Санкт-Петербургский политехнический университет Петра Великого

Институт компьютерных наук и кибербезопасности

Высшая школа компьютерных технологий и информационных систем

**КУРСОВАЯ РАБОТА**

**Работа с потоками в ОС Linux.**

**Тема: “Банковские кассы ”**

по дисциплине «Операционные системы»

Выполнил

студенты гр. 5130901/10201 Е.Т. Вайцеховский

Руководитель

Старший преподаватель З.В. Куляшова

«\_\_\_» \_\_\_\_\_\_\_\_\_\_ 2024 г.

Санкт-Петербург

2024

**Оглавление**

[1. Введение. Постановка цели исследования и описание задач для достижения поставленной цели. 3](#_Toc181540777)

[2. Описание задачи, которая будет реализована в приложении. 3](#_Toc181540778)

[3. Описание базового алгоритма функционирования 4](#_Toc181540779)

[4. Сравнение работы программы в однопоточном и многопоточном режимах 8](#_Toc181540780)

[5. Вывод 9](#_Toc181540781)

[6.Листинг программы 10](#_Toc181540782)

# 1. Введение. Постановка цели исследования и описание задач для достижения поставленной цели.

Современные программные системы позволяют обрабатывать одновременно множество операций, используя многопоточность для повышения эффективности и ускорения работы приложений. Это особенно важно в системах с ограниченными ресурсами и высокой степенью конкуренции за доступ к ним, как, например, в банках, где клиенты ожидают обслуживания у касс. В реальной жизни такие задачи предполагают координацию множества параллельных процессов: управление очередями, обслуживание различными операторами, выполнение транзакций и соблюдение очередности клиентов. Задачи подобного рода позволяют продемонстрировать возможности многопоточной обработки данных для оптимизации обслуживания и снижения времени ожидания.

**Целью данной работы** является исследование эффективности многопоточной обработки на примере моделирования работы банковских касс. В рамках данной задачи предполагается разработка системы, где несколько операторов (касс) обслуживают клиентов банка. Важно учитывать, что каждый оператор может обрабатывать определенные виды услуг, и не все кассы могут быть доступны для каждого типа оплаты.

Для достижения цели исследования были поставлены следующие задачи:

1. Проанализировать и определить требования к системе, включая возможные типы услуг и ограничения для различных операторов.
2. Спроектировать многопоточную модель для обслуживания очереди клиентов, распределяя их по доступным кассам с учетом особенностей каждого типа оплаты.
3. Реализовать модель на языке Java, обеспечив безопасную синхронизацию потоков и корректное управление доступом к ограниченным ресурсам (кассам).
4. Провести сравнительный анализ эффективности многопоточной и однопоточной реализации задачи с точки зрения времени обработки и ожидания клиентов.
5. Сделать выводы о преимуществах и недостатках многопоточного подхода для решения задачи управления очередями и распределения ресурсов в условиях ограничений.

# 2. Описание задачи, которая будет реализована в приложении.

Ограниченное количество касс (операторов в банке), разные виды услуг (оплат) и клиенты организации. Возможна модификация, что не все кассы (операторы) могут брать все варианты оплаты.

# 3. Описание базового алгоритма функционирования

Код представляет собой простую систему, моделирующую работу кассиров, обслуживающих клиентов. Давайте рассмотрим основные шаги и алгоритм действий этой программы:

* Класс Client: содержит статические переменные, хранящие информацию о клиенте (имя, тип услуги, оплата и т.д.).

Изображение выглядит как текст, Шрифт, снимок экрана

Автоматически созданное описание

* Класс Cashier: расширяет Thread и представляет собой поведение кассира.

Изображение выглядит как текст, снимок экрана, дисплей, программное обеспечение

Автоматически созданное описание

Кассир работает до тех пор, пока работает Thread WorkingDay, т.е. пока не закончится рабочий день. Внутри метода run(), кассир следит за временем, после 12 часов кассир имеет право уйти на обед на 1 час времени. В рабочее время кассир обслуживает клиента, «доставая» его из очереди clients, затраченное время и заработанные деньги зависят от услуги клиента. После того, как клиент был обслужен, к дневной выручке добавляется комиссия только что обслуженного клиента.

По окончании рабочего дня кассиру выплачивается зарплата в размере 3000 рублей.

* Класс WorkingDay расширяет Thread и представляет собой реализацию рабочего дня кассира.

Изображение выглядит как текст, снимок экрана, программное обеспечение

Автоматически созданное описание

WorkingDay является отдельной нитью и считает рабочее время. Рабочий день начинается с 9 утра и заканчивается в 18 вечера, чтобы ускорить симуляцию было принято решение взять следующее соотношение времени симуляции и реального времени:

1 секунда реального времени = 30 минут симуляции.

* Класс ServiceType, в котором содержатся все типы услуг банка.

Изображение выглядит как текст, снимок экрана, программное обеспечение, Шрифт

Автоматически созданное описание

Для каждой услуги была установлена комиссия банка и время его выполнения. Например, услуга MAKING\_LOAN (оформление кредита) принесёт комиссию в размере 5000 рублей и на эту услугу будет затрачиваться 1,5 часа времени симуляции.

* Метод GetClientGenerator(), который генерирует клиентов со случайными типами требуемых услуг

Изображение выглядит как текст, снимок экрана, программное обеспечение

Автоматически созданное описание

Данный метод работает в отдельном потоке. Каждому клиенту присваивается порядковый номер и случайная требуемая услуга. После создания экземпляра класса Client, экземпляр помещается в потокобезопасную очередь (ArrayBlockingQueue) clients.

* Основной метод main, который, выделяет некоторое количество кассиров на работу

Изображение выглядит как текст, снимок экрана

Автоматически созданное описание

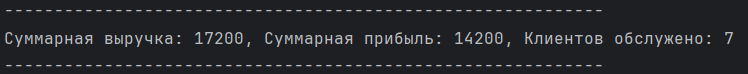
Данный метод работает в отдельном потоке. Внутри метода находится цикл, который генерирует некоторое количество кассиров, в зависимости от шага цикла. Таким образом, такая реализация генерации кассиров позволит увидеть принцип работы в однопоточном (1 кассир) и многопоточном режиме (>1 кассира), а также позволит выбрать наилучшее количество кассиров для получения максимальной прибыли. Перед началом новой итерации цикла for начинается новый рабочий день (запускается нить WorkingDay) и создаётся новая очередь (очищается старая очередь и пересоздаётся нить GetClientsGenerator()). В конце дня выводится дневная выручка, дневная прибыль (дневная выручка – зарплата кассиров) и количество обслуженных клиентов.

Предложенный код представляет базовую симуляцию работы кассиров, но может быть улучшен для обработки ошибок, улучшения пользовательского опыта и эффективного управления многопоточностью.

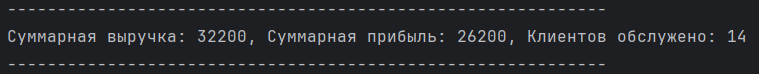
# 4. Сравнение работы программы в однопоточном и многопоточном режимах

Число потоков соответствует числу кассиров, которые вышли на рбаоту. Для сравнения было проведено десять тестов: 1 кассир, 2 кассира и т. д. Различия между тестами заключаются только в числе одновременно работающих кассиров. На рисунках приведены примеры с режимами однопоточной и многопоточной работы программы.

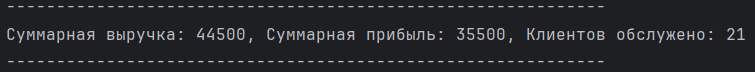
1. Результаты симуляции для 1 кассира



1. Результаты симуляции для 2-х кассиров



1. Результаты работы для 3-х кассиров

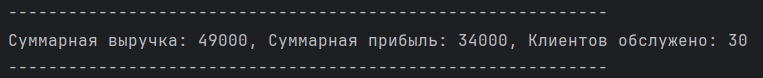


1. Результаты работы для 4-х кассиров

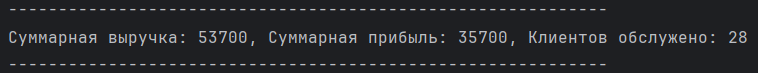
Изображение выглядит как снимок экрана, текст, Шрифт, линия

Автоматически созданное описание

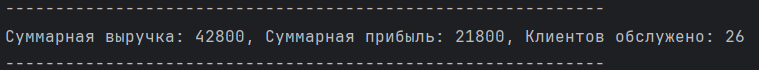
1. Результаты работы для 5-х кассиров



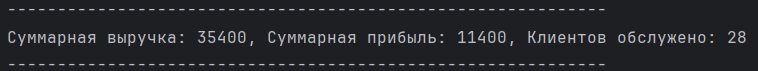
1. Результаты работы для 6-х кассиров



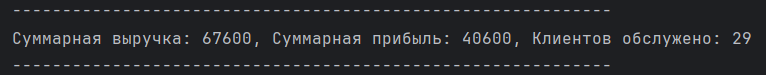
1. Результаты работы для 7-х кассиров



1. Результаты работы для 8-х кассиров



1. Результаты работы для 9-х кассиров



Из предложенных результатов работы можно сделать некоторые выводы для максимизации прибыли. 1, 2, 3 кассира, вышедших на работу будет недостаточно, т. к. такое количество кассиров не успевает обслужить дневной поток клиентов (по результатам тестирования видно, что средний дневной поток клиентов составляет 27–30 человек). 4–6 кассира, являются оптимальным решением задачи. Такое количество кассиров, позволяет обслужить весь дневной поток, при этом, дневные расходы, также остаются небольшими из-за небольшого количества кассиров, что приводит к максимальной прибиыли. 7–9 кассиров являются в среднем избыточным решением, т. к. они успевают обслужить весь дневной поток, при этом, дневная прибыль становиться меньше, по сравнению с 4–6 кассирами, т. к. увеличиваются дневные расходы вследствие большего числа кассиров.

Последний результат не подчиняется предложенному выше распределению и является исключительным случаем, т. к. большая часть клиентов в этот день пришла в банк с услугой MAKING\_LOAN, которая является дорогостоящей.

# 5. Вывод

Из этой работы можно сделать вывод о максимально использовании многопоточности в приложениях и операционных системах. Так, преждевременное суждение о том, что с увеличением количества потоков, работа некоторой системы является более эффективной, не является всегда верным. Важно правильно оценивать затрачиваемые ресурсы для запуска и поддержания потока для каждой системы, чтобы оценить и найти максимально «выгодную» конфигурацию, которая повозлит затрачивать наименьшее количество ресурсов и приносить наилучший результат.

# Листинг программы

Класс Bank:

import java.time.LocalTime;  
import java.util.Arrays;  
import java.util.concurrent.ArrayBlockingQueue;  
import java.util.concurrent.TimeUnit;  
import java.util.concurrent.atomic.AtomicInteger;  
  
  
public class Bank {  
 private final static ArrayBlockingQueue<Client> *clients* = new ArrayBlockingQueue<>(40);  
 private static WorkingDay *workingDay*;  
 private static Integer *dailyRevenue*;  
 private static Integer *dailyClientsServed* ;  
 public static void main(String[] args) {  
 long startWorking = System.*currentTimeMillis*();  
 for (int i = 1 ; i < 10 ; i++) {  
 *dailyRevenue* = 0;  
 *dailyClientsServed* = 0;  
 *clients*.clear();  
 *workingDay* = new WorkingDay();  
 *workingDay*.start();  
 *clients*.clear();  
 *GetClientGenerator*().start();  
 System.*out*.println("Количество кассиров, которые выйдут на работу: "+ i);  
 Cashier[] cashiers = new Cashier[i];  
 System.*out*.println("Начало рабочего дня");  
 for (int j = 0; j < i; j++) {  
 cashiers[j] = new Cashier();  
 cashiers[j].setName("Кассир №" + (j + 1));  
 System.*out*.println(cashiers[j].getName() + " Вышел на работу");  
 cashiers[j].start();  
 }  
 Arrays.*stream*(cashiers).forEach(cashier -> {  
 try {  
 cashier.join();  
 } catch (InterruptedException e) {  
 throw new RuntimeException(e);  
 }  
 });  
 System.*out*.println("------------------------------------------------------------");  
 System.*out*.printf("Суммарная выручка: %d, Суммарная прибыль: %d, Клиентов обслужено: %d \n",  
 *dailyRevenue*, *dailyRevenue* - i \* Cashier.*dailySalary*, *dailyClientsServed*);  
 System.*out*.println("------------------------------------------------------------");  
 }  
 System.*out*.println("Конец программы: " + (System.*currentTimeMillis*() - startWorking));  
 }  
  
 private static Thread GetClientGenerator() {  
 AtomicInteger order = new AtomicInteger(1);  
 Thread thread = new Thread(() -> {  
 while (*workingDay*.isAlive()) {  
 ServiceType serviceType = switch ((int) (Math.*random*() \* 3) + 1) {  
 case 1 -> ServiceType.*MAKING\_LOAN*;  
 case 2 -> ServiceType.*CREDIT\_CARD*;  
 case 3 -> ServiceType.*CURRENCY\_EXCHANGE*;  
 default -> throw new IllegalStateException("Unexpected value");  
 };  
 *clients*.offer(new Client(order.getAndIncrement(), serviceType));  
 try {  
 Thread.*sleep*((int) (Math.*random*() \*  
 WorkingDay.*getHalfAnHourDurationToMills*()) + 200);  
 } catch (InterruptedException e) {  
 throw new RuntimeException(e);  
 }  
 }  
 });  
 return thread;  
 }  
  
 static class Cashier extends Thread {  
 boolean isRested;  
 static int *dailySalary* = 3000;  
 @Override  
 public void run() {  
 while (*workingDay*.isAlive()) {  
 if (!isRested && *workingDay*.getCurrentTime().isAfter(LocalTime.*of*(12,0))) {  
 isRested = true;  
 System.*out*.println(this.getName() + " Ушёл на обед на 1 час");  
 try {  
 Thread.*sleep*(2L \* WorkingDay.*getHalfAnHourDurationToMills*());  
 } catch (InterruptedException e) {  
 throw new RuntimeException(e);  
 }  
 }  
 Client client;  
 try {  
 client = *clients*.poll(5, TimeUnit.*SECONDS*);  
 System.*out*.printf("%s Обслуживает клиента №%d. Тип услуги: %s\n",  
 this.getName(), client.order(), client.serviceType().name());  
 try {  
 Thread.*sleep*(client.serviceType().getDuration());  
 *dailyRevenue* += client.serviceType().getPrice();  
 *dailyClientsServed*++;  
 } catch (InterruptedException e) {  
 throw new RuntimeException(e);  
 }  
 System.*out*.println(this.getName() + " Обслужил клиента №" + client.order());  
 } catch (Throwable \_) {  
 }  
 }  
 }  
 }  
}

Класс Clinet:

public record Client(  
 int order,  
 ServiceType serviceType  
) {  
}

Класс ServiceType:

public enum ServiceType {  
  
 *MAKING\_LOAN*(5\_000, WorkingDay.*getHalfAnHourDurationToMills*() \* 3),  
 *CREDIT\_CARD*(1000, (int) (WorkingDay.*getHalfAnHourDurationToMills*() \* 2.5)),  
 *CURRENCY\_EXCHANGE*(100, (int) (WorkingDay.*getHalfAnHourDurationToMills*() \* 1.3));  
  
 private final int price;  
 private final int duration;  
  
 ServiceType(int price, int duration) {  
 this.price = price;  
 this.duration = duration;  
 }  
 public int getPrice() {  
 return price;  
 }  
 public int getDuration() {  
 return duration;  
 }  
}

Класс WorkingDay:

import java.time.LocalTime;  
  
public class WorkingDay extends Thread{  
 // 30 минут симуляции = 1 секунда  
 private static final int *halfAnHourDurationToMills* = 1000;  
 private static LocalTime *currentTime*;  
 @Override  
 public void run() {  
 *currentTime* = LocalTime.*of*(9,0);  
 while(*currentTime*.isBefore(LocalTime.*of*(18,0))) {  
 try {  
 Thread.*sleep*(*halfAnHourDurationToMills*);  
 *currentTime* = *currentTime*.plusMinutes(30);  
 } catch (InterruptedException e) {  
 throw new RuntimeException(e);  
 }  
  
 }  
 System.*out*.println("Рабочий день закончен!");  
 }  
 public LocalTime getCurrentTime() {  
 return *currentTime*;  
 }  
  
 public static int getHalfAnHourDurationToMills() {  
 return *halfAnHourDurationToMills*;  
 }  
}