Engenharia Informática 23/24

**Sistemas Operativos**

Relatório - **satelliteSimulation**

horizontal line

# 

Realizado por:

Sónia Oliveira 8220114 LEI

João Santos 8220256 LEI

Ian Dinarte 8220005 LEI

# 

# Índice:

Resumo……………………………………………………………………………………………………………………………………….2

Introdução…………………………………………………………………………………………………………………………………3

Design………………………………………………………………………………………………………………………………………….4

Implementação…………………………………………………………………………………………………………………….4-17

Conclusão………………………………………………………….……………………………………………………………………… 17

# Resumo:

Este trabalho aborda a simulação do sistema operativo de um satélite com o objetivo de projetar a comunicação entre processos recorrendo a **threading** em java. O sistema inclui componentes que representam um **middleware**, **kernel**, **cpu**, **memmory** e **tarefas executáveis** (runnable tasks), cada um desempenhando funções distintas.  
  
A comunicação entre estes componentes é realizada através de várias estruturas de dados, como filas (queues), listas ligadas e listas ordenadas.

Alguns destes componentes, o CPU e tarefas executáveis trabalham com recurso a threads e todos os outros componentes utilizam elementos de sincronização para prevenir o aparecimento de problemas de starvation/deadlock e concorrência.

Este projeto oferece uma visão abrangente da arquitetura do sistema, destacando a importância dos vários mecanismos de sincronização e a implementação do código para garantir o funcionamento eficaz do sistema operacional simulado.

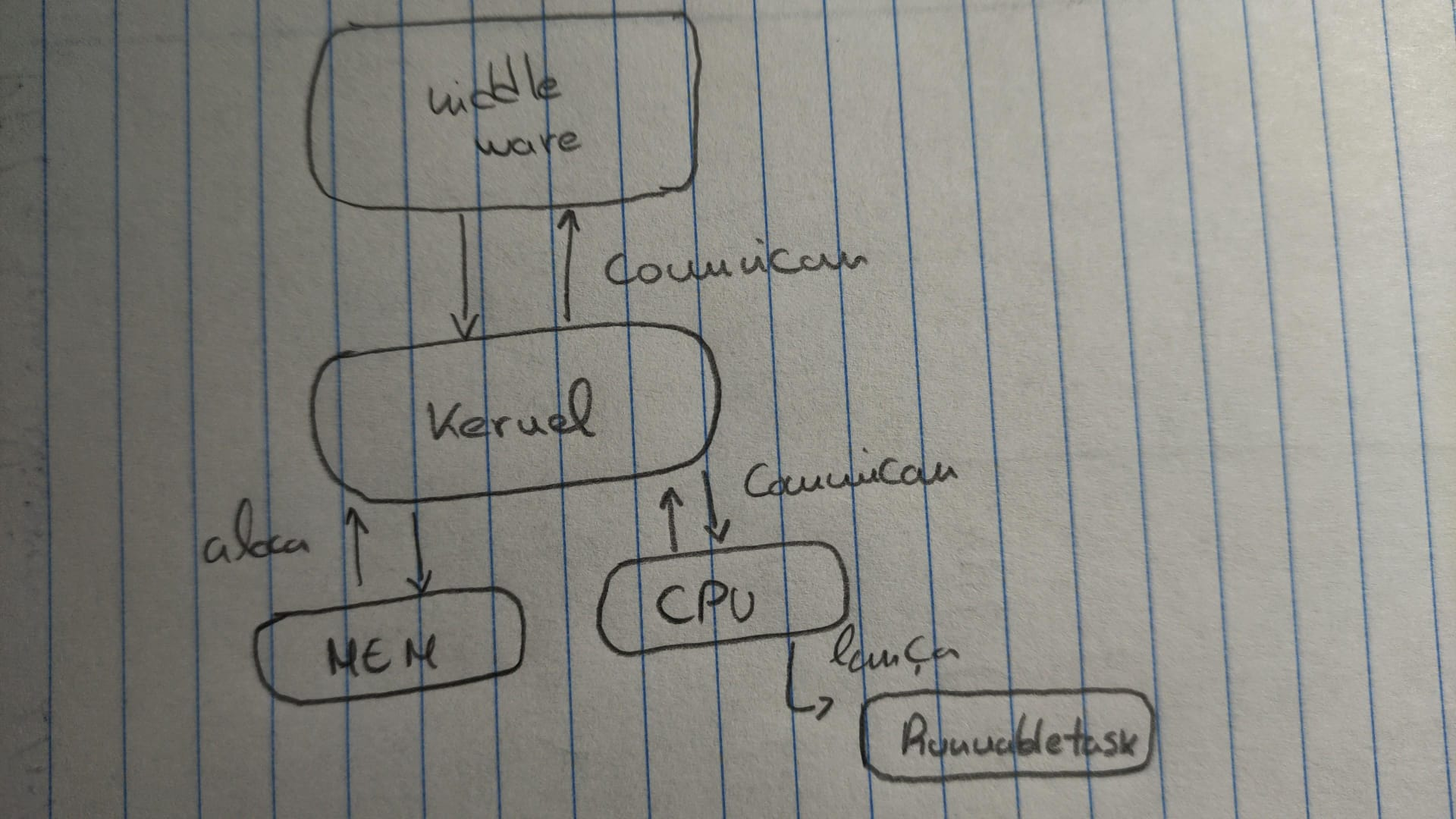
**Introdução:   
Este relatório tem como objetivo documentar a evolução da implementação do código** para o projeto de sistemas operativos, no qual simulamos o sistema operativo de um satélite.

**O projeto tem como objetivo refletir o conhecimento do grupo sobre threads** e programação em java e o **conhecimento de vários algoritmos de escalonamento** lecionados na cadeira de Sistemas Operativos I.

Este foi um trabalho que se demonstrou como um grande desafio, maioritariamente pela vaguidade do enunciado do trabalho o que permite uma enorme liberdade na tomada de decisões no que toca à implementação do projeto.  
  
O grupo decidiu criar um sistema operativo, onde temos 3 camadas de comunicação, a primeira o middleware, com o qual o utilizador interage, a segunda sendo o kernel cuja responsabilidade é de receber as tarefas a ser cumpridas pelo middleware, enviar as mesmas para o CPU, onde estas serão executadas e ao mesmo tempo, alocar a memória presente no kernel.   
  
Fundamentalmente, podemos assumir que o middleware apenas tem conhecimento do kernel, o kernel tem conhecimento do middleware, do cpu e da memória; O CPU tem conhecimento apenas do kernel, e a memória apenas tem conhecimento do kernel.

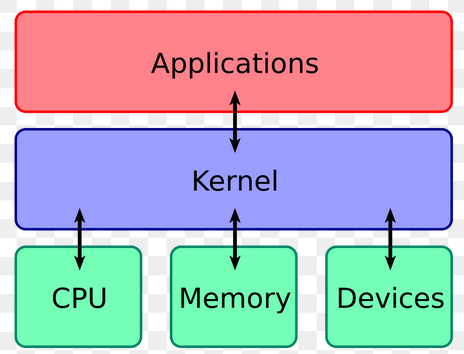
**Design:**

A arquitetura do sistema é composta por cinco componentes principais: middleware, kernel, CPU, memória e tarefas executáveis. Estes elementos interagem através de várias estruturas de dados, para garantir uma comunicação eficiente e. Diagramas detalhados ilustram a estrutura e os princípios de design adotados para cada componente.



***. Diagrama de componentes para o programa***

***-*** Este diagrama tira influencia direta do seguinte e outros parecidos online acessíveis online

******

Em termos de diagrama de classes, ele é quase igual ao diagrama de classes, pois vimo-nos obrigados a implementar o middleware, o kernel, o CPU e a MEM como classes “one to one” do diagrama de componentes.

**Implementação:**

**Classe Middleware:**

private Kernel kernel;  
private SinglyLinkedOrderedList<RunnableTaskClass> bufferInMiddleware;  
private Semaphore semaphoreForSendingTasks;  
private Queue<RunnableTaskClass> middlewareRegistryOfCompletedTasks;

public Middleware () {  
 this.kernel = new Kernel(this);  
 this.bufferInMiddleware = new SinglyLinkedOrderedList<>();  
 this.semaphoreForSendingTasks = new Semaphore(5); // size 5 para o bufferInMiddleware  
 this.middlewareRegistryOfCompletedTasks = new LinkedBlockingQueue<>();  
}

public void sendJobToMiddleware(RunnableTaskClass taskToSend) {  
 try {  
 semaphoreForSendingTasks.acquire();  
 } catch (InterruptedException e) {  
 e.printStackTrace();  
 }   
 synchronized (bufferInMiddleware) {  
 for (RunnableTaskClass task : bufferInMiddleware) {  
 task.decreasePriority();  
 }  
 bufferInMiddleware.add(taskToSend);   
 }  
 RunnableTaskClass taskToSendToKernel;  
 synchronized (bufferInMiddleware) {  
 taskToSendToKernel = bufferInMiddleware.removeFirst();  
 }  
 kernel.receiveTaskFromMiddleware(taskToSend);  
}

* Este método é utilizado para **lançar tasks definidas pelo utilizador para o kernel**. Neste método utilizámos semáforos para controlar que nunca haveriam mais do que 5 tasks a correr no programa a qualquer ponto. Este mesmo semáforo é libertado apenas quando o kernel recebe a tarefa de volta quando esta foi executada.
* Partes críticas do código são

synchronized (bufferInMiddleware){

taskToSendToKernel = bufferInMiddleware.removeFirst();

}

synchronized (bufferInMiddleware) {

for (RunnableTaskClass task : bufferInMiddleware) {

task.decreasePriority();

}

bufferInMiddleware.add(taskToSend);

}

E estão precavidas pela palavra-chave synchronized para prevenir que múltiplas threads modifiquem o buffer concorrentemente, o mesmo acontece para as prioridades.

public void turnOnOperatingSystem() {  
 System.out.println("buenas tardes :)");  
 kernel.turnOn();  
 System.out.println("casa abierta");  
}  
  
synchronized public void turnOffOperatingSystem() {  
 System.out.println("buenas noches :(");  
 kernel.shutdown();  
 System.out.println("Estoy cansado...");  
}

- Estes métodos são responsáveis apenas por **simular o início e término** **da sessão dentro do sistema.**

public synchronized void receiveTaskInfoAfterCompletion() {  
  
 RunnableTaskClass answeredTask = kernel.sendTaskToMiddleware;  
  
 synchronized (middlewareRegistryOfCompletedTasks) {  
 if (answeredTask != null) {  
 middlewareRegistryOfCompletedTasks.add(answeredTask);  
 }  
 }  
 semaphoreForSendingTasks.release(); //testar com isto aqui ou antes do synchronized block  
 //System.out.println("dei release ao semáforo");  
}

- Este método é responsável **por receber as tasks que já foram executadas** por parte do kernel e do CPU. É precavido pela palavra chave synchronized, pois se trata de um **objeto partilhado** (ou seja, todas as threads que correm no programa vão querer aceder ao middlewareRegistryOfCompletedTasks quando elas forem completas e enviadas pelo CPU).

**Este middleware usa semáforos para controlar que apenas há 5 tasks a qualquer momento dentro do programa pois foi o que entendemos por “O middleware deve ter um buffer de capacidade 5“.**

**Classe Kernel:**

public class Kernel {  
  
 private final Middleware middleware;  
 public RunnableTaskClass sendTaskToMiddleware;  
 private MEM RAM\_Controlled\_By\_Kernel;  
 private List<RunnableTaskClass> tasksToExecute;  
 private SinglyLinkedOrderedList<RunnableTaskClass> tasksExecutedAndReadyToSendToMiddleware;  
 private volatile boolean isOn;  
 private volatile boolean isShuttingDown;  
 private CPU CPU\_Controlled\_By\_Kernel;  
 Thread cpuThread;

* A classe kernel tem as seguintes variáveis, middleware MEM e CPU pois são os componentes que o kernel conhece.
* A lista tasksToExecute e tasksExecutedAndReadyToSendToMiddleware são listas ordenadas onde as tasks com menor priority ficam à frente da lista. Por default, uma tarefa começa com prioridade 10.
* Boolean isOn e isShuttingDown são flags utilizadas pelo CPU para saber quando deve parar de operar. São volative pois são partilhadas pelas threads. A palavra chava volative certifica que uma thread tem acesso à modificação mais recente da variável e sem esta palavra, o nosso métodoshutDown não funcionaria.
* public synchronized void AllocateMemmory(RunnableTaskClass task) {  
   if (RAM\_Controlled\_By\_Kernel.getUsedRAM() + (int) task.Getmemmory() > 200) {  
   try {  
   System.out.println("Not enough memory, waiting...");  
   wait();  
   } catch (InterruptedException e) {  
   throw new RuntimeException("Thread interrupted while waiting for memory", e);  
   }  
   }  
   this.RAM\_Controlled\_By\_Kernel.alocateMemmory((int) task.Getmemmory());  
  }
* Este método é utilizado pelo kernel para alocar memória. A thread que tenta alocar memória e encontra uma situação onde não haja memória que chegue, ela fica em wait().
* public synchronized void DeallocateMemmory(RunnableTaskClass task) {  
   this.RAM\_Controlled\_By\_Kernel.deallocateMemmory((int) task.Getmemmory());  
   notifyAll();  
  }
* Este método desaloca memória e notifica qualquer thread que esteja à espera de poder alocar mais memória, utilizando o método notifyAll().
* public void receiveTaskFromMiddleware(RunnableTaskClass taskToSend) {  
   /\* workflow deste método  
   \* se o kernel n estiver shuttindown;  
   \* adiciona a task recebida à queue  
   \*/  
   if (!getIsShuttingDown()) {  
   tasksToExecute.add(taskToSend);  
   System.out.println("Fui recebida do middleware e estou no kernel agora");  
   } else {  
   System.out.println("Kernel is shutting down, not accepting tasks from middleware.");  
   }  
  }  
    
  public synchronized void sendTaskToMiddleware(RunnableTaskClass task) {  
   /\* Recebe task do cpu, e envia para o middleware tasksDoneExecuting buffer  
   \* \*/  
   /\*workflow do método  
   \* se a tasksToExecute tiver a task, remover pq já foi executada e só é suposto ser executada uma vez; // sim  
   \* remove a primeira tasks no tasksToExecute; // sim  
   \* se a task n for nula ENVIAR PARA O CPU // sim  
   \* remove a task do buffer kernel(?) - falta; // sim  
   \* chama o cpu e o método receiveTaskFromKernel - DONE: criar receiveTaskFromKernel  
   \* \*/  
    
   if(tasksToExecute.contains(task)){tasksToExecute.remove(task);}  
   RunnableTaskClass taskToReturnToCPU = tasksExecutedAndReadyToSendToMiddleware.removeFirst();  
   middleware.getmiddleWareRegistryOfCompletedTasks().add(task);  
   System.out.println("Task sent to middleware for further processing.");  
  }

> Os métodos sendTaskToMiddleware e receiveTaskFromMiddleware são usados para comunicar com o middleware pelo kernel. ReceiveTask() é o método responsável por retirar as tasks pedidas por parte do utilizador, antes das executar.   
O método sendTaskToMiddleware é usado quando uma task foi completa e precisa ser mandada de volta para o middleware.

public void turnOn() {  
 this.RAM\_Controlled\_By\_Kernel = new MEM();  
 this.CPU\_Controlled\_By\_Kernel = new CPU(this);  
 Thread cpuThread = new Thread(this.CPU\_Controlled\_By\_Kernel);  
 cpuThread.start();  
 this.isOn = true;  
}

* Método responsável por inicializar a unidade de memória e CPU;
* synchronized public void sendTaskToCPU() {  
   /\*workflow do método  
   \* se a tasksToExecute n está vazia; // sim  
   \* remove a primeira tasks no tasksToExecute; // sim  
   \* se a task n for nula ENVIAR PARA O CPU // sim  
   \* remove a task do buffer kernel(?) - falta; // sim  
   \* chama o cpu e o método receiveTaskFromKernel - DONE: criar receiveTaskFromKernel  
   \* \*/  
   if (!tasksToExecute.isEmpty()) {  
   RunnableTaskClass taskToReturnToCPU = tasksToExecute.remove(0);  
   if (taskToReturnToCPU != null) {  
   System.out.println("Fui removida do buffer de tasksToExecute do kernel");  
   this.CPU\_Controlled\_By\_Kernel.receiveTaskFromKernel(taskToReturnToCPU);  
   }  
   }  
  }
* SendTaskToCPU é o método responsável por enviar as tasks do kernel para o CPU, onde sim, serão então executadas dentro do roundRobin algoritmo.

**Classe MEM:**

public class MEM {  
  
  
 // This is a class that represents RAM, only the kernel knows of it's existence.  
  
 /\* represents the max size of RAM that the satellite has.  
 \*/  
 private final int RAM\_SPEC = 200;  
 /\* Tracks the current amount of memory being used at the present time.  
 \*/  
 private int usedRAM;  
 /\* tracks if RAM is full or not.  
 \*/  
 private boolean isfull(){  
 return usedRAM == RAM\_SPEC;  
 }  
  
 public MEM(){  
 this.usedRAM = 0;  
 }  
  
 public int getUsedRAM(){return usedRAM;}  
  
 public synchronized void alocateMemmory(int memmoryToAllocate) {  
  
 if(isfull() || usedRAM + memmoryToAllocate > RAM\_SPEC) {  
 throw new OutOfRAMException("there isn't enough RAM to perferm the task.");  
 }  
  
 this.usedRAM += memmoryToAllocate;  
  
 }  
  
 public synchronized void deallocateMemmory(int memmoryToDeallocate) {  
  
 if (usedRAM - memmoryToDeallocate < 0 ) {  
 throw new RAMnegativeNumbers("there's been an error, RAM can never be lower tan zero.");  
 }  
  
 this.usedRAM -= memmoryToDeallocate;  
 }  
}

Esta classe é relativamente simples, tem uma variável estática private final int RAM\_SPEC = 200;

* Que simula uma RAM limitada, e cada task tem um numero de ints associados e são retirados da RAM de acordo com o numero de tasks a correr.

**Classe CPU:**

public class CPU implements Runnable {  
  
  
 private final Kernel kernel;  
 private final Queue<RunnableTaskClass> taskQueue;  
 private int timeQuantum; // Time quantum for Round Robin  
  
 public CPU(Kernel kernel) {  
 this.kernel = kernel;  
 /\*In this code, RunnableTaskClass.setCPU(this) shows that  
 there's a class or method named RunnableTaskClass with a  
 static method setCPU that takes a cpu object as an argument.  
 This method is being called in the constructor of the cpu  
 class to inform all tasks about the CPU.  
  
 Understanding the Flow:  
  
 - When an instance of the cpu class is created, the constructor is called.  
 - The kernel field is initialized with the provided kernel parameter.  
 - The TaskMessage.setCPU(this) method is called, passing the current cpu object (this) as an argument.  
 - The setCPU method in the TaskMessage class stores a reference to the cpu object in the cpuReference variable.  
 \*/  
 this.taskQueue = new LinkedList<>();  
 this.timeQuantum = 2;  
 RunnableTaskClass.SetSatelliteCPU(this);  
 }

* taskQueue é o array de tasks a ser executadas pelo CPU, timeQuantum é o tempo permitido pelo CPU a cada task.
* @Override  
  public void run() {  
   while (kernel.getisOn() && !kernel.getIsShuttingDown()) {  
    
   if (!kernel.getTasksToExecuteFromKernel().isEmpty()) {  
   RunnableTaskClass taskForCycle = kernel.getTasksToExecuteFromKernel().remove(0);  
   if (taskForCycle != null) {  
   taskQueue.add(taskForCycle);  
   System.out.println("fui adicionada ao buffer do cpu");  
   }  
   }  
    
   // Execute tasks using Round Robin  
   while (!taskQueue.isEmpty()) {  
   RunnableTaskClass currentTask = taskQueue.remove();  
   System.out.println("fui removida da taskQueue do CPU");  
    
   // Check if shutting down after removing a task from the queue  
   if (kernel.getIsShuttingDown()) {  
   break;  
   }  
    
   kernel.AllocateMemmory(currentTask);  
   executeTask(currentTask);  
   System.out.println("o CPU tentou correr a task");  
    
   if (currentTask.getisDone()) {  
   kernel.DeallocateMemmory(currentTask);  
   System.out.println("dealoquei a memoria");  
   sendTaskToKernel(currentTask);  
   System.out.println("mandei para o kernel");  
   System.out.println(kernel.getTasksExecutedAndReadyToSendToMiddleware().size());  
   kernel.sendTaskToMiddleware(currentTask);  
   } else {  
   taskQueue.add(currentTask);  
   }  
    
   // Check if the time quantum has been reached  
   if (--timeQuantum == 0) {  
   break; // Move to the next task if the time quantum is reached  
   }  
   }  
    
   // Reset the time quantum for the next cycle  
   timeQuantum = 2;  
    
   // Process remaining tasks before shutting down  
   while (!taskQueue.isEmpty()) {  
   RunnableTaskClass currentTask = taskQueue.remove();  
   executeTask(currentTask);  
    
   if (currentTask.getisDone()) {  
   sendTaskToKernel(currentTask);  
   }  
   }  
    
   // Remove remaining tasks if the kernel wants to shut down  
   if (kernel.getIsShuttingDown()) {  
   taskQueue.clear();  
   }  
   }  
  }

Flow deste método run()  
  
flow of the code  
 if kernel is not shutting down and on  
 take task from the kernel's array of tasks  
 execute them for a certain time ( round robin )  
 are they done? if so, send them back up, if not send them back to waiting  
 check for new tasks and throw them if they exist.  
 if the kernel wants to shut down  
 finish tasks in the kernel that have been assigned to be executed,  
 but stop taking tasks from the middleware into the kernel.  
 after the tasks are done, stop executing the thread.

private void sendTaskToKernel(RunnableTaskClass taskToExecute) {  
 kernel.getTasksExecutedAndReadyToSendToMiddleware().add(taskToExecute);  
}  
  
/\*  
private RunnableTaskClass receiveTasksFromMiddleware() {  
 RunnableTaskClass taskToBeExecutedByCPU = kernel.getTasksToExecuteFromKernel().removeFirst();  
 return taskToBeExecutedByCPU;  
}\*/  
  
public void receiveTaskFromKernel(RunnableTaskClass taskToBeExecutedByCPU) {  
 /\* workflow deste método  
 \* se o kernel n estiver shuttindown;  
 \* adiciona a task recebida à queue  
 \*/  
 if (!kernel.getIsShuttingDown()) {  
 if (!taskQueue.contains(taskToBeExecutedByCPU)) {  
 taskQueue.add(taskToBeExecutedByCPU);  
 System.out.println("fui recebida pela Queue de tasks no cpu.");  
 }  
 }  
}

* sendTaskToKernel() – O método que adiciona tasks acabas para o kernel;
* consequencialmente o método receiveTaskFromKernel() faz exatamente isso;
* private void executeTask(RunnableTaskClass currentTask) {  
    
   try {  
   Thread.sleep(1000);  
   } catch (InterruptedException e) {  
   Thread.currentThread().interrupt();  
   e.printStackTrace();  
   }  
    
   currentTask.run();  
    
   try {  
   Thread.sleep(1000);  
   } catch (InterruptedException e) {  
   Thread.currentThread().interrupt();  
   e.printStackTrace();  
   }  
  }

* método responsável por correr as tasks.

**Classe RunnableTaskClass:**

public class RunnableTaskClass implements Runnable, Comparable<RunnableTaskClass>{  
  
 //ONLY THE KERNEL KNOWS OF THE CPU's EXISTENCE.  
  
 private Message messageInTask; // message to be changed in the run method.  
 private boolean isDone;  
 private TaskIdentifierFlags taskFunctionId; // flag that identifies the task for this task  
 private double memmory; // memmory that this task owns  
 private static CPU satelliteCPU; // cpu that all threads (tasks) share  
 private int priority;

* Esta classe representa as tasks em si ( as threads ) e têm uma variável Message que contém content e title, um boolean isDone para verificar se acabaram de correr, um TaskIdentifierFlag para distinguir que método deve correr quando a thread faz start(), uma memória associada a cada task para ser uma pelo RAM, e o satélite que é static porque todas as threads usam o mesmo CPU e priority que começa a 10 para todas, a descem sempre que o CPU adiciona mais uma task, simulando assim a criação de filas de prioridade para tasks.
* /\*\*  
   \* this method changes every character in a message's content to +2 value  
   \*/  
  private void encode() {  
   if (messageInTask.getContent() != null) {  
   char[] contentChars = messageInTask.getContent().toCharArray();  
   for (int i = 0; i < contentChars.length; i++) {  
   contentChars[i] += 2; // Shift each character by 2 (simple encoding)  
   }  
   messageInTask.setContent(new String(contentChars)); // Update the content in the original Message object  
   messageInTask.setTitle("secret message : ");  
   } else {System.out.println("there's no message to encode :(");}  
  }  
    
   /\*\*  
   \* this method changes every character in a message's content to -2 value  
   \*/  
   private void decode(){  
   if (messageInTask.getContent() != null) {  
   char[] contentChars = messageInTask.getContent().toCharArray();  
   for (int i = 0; i < contentChars.length; i++) {  
   contentChars[i] -= 2; // Shift each character by 2 (simple encoding)  
   }  
   messageInTask.setContent(new String(contentChars)); // Update the content in the original Message object  
    
   String result = "Encoded Message: " + messageInTask.getContent();  
   } else {System.out.println("No message set for encoding.");}  
   }  
    
   /\*\*  
   \* this method creates a location report and sets it as the content for the message of this task  
   \*/  
   private void locationCreation() {  
   // latitude and longitude are randomly generated for each task  
   double latitude = Math.random() \* 90; // Random latitude between -90 and 90  
   double longitude = Math.random() \* 180; // Random longitude between -180 and 180  
   String coordinates = new String("Current Location -> Latitude: " + latitude + ", Longitude : " + longitude + ".");  
   this.messageInTask.setContent(coordinates);  
   System.out.println(coordinates);  
   }  
    
   /\*\*  
   \* this method prints out a report for the satellite.  
   \* it'll present satelliteName, batteryLevel, SignalStrength, and if it's operational or not  
   \*/  
   private void satelliteReport() {  
   String satelliteName = "Satellite-SO101";  
   int batteryLevel = (int) (Math.floor(Math.random() \* (100 - 1 + 1) + 1 )); // Random battery level between 0 and 100  
   int signalStrength = (int) (Math.floor(Math.random() \* (100 - 50 + 1) + 50 )); // Random signal strength between 0 and 100  
   String isOperational = new String(" Satellite is operational. "); // Randomly set as operational or not  
   // Format the satellite report  
   String report = String.format("Satellite Report:\nName: %s\nBattery Level: %d%%\nSignal Strength: %d%%\nOperational: %s",  
   satelliteName, batteryLevel, signalStrength, isOperational);  
   // Print the report  
   System.out.println(report);  
   }
* Estas são as 4 funcionalidades diferentes oferecidas pelo sistema;

@Override  
public void run() {  
 try {  
 Thread.sleep(500);  
 StringBuilder result = new StringBuilder();  
  
 switch (taskFunctionId) {  
 case ENCODE:  
 System.out.println("encoding the " + messageInTask.getTitle() + " message");  
 result.append("encoding the ").append(messageInTask.getTitle()).append(" message").append(System.lineSeparator());  
 encode();  
 result.append(messageInTask.getContent());  
 result.append("[").append(messageInTask.getTitle()).append("] has been encoded to : ").append(messageInTask.getContent()).append(System.lineSeparator());  
 //satellitecpu.taskDoneExecuting(this); // task done executing so goes up to cpu  
 break;  
 case DECODE:  
 System.out.println("encoding the " + messageInTask.getTitle() + " message");  
 result.append("decoding the ").append(messageInTask.getTitle()).append(" message").append(System.lineSeparator());  
 decode();  
 result.append(messageInTask.getContent());  
 result.append("[").append(messageInTask.getTitle()).append("] has been decoded to : ").append(messageInTask.getContent()).append(System.lineSeparator());  
 System.out.println("[" + messageInTask.getTitle() + "] has been encoded to : " + messageInTask.getContent());  
 //satellitecpu.taskDoneExecuting(this); // task done executing so goes up to cpu  
 break;  
 case REPORT:  
 System.out.println("About to receive Satellite Report");  
 result.append("About to receive Satellite Report").append(System.lineSeparator());  
 satelliteReport();  
 result.append(messageInTask.getContent());  
 result.append("End of satellite report").append(System.lineSeparator());  
 System.out.println("End of satellite report");  
 //satellitecpu.taskDoneExecuting(this);  
 break;  
 case LOCATION:  
 System.out.println("About to print out satellite Location");  
 result.append("About to print out satellite Location").append(System.lineSeparator());  
 locationCreation();  
 result.append(messageInTask.getContent());  
 result.append("End of location report.").append(System.lineSeparator());  
 System.out.println("End of location report.");  
 //satellitecpu.taskDoneExecuting(this);  
 break;  
 default:  
 System.out.println("this task doesn't have an identifier associated");  
 break;  
 }  
  
 this.isDone = true;  
  
 // Write the result to a file  
 writeResultToFile(result.toString(), "Responses.txt");  
 } catch (InterruptedException e) {  
 Thread.currentThread().interrupt();  
 e.printStackTrace();  
 }  
}

* Este é o método run() para as tasks, utiliza um switch case com a variável taskIdentifierFlag para saber que tipo de task lhe está atribuído e executa o mesmo.
* private void writeResultToFile(String result, String fileName) {  
   try (BufferedWriter writer = new BufferedWriter(new FileWriter(System.getProperty("user.home") + "/Desktop/" + fileName))) {  
   writer.write(result);  
   writer.newLine();  
   } catch (IOException e) {  
   e.printStackTrace();  
   }  
  }  
    
    
  public void decreasePriority() {  
   this.priority--;  
  }
* writeResultToFile() é responsável por escrever o resultado para ficheiros como pedido nas funcionalidades extra, e decreasepriority() é usado para decrementar a prioridade de uma task.

**Conclusão :**

**.** O trabalho foi relativamente rápido de se completar, por volta de 20 horas.   
Apareceram vários problemas ao longo da codificação, maioritariamente no que tocava a race conditions entre classes, quando uma task era removida de um array, entrava em ordem errada noutro e outras situações assim.   
Não tivemos problemas de starvation por causa da simplicidade do programa e do algoritmo usado pelo CPU sendo o round robin.

Acreditamos que o trabalho esteja bem feito, não criámos interfaces gráficas devido a constrangimentos de tempo.