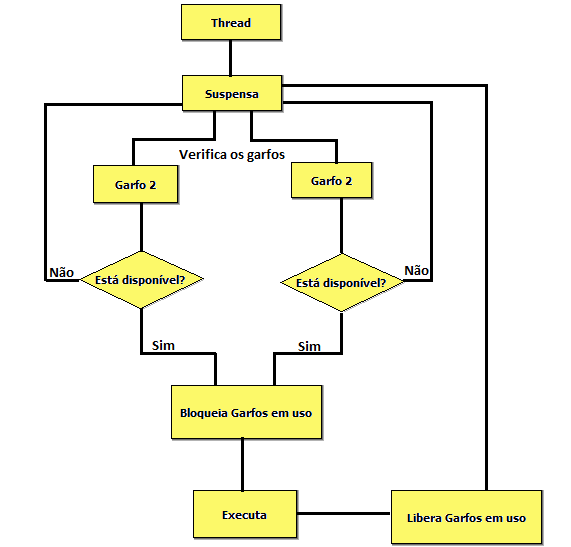
1. **O Problema dos Filósofos**

O problema consiste em que cada filósofo precisa utilizar dois garfos para comer, e não pode pegar um garfo que já esteja em uso por outro colega.  
  
 Abstraindo essa idéia, podemos dizer que, neste caso, 5 threads precisam ser executadas, compartilhando determinada informação entre si, não podendo ser executadas, a menos que estas informações estejam disponíveis.

Neste caso, cada thread precisa de duas informações distintas(garfos) para que possa entrar em execução, e não pode ser executada se estas duas informações não estejam disponíveis.

O diagrama básico de ação, válido para cada thread, poderia ser descrito como na **Imagem 1**, logo abaixo.

  
***Imagem 1. : Diagrama de interação e execução das threads.***

Explicando a idéia do diagrama, cada thread fica “suspensa”, ou no conceito do problema, “pensando”, enquanto os dados não estiverem disponíveis para que esta execute sua tarefa. Uma vez que ambos os dados estejam disponíveis, ela bloqueia os mesmos, definindo que esta está em uso por ela, impedindo que outra thread possa acessar os dados, até que ela termine seu processo. Uma vez que o precesso seja concluído, ela libera o acesso aos dados que esta estava utilizando, permitindo que outra thread possa acessá-los, voltando ao ponto de partida, reiniciando o clico, até que este seja interrompido pelo programa.

**O Algoritmo**

O algoritmo do progrma pode ser descrito, basicamente, nos seguintes passos:

1. Está suspenso/pensando
2. A cada intervalo de tempo, tenta entrar na função de verificação dos garfos disponíveis
3. Se não conseguir entrar na função, é porque esta está em uso por outra thread, então volta ao passo 1.
4. Se entrar na função, bloqueia esta para outras threads, até que esta conclua a mesma.

4.1 Dentro da função, verifica se os garfos estão disponíveis.   
 Se estiverem, define estes como em uso por ela, e sai da função.  
 Se os garfos não estiverem disponíveis, não faz nada, e sai da função.

**5.** Libera a função para que outra thread possa utilizá-la.  
 Se os garfos não estavam disponíveis, volta ao passo 1.

**6.** Com os garfos disponíveis, e definidos como em uso por ela,   
 passa a executar a função(comer), pelo tempo definido.

**7.** Terminado o processo de execução, libera os garfos em uso, para que   
 outras threads possam utilizá-los nos passos 4 e 4.1

**8.** Volta ao passo 1 para iniciar novamente o ciclo.

O algoritmo é bastante simples, sem muitas complexidades em termos de decisões a serem tomadas. Se os dados necessários estiverem disponíveis, pega os mesmos e executa, do contrário, volta a fazer outro processo, repetindo até que os dados necessários estejam disponíveis.

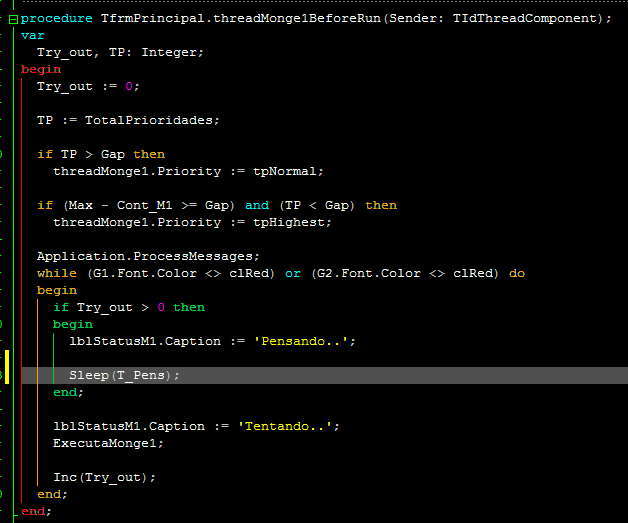
**O Programa explicado em código**

O programa foi feito em Delphi, mais especificemente na versão Xe7, que também possui suporte nativo à threads.

Primeiramente, foi criado uma variável chamada ON\_OFF, do tipo booleana.  
ON_OFF.png

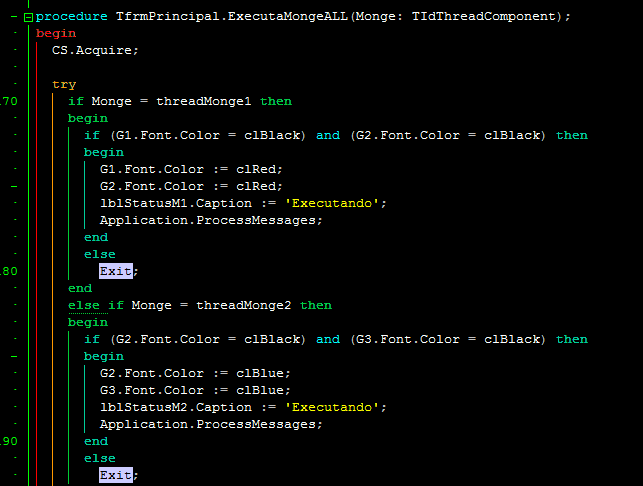
Essa variável é responsável por controlar a execução do programa. Enquanto **True**, o programa vai continuar o ciclo de execução, e todas as threads vão continuar executando.  
Quando desligada (**False**), a execução para, e as threads encerram a execução.

Antes de inicar a execução da tarefa principal, é necessário que cada thread verifique se seus respectivos garfos estão disponíveis.



No caso de os garfos não estarem disponíveis, as threads ficam suspensas, ou “Pensam”. No caso desse exemplo, foi utilizado a função ***Sleep();*** , mas poderia ser executado qualquer outra tarefa que não dependesse dos garfos. O tempo em que este fica pensando, ou na função ***Sleep()***, é configurado como padrão em 1 segundo, mas pode ser alterado na tela de configurações.  
 A variável ***Gap*** é definida como o limite tolerável entre o máximo ciclo, e o mínimo das threads, ou seja, o limite que o monge que mais comeu pode ficar, quando comparado com o monge que menos comeu. Essa variável é definida por padrão com o valor 3, mas é configurável através da tela de configurações.  
 Já ***TP*** é uma variável que retorna o número de threads com alta prioridade. É possível que, quando mais de uma thread esteja fora do limite do ***Gap***, mais de uma thread entre com máxima prioridade. Quando muitas threads entram com alta prioridade, pode-se dizer que a execução volta a estaca zero, então nesse caso, a procedure ***TotalPrioridades*** retorna o número de threads com alta prioridade. Se o número de prioridades for maior que o valor do ***Gap***, então as prioridades voltam ao normal, deixando o controle para a execução do ***Timer***.

A execução da thread chama a procedure ***ExecutaMongeALL***, que recebe a própria thread como parâmetro, e dentro desta função, baseada em qual thread chamou a mesma, verifica se os garfos estão disponíveis.



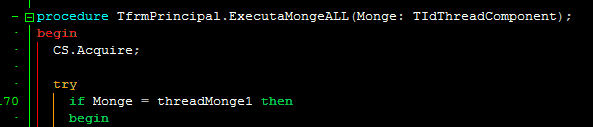
Para este exemplo, foi utilizado como parâmetro de uso dos garfos, a cor da fonte dos mesmos. Caso os garfos estejam disponíveis, tranca os mesmos para si (através de atualização da cor da fonte), e sai da função. Do contrário, simplesmente sai da função, e fica suspensa por um tempo determinado.

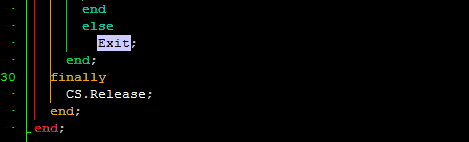
**Problema: Duas threads acessando a mesma função**

Neste ponto, nos deparamos com o primeiro problema. Sem um controle específico, duas threads podem entrar na mesma função ao mesmo tempo, e consequentemente acessar os mesmo dados ao mesmo tempo.   
 Para contornar esta situação, foi necessário criar uma seção crítica para a função. No delphi, é possível importar a classe SyncObjects (***System.SyncObjs***) para o projeto, e a partir dela, é possível instanciar variáveis do tipo ***TcriticalSection***.

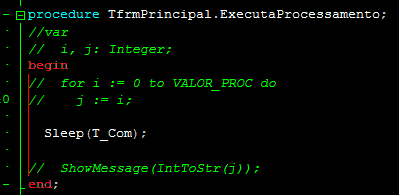
Sendo assim, foi criada uma variável do tipo ***TcriticalSection*** para controlar o acesso à função ***ExecutaMongeALL***.  
Critical.png

A variável do tipo TcriticalSection, permite dois métodos: ***Acquire*** e ***Release***. O método acquire, quando no começõ de uma função, bloqueia o acesso da mesma por outras threads que não sejam a que esteja acessando ela no momento. E, o método Release, libera o acesso à função para outras threads.  
 Assim, um Acquire no começo da função tranca o acesso por demais threads, garantindo que somente uma thread por vez possa acessar os dados. E um Release ao fim da função, libera o acesso à procedure pelas demais threads, garantindo desta forma que somente uma única thread acesse e atualize os dados.

***Acquire no começo da função, bloqueando o acesso simultâneo de mais de uma thread.***

  
***Release no fim da procedure, permitindo novamente o acesso, após a atualização dos dados.***

Se os garfos estiverem disponíveis, a thread executa a função, que neste caso, também foi posto um tempo específico para comer, através da função ***Sleep();***.

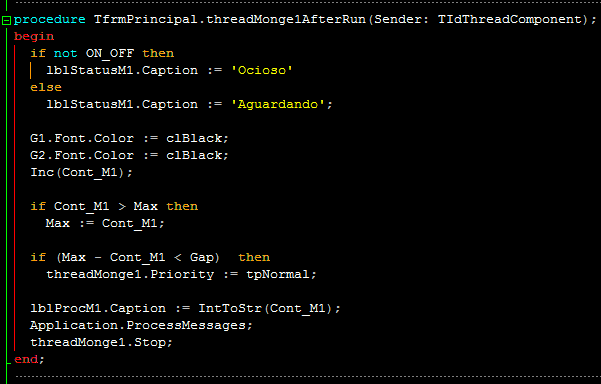


Da mesma forma que i tempo para “Pensar”, o tempo utilizado para “comer” é configurável através das configurações, mas está pre-definido em 4 segundos.

Os tempos são armazenados nas variáveis ***T\_Com*** (Tempo Comendo) e ***T\_Pens*** (Tempo Pensando).

Come e Pensa.png

Após o término da execução, cada thread devolve seus garfos, tornando os mesmos disponíveis (representado pela fonte preta) para que outra thread as use, e para o processo.



Além de liberar os garfos em uso, ao término da execução, ela atualiza o respectivo contador de ciclos, e a execução de maior valor (definido pela variável ***Max***), para controle de execução. Caso o número de execuções estejam dentro do ***Gap***, a prioridade passa a ser normal.

As threads param após a execução, sendo reiniciadas a cada 1,8 segundos por um ***Timer***, que verifica as threads que encerraram seus processos, reiniciando as mesmas, iniciando sempre o ciclo. Este timer reinicia as threads, somente se a execução do programa estiver ligada (variável ***ON\_OFF = True***).  
 Além disso, antes de iniciar a thread, o Timer compara o contador de cada thread com a thread com o número mínimo de execuções (através da função ***VerificarMinimo***). Caso esse valor seja maior que o ***Gap***, a thread permanece suspensa, ou “pensando”, até que todas as threads estejam dentro do limite definido.

O controle de execuções através de um limite estabelecido, é um controle para evitar o starvation das threads, garantindo que nenhuma thread fique sem ser executada or muito tempo.

Os botões ***Start*** e ***Stop***, inciam e param a execução do programa. Start liga a variável ***ON\_OFF*** (**True**) e inicia o processo das threads, enquanto que Stop desliga a variável ***ON\_OFF*** (**False)**, e para os processos das threads.

A tela de configuração dos tempos para Comer e Pensar, pode ser acessado pressionando as teclas alt+F12.

**Exemplo de uso do problema**

**Conclusão do Problema**

O trabalho foi relativamente simples de realizar, dados os conhecimentos na linguagem, dando relativo trabalho ao lidar com o acesso múltiplo das threads à mesma procedure, e consequentemente aos mesmo dados. Foi necessário estudar melhor o funcionamento das seções críticas, que foram indispensáveis para a realização da tarefa.  
 Uma dificuldade que encontramos, foi trabalhar justamente com as múltiplas execuções. Mesmo o controle de prioridades das threads não garantiu um resultado satisfatório em se tratando de starvation das threads, sendo necessário criar um controle manual, explicado em tópico anterior.

O código fonte completo seguem em anexo junto ao relatório.