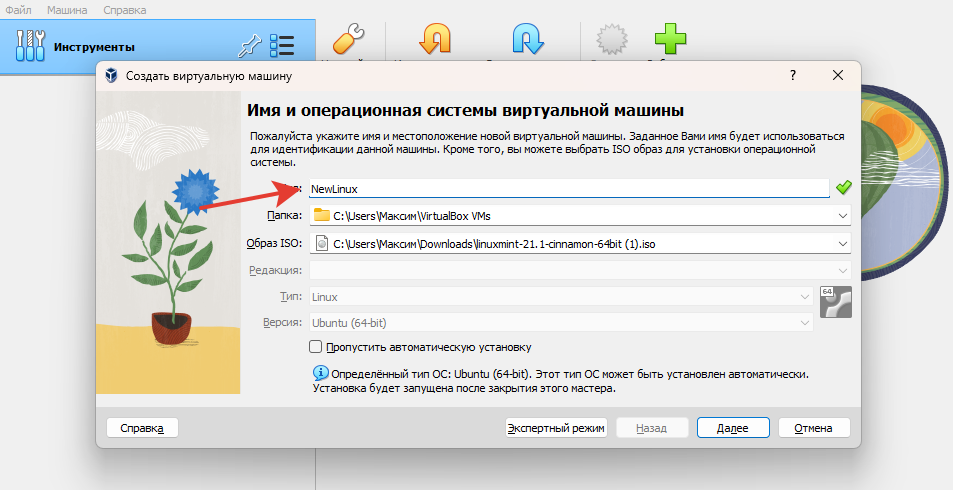
Чтобы установлю Linux Mint Cinnamon Edition, скачиваем iso образ с удобного зеркала.



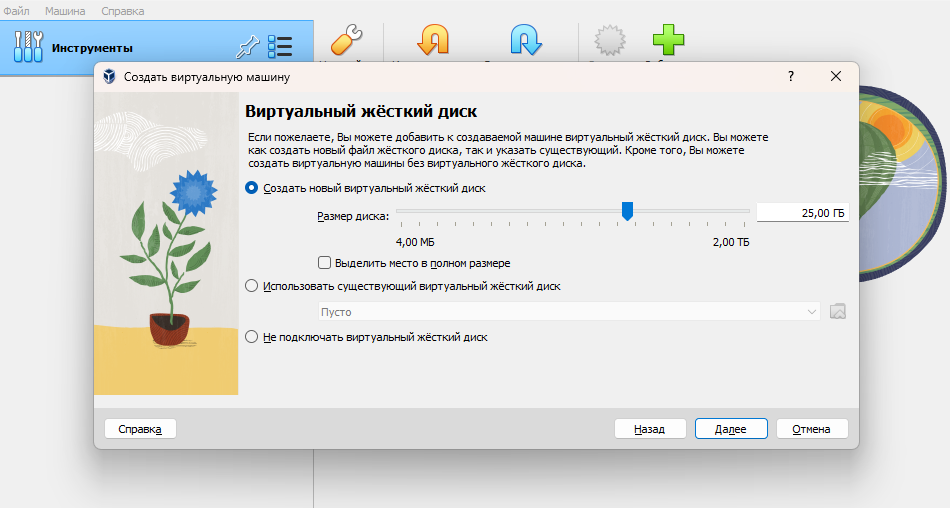
Пока что все очевидно, создаем новую виртуальную машину.



Первым делом стоит указать путь до iso образа. Программа автоматически соотнесет тип и версию. Вам останется указать лишь имя виртуальной машниы, которое будет использоваться в VirtualBox.



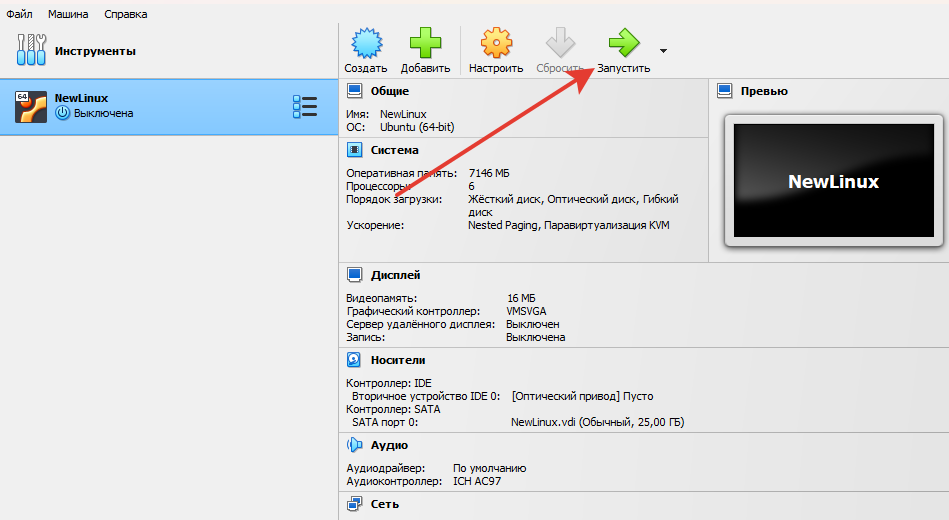
Память и количество ядер стоит выбирать, не выходя за красную линию. Таким образом вы исключите возможность зависаний вашей «настоящей» операционной системы.

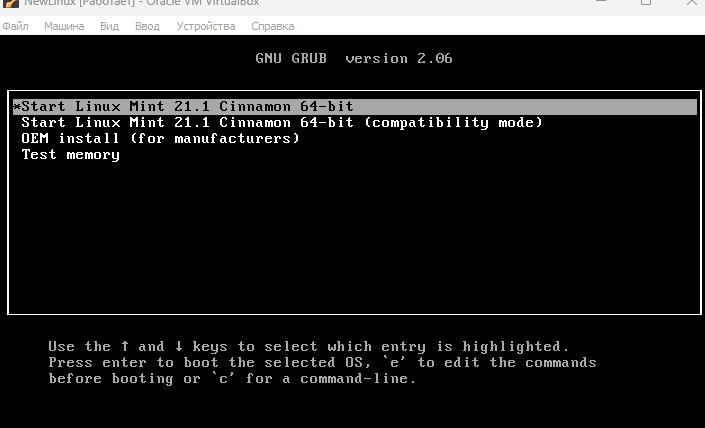


Создайте виртуальный жесткий диск объемом более 20 гигабайтов, иначе не хватит места даже для дальнейшей установки linux.

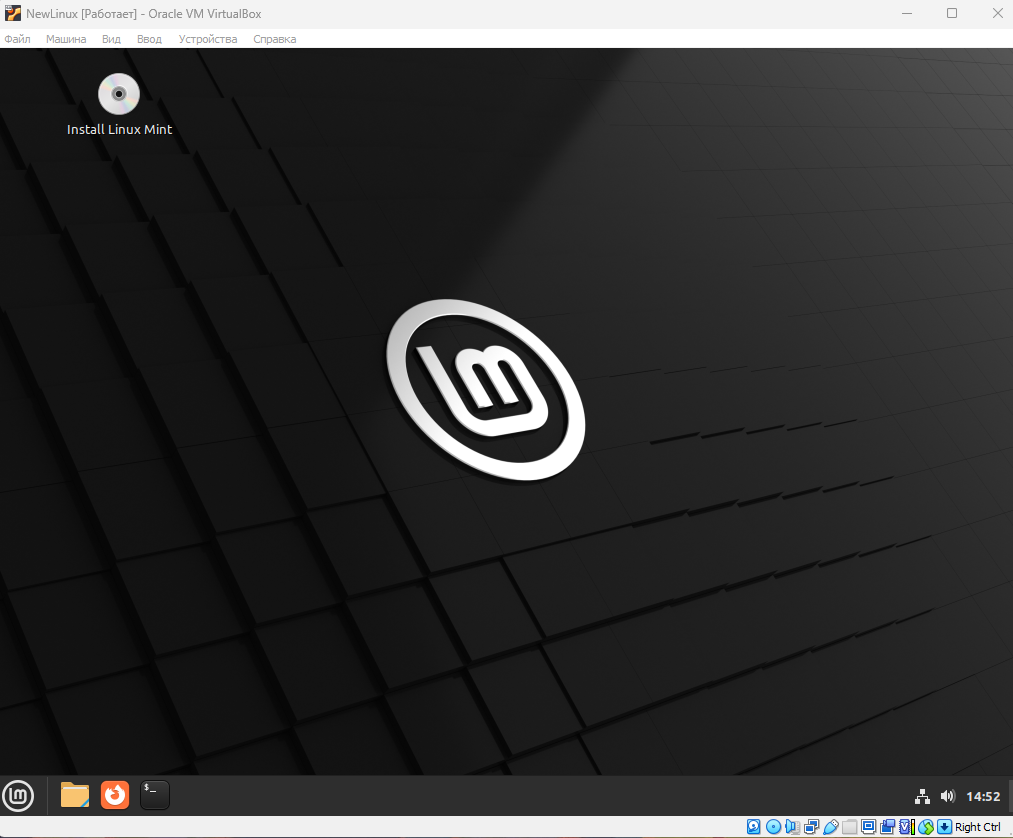
Если не ставить никаких галочек, по умолчанию стоит динамический диск. *Резервация места с настоящего диска произойдет, только после фактического заполнения виртуального.*

Нажимайте далее-готово. После некоторого ожидания можно запустить виртуальную машину и установить новую операцинную систему.





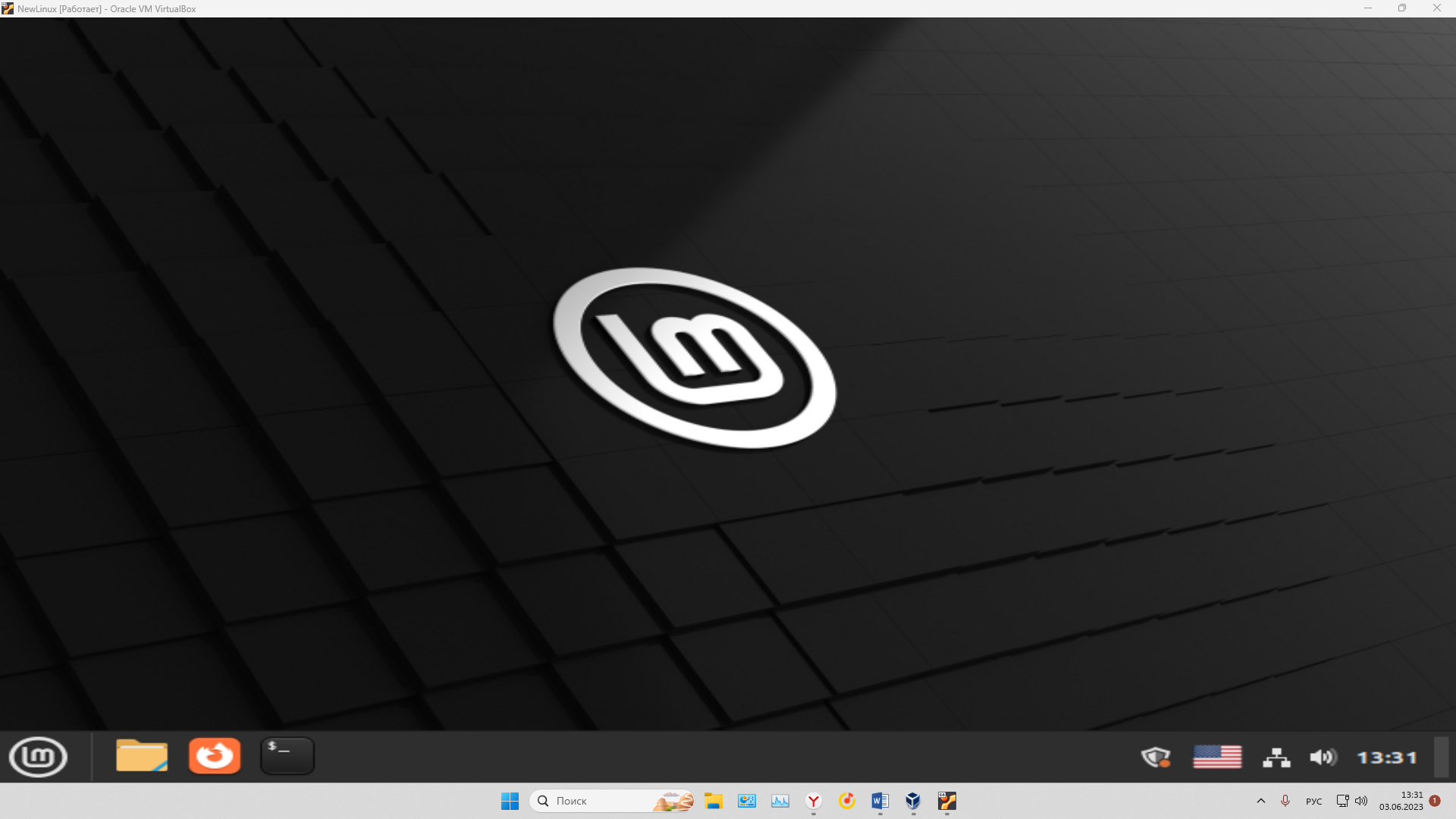
Сейчас линукс находится в виде iso образа на «виртуальной флешке». Выберите start. У вас запустится «переносная операционка» с флешки.



Последний шаг: запустите install и следуйте простым указаниями. После этого «переносная» операционка превратится в «стационарную».

На мой взгляд режим маштрабирования самый удобный. Вид-> масштабирование экрана.

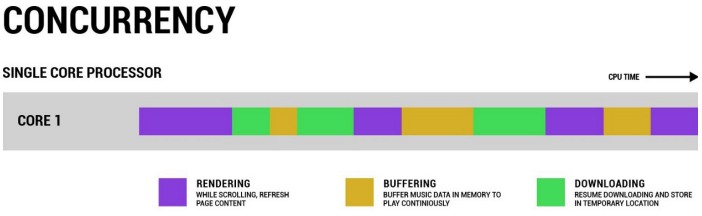
Более никаких преобразований не требуется.



## Реализация второй задачи

В компьютерном программировании конкурентность — способность компьютера справляться с множеством разных задач одновременно. Давайте возьмем для примера использование браузера. Если мы гуляем по сети, происходит много вещей одновременно. Например, идет скачивание файла в то время как мы слушаем музыку и скролим веб сайт в другой в кладке. Можно сказать, что компьютер выполняет несколько задач одновременно. Если бы компьютеры так не умели, нам бы пришлось ждать окончание скачивания файла, чтобы продолжить слушать музыку или скролить сайт.

Архитектура ЦПУ устроена таким образом, что одно ядро может выполнять лишь одно действие в единицу времени. В наши дни на рынке все еще есть множество одноядерных процессоров. Но если у вас компьютер с одноядерным процессором, то неужели мы не сможем скачивать файл, слушать музыку и листать новостную ленту в другой вкладке одновременно? Конечно сможем, а достигается это с помощью конкурентности. Давайте посмотрим диаграмму, которая демонстрирует как одноядерный ЦПУ справляется с примером с браузером.



Как мы можем увидеть, все задачи разбиваются на кусочки, которые делятся по приоритетности, а ЦПУ постоянно переключается между ними, создавая иллюзию одновременного исполнения.

Создадим программу на языке программирования JAVA, которая будет получать уже разбитые задачи и выполнять их. Программу реализуем с помощью событийно-ориентированной модели программирования. В центральном классе будет находится основной цикл, который выбирает и обрабатывает событие, при этом создаваться события будут в цикле.

Т.е. главный цикл обрабатывает список событий, в который динамически добавляются новые события.

У нас будет 3 типа событий: 1 – запрос новой задачи и предоставление доступа к процессору, 2 – выполнение задач, 3 – завершение работы программы при выполнении заранее запланированной задачи или при переполнении максимального возможного количества задач.

*Рассмотрим события более детально.*

*1 тип:* c самого начала у нас одно событие этого типа. При обработке происходит добавление задачи (увеличение L), добавление следующего события 1-го типа (с удалением обработанного), а если процессор свободен, то добавляется событие два (обработка только что добавленной задачи)

Время, которое идет вместе с кодом события указывает в какой момент времени оно случилось, т.е. выполняется моментально.

*2 тип:* данное событие символизирует конец обработки задачи и проверяет остались ли еще задачи. Если (L>0) есть еще необработанные задачи, то добавляется следующее событие 2.

Время, которое идем вместе с кодом события указывает в какой момент времени закончилась работа задачи.

**public** **class** **Proceses** {

//Количество необработанных процессов или очередь.

**static** **int** L=**0**;

//Состояние процессора, где "Free" - свободен (процессор не занят процессом), "Busy" - занят (процессор выполняет процесс).

**static** String S="Free";

//При переключении на "True" прекращается работа цикла.

**static** **boolean** finish=**false**;

//Тип события, определяет какой алгоритм запустится.

**static** **int** eventMode;

//Время события, определяет в какой момент времени работает алгоритм.

**static** **int** eventTime;

//Список процессов оформлен в виде двух массивов. typeEvent - тип события, eventTimeM - время события (для 1 типа указан момент выполненния,для 2 типа конец выполнения)

**public** **static** **int**[] typeEvent=**new** **int**[**2**];

**public** **static** **int**[] eventTimeM=**new** **int** [**2**];

//Количество выполненых процессов.

**static** **int** n=**0**;

//Максимальное допустимое количество процессов. По достижении этой отметки n, программа будет прервана.

**static** **int** nMax=**100**;

**public** **static** **void** **main**(String args[]){

L=**0**;

//Занят ли процессор выполнением задачи.

S="Free";

finish=**false**;

//Начальные значения, чтобы стартовать процесс (Иницилизация)

eventMode=**1**; //Режим

eventTime=**0**; //Время (используется для создания новых задач)

typeEvent=**new** **int** [**1**];

eventTimeM=**new** **int** [**1**];

//Количество задач

n=**0**;

//Максимальное допустимое количество задач

nMax=**100**;

typeEvent[**0**]=**3**; //Создаем задачу, иницилизирующий конец.

eventTimeM[**0**]=**120**;

**while** (finish==**false**){

**if**(typeEvent.length==**1**){//Если кол-во процессов равно 1, значит имеется только процесс иницилизирующий конец.

eventMode=**1**; //Тогда будет задействован сценарий "1"

} **else**

{

eventMode=typeEvent[**0**]; //Когда у нас есть другие задачи, то время и режим определяется списком задач.

eventTime=eventTimeM[**0**];

}

System.out.println("Режим: "+eventMode+". Время "+eventTime+". L: "+L);

System.out.println(Arrays.toString(typeEvent));

System.out.println(Arrays.toString(eventTimeM));

System.out.println("-----------------\n");

**switch**(eventMode){

**case** **1** -> A1();

**case** **2** -> A2();

**case** **3** -> A3();

}

}

}

**public** **static** **void** **plan**(**int** E,**int** T){

//System.out.println("plan ("+E+","+T+")");

**if** (E==**1**){

addElement();

typeEvent[typeEvent.length-**1**]=**1**;

eventTimeM[typeEvent.length-**1**]=T+getTask(**1**);

quickSort(eventTimeM,typeEvent,**0**,typeEvent.length -**1**);

//добавить в массивы в конец по событию и отсортировать. По времени

}

**if** (E==**2**){

addElement();

typeEvent[typeEvent.length-**1**]=**2**;

eventTimeM[typeEvent.length-**1**]=T+getTask(**2**);

quickSort(eventTimeM,typeEvent,**0**,typeEvent.length-**1** );

//добавить в масивы в конец по событию и остортировать.

}

**if** (E==**3**){

A3();

//isEnableadReadyForAgainEnter(true);

//COPYGUI.writeln("Программа завершила работу. Начат процесс,/n иницилизирующий завершение.");

}

}

//Предоставляет доступ к процессору

**public** **static** **void** **A1**(){

//System.out.println("A1");

**if** (typeEvent.length!=**1**)

{removeElement();

}

plan(**1**,eventTime);

**if** ("Free".equals(S)){

S="Busy";

plan(**2**,eventTime);

}**else**{

// System.out.println("L++");

L++;

}

// System.out.println("-----------------");

}

// o\_O

**public** **static** **void** **A2**(){

// System.out.println("A2");

n++;

**if**(n>=nMax){

System.out.println("Количество потоков превышено.");

plan(**3**,eventTime);

}**else**{

**if** (L>**0**){

plan(**2**,eventTime);

//System.out.println("L--");

L--;

**if** (typeEvent.length!=**1**)

{removeElement();

}

}**else**{

S="Free";

// plan(1,eventTime);

**if** (typeEvent.length!=**1**)

{removeElement();

}

}

}

// System.out.println("-----------------");

}

//Завершает работу приложения

**public** **static** **void** **A3**(){

//System.out.println("A3");

finish=**true**;

System.out.println("Программа завершила работу.");

}

//Увеличивает массивы. Добавляя сзади пустую ячейку

**public** **static** **void** **addElement**(){

**int**[] copy\_listTimeForStart = **new** **int**[typeEvent.length];

**int**[] copy\_listTimeForFinish = **new** **int**[typeEvent.length];

copy\_listTimeForStart=Arrays.copyOf(typeEvent, typeEvent.length);

copy\_listTimeForFinish=Arrays.copyOf(eventTimeM, typeEvent.length);

typeEvent = **new** **int**[typeEvent.length+**1**];

eventTimeM = **new** **int**[eventTimeM.length+**1**];

typeEvent=Arrays.copyOf(copy\_listTimeForStart, typeEvent.length);

eventTimeM=Arrays.copyOf(copy\_listTimeForFinish, eventTimeM.length);

}

//Следующие два метода сортируют массивы, сортировка выполняется по массиву времени.

**static** **int** **partition**(**int**[] array, **int**[]array1, **int** begin, **int** end) {

**int** pivot = end;

**int** counter = begin;

**for** (**int** i = begin; i < end; i++) {

**if** (array[i] < array[pivot]) {

**int** temp = array[counter];

**int** temp1= array1[counter];

array[counter] = array[i];

array1[counter] = array1[i];

array[i] = temp;

array1[i] = temp1;

counter++;

}

}

**int** temp = array[pivot];

**int** temp1=array1[pivot];

array[pivot] = array[counter];

array1[pivot]=array1[counter];

array[counter] = temp;

array1[counter] = temp1;

**return** counter;

}

**public** **static** **void** **quickSort**(**int**[] array, **int**[]array1, **int** begin, **int** end) {

**if** (end <= begin)

{

eventTimeM=array;

typeEvent=array1;

**return**;

}

**int** pivot = partition(array,array1, begin, end);

quickSort(array,array1, begin, pivot-**1**);

quickSort(array,array1, pivot+**1**, end);

}

//Возвращает случайное значение из двух разных диапозонов в зависимости от аргумента

**public** **static** **int** **getTask**(**int** i){

**double** time1=getRandomIntegerBetweenRange(**5**,**12**);

**double** time2=getRandomIntegerBetweenRange(**9**,**15**);

**if** (i==**1**){

**return** (**int**)(time1);

}

**else** {

**return** (**int**)(time2);

}

}

**public** **static** **double** **getRandomIntegerBetweenRange**(**double** min, **double** max){

**double** x = (**int**)(Math.random()\*((max-min)+**1**))+min;

**return** x;

}

//Удаляет первый элемент из массивов и сдвигает их

**public** **static** **void** **removeElement**(){

**int**[] copy\_listTimeForStart = **new** **int**[typeEvent.length];

**int**[] copy\_listTimeForFinish = **new** **int**[eventTimeM.length];

//System.out.println(Arrays.toString(typeEvent));

//System.out.println(Arrays.toString(eventTimeM));

copy\_listTimeForStart=Arrays.copyOf(typeEvent, typeEvent.length);

copy\_listTimeForFinish=Arrays.copyOf(eventTimeM, typeEvent.length);

typeEvent = **new** **int**[typeEvent.length-**1**];

eventTimeM = **new** **int**[eventTimeM.length-**1**];

**for**(**int** i=**0**;i<copy\_listTimeForStart.length-**1**;i++){

typeEvent[i]=copy\_listTimeForStart[i+**1**];

eventTimeM[i]=copy\_listTimeForFinish[i+**1**];

}

}

}

## Реализация третей задачи

В [вычислениях](https://ru.wikibrief.org/wiki/Computing) проблема производитель – потребитель (также известная как проблема ограниченного буфера ) является классическим примером проблемы синхронизации нескольких [процессов](https://ru.wikibrief.org/wiki/Process_(computing)), предложенной [Эдсгером В. Дейкстрой](https://ru.wikibrief.org/wiki/Edsger_W._Dijkstra). Проблема описывает два процесса, производитель и потребитель, которые совместно используют общий буфер [фиксированного размера](https://ru.wikibrief.org/wiki/Buffer_(computer_science)), используемый в качестве [очереди](https://ru.wikibrief.org/wiki/Queue_(data_structure)).

Проблему потребителей-производителей можно решить с помощью семафоров или с помощью оператора synchronized. Для начала определим задачи потребителей и производителей:

1. Производитель должен производить готовые товары и оставлять их в буфере.
2. Потребитель должен забирать товары из буфера, как только они появляются.

Семафоры и synchronized помогут нам решить следующие проблемы:

1. Доступ к буферу может быть одновременно разрешен только одним потоком.
2. Если буфер заполнен, производитель не может добавлять в него больше элементов.
3. Если буфер пуст, потребитель не может забирать из него элементов.

**Решение с помощью семафоров:**

1. Создаем 2 семафора - free и full. Первый указывает на количество пустых ячеек в буфере. Второй - на количество заполненных ячеек.
2. Инициализируем free со значением размера буфера.
3. Инициализируем full с нулем.
4. Поток производителя будет увеличивать значение full при добавлении элемента в буфер, уменьшать значение free при выходе из буфера.
5. Поток потребителя будет увеличивать значение free при удалении элемента из буфера, уменьшать значение full при выходе из буфера.

При этом на каждой итерации производитель проверяет значение semEmpty. Если оно равно 0, то производитель должен ждать освобождения ячеек в буфере.

Аналогично на каждой итерации потребитель проверяет значение semFull. Если оно равно 0, то потребитель должен ждать заполнения ячеек в буфере.

В итоге, использование семафоров позволяет нам синхронизировать работу потоков, чтобы избежать ошибок, связанных с доступом к общим ресурсам.

**Решение с помощью synchronized:**

1. Создаем и помечаем методы addElem и readElem модификатором доступа synchronized.
2. Заполняем, новые методы, следуя следующей логике.

addElem будет переходить в ожидание если место кончилось, а readElem будет переходить в ожидание если нечего читать.

Если readElem прочитал заполненный массив, то он пробудит все ожидающие потоки.

Если addElem заполнил пустой массив, то он пробудит все ожидающие потоки.

App.java

**public** **class** **App**{

**public** **static** **void** **main**(String[] args) {

// Можно использовать конструкторы любой реализации ресурса

IBuffer b = **new** MonitorSecureBuffer(**8**);

//Создаем читателей и писателей

List<Thread> lt = **new** ArrayList<>();

//Теперь помещаем потоки.

**for** (**int** i=**0**;i<**3**;i++) lt.add(**new** Thread(**new** Writer(b)));

**for** (**int** i=**0**;i<**1**;i++) lt.add(**new** Thread(**new** Reader(b)));

//Стартуем читателей и писателей

**for** (Thread thread : lt) {

thread.start();

}

**try** {

Thread.sleep(**3000**); // блокируем поток на 3000 мс = 3 с

} **catch** (InterruptedException e) {}

**for** (Thread thread : lt) {

thread.interrupt();

}

}

}

IBuffer.java

**public** **interface** **IBuffer** {

**public** **void** **addElem**();

**public** **void** **readElem**();

}

InSecureBuffer.java

**public** **class** **InSecureBuffer** **implements** IBuffer {

**protected** **int** n;

**protected** **int** index=**0**;

**private** **int**[] li;

**public** **InSecureBuffer**(**int** n) {

**super**();

**this**.n = n;

**this**.li = **new** **int**[n];

}

// ------------------------------------------------------

**protected** **void** **add**() {

li[index] = **0**;

System.out.println("Записали 0 в ячейку "+index);

index++;

}

**protected** **void** **read**() {

System.out.println("Считали "+li[index-**1**] +" из ячейки "+(index-**1**));

index--;

}

//-----------------------------------------------------

// Обрабатываются в потоках

**@Override**

**public** **void** **addElem**() {

add();

System.out.println("\*\*\*\*\*");

}

**@Override**

**public** **void** **readElem**() {

read();

System.out.println("\*\*\*\*\*");

}

}

LockSemSecureBuffer.java

**public** **class** **LockSemSecureBuffer** **extends** InSecureBuffer **implements** IBuffer{

**private** **final** Lock l = **new** ReentrantLock();

**private** **final** Semaphore free = **new** Semaphore(n);

**private** **final** Semaphore full = **new** Semaphore(**0**);

**public** **LockSemSecureBuffer**(**int** n) {

**super**(n);

}

**public** **void** **addElem**() {

**try** {

free.acquire();

} **catch** (InterruptedException e1) {

Thread.currentThread().interrupt();

}

l.lock();

**try** {

add();

}

**catch** (Exception e) {}

**finally** {

l.unlock();

}

full.release();

}

//-----------------------------------

**public** **void** **readElem**() {

**try** {

full.acquire();

} **catch** (InterruptedException e1) {

Thread.currentThread().interrupt();

}

l.lock();

**try** {

read();

}

**catch** (Exception e) {}

**finally** {

l.unlock();

}

free.release();

}

}

MonitorSecureBuffer.java

**public** **class** **MonitorSecureBuffer** **extends** InSecureBuffer **implements** IBuffer{

**public** **MonitorSecureBuffer**(**int** n) {

**super**(n);

}

**@Override**

**public** **synchronized** **void** **addElem**() {

**while** (index==n)

**try** {

wait();

} **catch** (InterruptedException e) {

Thread.currentThread().interrupt();

**break**;

}

**if** (index!=n) {

add();

**if** (index==**1**) notifyAll();

}

}

**@Override**

**public** **synchronized** **void** **readElem**() {

**while** (index==**0**)

**try** {

wait();

} **catch** (InterruptedException e) {

Thread.currentThread().interrupt();

**break**;

}

**if** (index!=**0**) {

read();

**if** (index==(n-**1**)) notifyAll();

}

}

}

Reader.java

**public** **class** **Reader** **implements** Runnable {

**private** IBuffer b; // ссылка на объект ресурса

// Конструктор с нинициализацией

**public** **Reader**(IBuffer b) {

**super**();

**this**.b = b;

}

**@Override**

**public** **void** **run**() {

**while**(**true**) {

// Проверяем, был ли получен сигнал на прерывание потока, если да, то выходим

// из цикла и завершаем работу потока

**if** (Thread.currentThread().isInterrupted()) **break**;

//Обращаемся к ресурсу и считываем элемент из массива

b.readElem();

//Проверяем, был ли получен сигнал на прерывание потока, если да, то

//выходим из цикла и завершаем работу потока

**try** {

Thread.sleep(**10**);

}

**catch** (InterruptedException e) {

**break**;

}

}

System.out.println("Читатель остановлен...");

}

}

Writer.java

**public** **class** **Writer** **implements** Runnable{

**private** IBuffer b; // ссылка на объект ресурса

// Конструктор с нинициализацией

**public** **Writer**(IBuffer b) {

**super**();

**this**.b = b;

}

**@Override**

**public** **void** **run**() {

**while**(**true**) {

// Проверяем, был ли получен сигнал на прерывание потока, если да, то выходим

// из цикла и завершаем работу потока

**if** (Thread.currentThread().isInterrupted()) **break**;

//Обращаемся к ресурсу и записываем элемент в массив

b.addElem();

// Проверяем, был ли получен сигнал на прерывание потока, если да, то

//выходим из цикла и завершаем работу потока

**try** {

Thread.sleep(**10**);

} **catch** (InterruptedException e) {

**break**;

}

}

System.out.println("Писатель остановлен...");

}

}

## З**аключение**

1. Как мы видим, установка операционной системы на виртуальную машину не является сложной задачей, но требует некоторой подготовки и внимательности. Следуя указанным шагам, вы сможете успешно установить ОС на свою виртуальную машину.
2. Эта программа будет очень полезной для тех, кто работает с потоками и процессами. Она позволит имитировать обработку потоков процессором и оптимизировать работу программы.
3. Выводы по решению задачи читатель-писатель с помощью многопоточности показывают, что многопоточность позволяет эффективно работать с ресурсами и обеспечивать безопасность данных. Однако, при работе с многопоточностью необходимо учитывать множество факторов, чтобы избежать ошибок в работе программы. Надеемся, что предложенные в нашем решении методы позволят вам безопасно и эффективно работать с потоками.

## Список литературы

<https://coderbooks.ru/books/go/yazyk_go_dlya_nachinayushchih_zhashkevich_2020/>

<https://ru.wikipedia.org/wiki/Критическая_секция>

<https://gb.ru/blog/linux/>

<https://github.com/verliokann/concurrencyhw>

<https://metanit.com/java/tutorial/>

<https://ru.wikibrief.org/wiki/Producer–consumer_problem>