

UNIVERSIDADE FEDERAL DE MINAS GERAIS  
DEPARTAMENTO DE ENGENHARIA ELETRÔNICA

**AUTOMAÇÃO EM TEMPO REAL**  
**TRABALHO PRÁTICO - PARTE 1**

Leiza Souza - 2017102100

Gabriel Lara - 2017088182

BELO HORIZONTE

2021

## **SUMÁRIO**

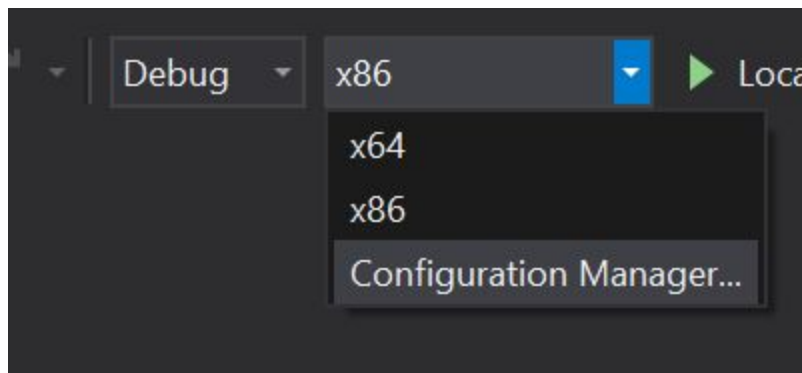
TAREFA DE LEITURA DO SISTEMA DE MEDIÇÃO	<b>4</b>
TAREFA DE LEITURA DE DADOS DO PROCESSO	<b>4</b>
TAREFA DE CAPTURA DE MENSAGENS	<b>4</b>
TAREFA DE EXIBIÇÃO DE DADOS DO PROCESSO	<b>4</b>
TAREFA DE ANÁLISE DE GRANULOMETRIA	<b>4</b>
TAREFA DE LEITURA DO TECLADO	<b>4</b>
PROCESSOS	<b>5</b>
THREADS	<b>6</b>
DADOS COMPARTILHADOS	<b>6</b>
SINCRONIZAÇÃO	<b>7</b>
MECANISMOS DE TEMPORIZAÇÃO	<b>10</b>
ACESSO A LISTAS CIRCULARES	<b>11</b>
COMPILAÇÃO	<b>14</b>
DEPURAÇÃO	<b>14</b>

*Nota sobre execução: A solução do Visual Studio está dividida em três projetos.*

*A abertura da solução no Visual Studio deve ser feita carregando o arquivo "projeto/granulometria/granulometria.sln"*

*Dessa forma os outros projetos são carregados da maneira correta.*

*Verifique que ao lado de "Debug" está selecionado "x86":*



*Em seguida compila-se o programa com a função build.*

## **1. INTRODUÇÃO**

O aço produzido nas siderúrgicas compõe vários materiais que estão presentes no dia a dia da população, sejam eles eletrodomésticos, veículos de locomoção, construções civis e até mesmo utensílios. Para obtê-lo, é necessário que o minério de ferro passe por um processo térmico com objetivo de transformá-lo em pellets, ou pelotas, o que atribui o nome à tecnologia: Pelotização. O processo surgiu no século XX e tem como produto pequenas esferas férreas com diâmetro entre 8mm e 18mm, após a transformação do minério nas etapas de moagem, espessamento, homogeneização, filtragem, pelotamento e queima, respectivamente.



Figura 1 - Comparação entre minério granulado e pelotas de ferro [1]

Nos altos-fornos das usinas siderúrgicas os pellets são transformados em ferro-gusa e, para que isso ocorra, é necessário que apresentem características que agreguem qualidade ao produto, sendo um dos principais indicativos de qualidade a granulometria: as pelotas devem ser pequenas, porém devem possuir tamanho suficiente para que haja lacunas entre elas, facilitando a circulação de ar e aumentando a superfície de contato. Para avaliar a granulometria de pelotas cruas, será desenvolvido um sistema a ser implantado nos discos de pelotização de uma usina, responsável por ler os dados do sistema de medição de granulometria e os dados do Controlador Lógico Programável (CLP) e apresentá-los em terminais dedicados, com o objetivo de automatizar o controle e a supervisão do processo. Os dados provenientes da aplicação de software serão analisados pelos operadores de processo no primeiro terminal e por especialistas de pelotização no segundo terminal, voltado à granulometria.

## 2. ESTRUTURA

A aplicação *multithread* desenvolvida no Ambiente de Desenvolvimento Integrado (IDE) *Microsoft Visual Studio Community Edition*, na linguagem C/C++, faz o uso API Win32 para garantir maior eficiência e facilidade de desenvolvimento, por se tratar de uma API nativa do Windows, e foi desenvolvida baseada nas seguintes tarefas:

### 2.1. TAREFA DE LEITURA DO SISTEMA DE MEDIÇÃO

Responsável por ler as mensagens originadas do sistema de medição online de granulometria das pelotas produzidas pela usina de mineração e depositá-las em uma primeira lista circular de memória RAM, que possui capacidade de 200 mensagens.

## 2.2. TAREFA DE LEITURA DE DADOS DO PROCESSO

Responsável por ler as mensagens originadas do CLP e depositá-las na mesma lista circular citada anteriormente.

## 2.3. TAREFA DE CAPTURA DE MENSAGENS

Responsável por consumir as mensagens da primeira lista circular de memória e enviá-las para a *tarefa de exibição de dados de processo*, caso sejam provenientes do CLP, ou depositá-las em uma segunda lista circular em memória, com capacidade de 100 mensagens, caso sejam provenientes do sistema de medição granulometria.

## 2.4. TAREFA DE EXIBIÇÃO DE DADOS DO PROCESSO

Responsável por receber as mensagens de dados de processo do CLP através da *tarefa de captura de mensagens* e exibi-las em uma tela dedicada na sala de controle.

## 2.5. TAREFA DE ANÁLISE DE GRANULOMETRIA

Responsável por consumir da segunda lista circular em memória as mensagens provenientes do sistema de medição de granulometria e exibi-las em uma segunda tela dedicada na sala de controle.

## 2.6. TAREFA DE LEITURA DO TECLADO

Responsável por aguardar comandos do operador (caracteres no teclado) e tratá-los, de acordo com o estado anterior de cada tarefa, bloqueando-a ou desbloqueando-a por meio dos seguintes objetos “evento” de sincronização do kernel:

- 2.6.1. <g>: referente à *tarefa de leitura do sistema de medição*;
- 2.6.2. <c>: referente à *tarefa de leitura de dados do processo*;
- 2.6.3. <r>: referente à *tarefa de captura de mensagens*;
- 2.6.4. <p>: referente à *tarefa de exibição de dados de processo*;
- 2.6.5. <a>: referente à *tarefa de análise de granulometria*;
- 2.6.6. <l>: responsável por notificar à *tarefa de exibição de dados de processo* que deve limpar a janela de console;
- 2.6.7. <ESC>: responsável por notificar todas as tarefas que devem encerrar a execução.

As tarefas indicadas nos itens 2.1, 2.2 e 2.3 devem se bloquear quando as respectivas listas em memória estiverem cheias e se desbloquear quando houver uma nova posição livre.

## 2.7. PROCESSOS

Um processo pode ser definido como um programa ou uma atividade em execução que possui um conjunto de recursos exclusivos gerenciados pelo Sistema Operacional, como registradores e espaços de endereçamento virtual, e é composto por diversos objetos relacionados à sua execução. Sempre que um processo cria um objeto do *kernel*, o S.O. retorna um *handle* através do qual é possível manipulá-lo. Os estados manipuláveis e possíveis nesta aplicação serão:

### 2.7.1. Pronto para executar

Aguardando a sinalização de um objeto do *kernel*, no caso um mutex, para possuir a CPU:

*WaitForSingleObject(sem\_livre, timeout)*

### 2.7.2. Executando

Possui a CPU durante um intervalo de tempo, após a sinalização do mutex.

### 2.7.3. Bloqueado

Aguardando a ocorrência de um evento de E/S para continuar, no caso a leitura de duas teclas (ESC ou o respectivo caractere sinalizador) ou a liberação de posição na lista circular:

*WaitForMultipleObjects(2, Events, FALSE, INFINITE)*

*WaitForMultipleObjects(2, buffer\_block\_objects, FALSE, INFINITE)*

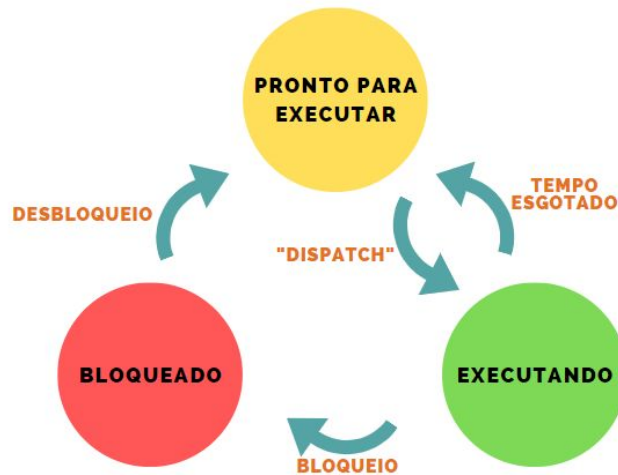


Figura 2 - Representação de estados

## 2.8. THREADS

Threads são linhas de execução que compartilham todos recursos em comum do seu processo criador, promovendo simplicidade e rapidez de tratamento. Além disso, podem possuir os mesmo estados representados na Figura 2.



Figura 3 - Representação de processos e threads da aplicação

## 2.9. DADOS COMPARTILHADOS

A primeira lista circular em memória é compartilhada pelas tarefas de *leitura do sistema de medição*, de *leitura de dados de processo* e de *captura de mensagens*, enquanto a segunda lista é compartilhada pelas tarefas de *captura de mensagens* e de *análise de*

*granulometria*. Para acessar a lista, os *handles* dos semáforos de sincronização também são compartilhados entre as *threads*:

*HANDLE sem\_livre;*

*HANDLE sem\_ocupado;*

*HANDLE sem\_rw;*

Para encerrar o processo, outro *handle* é compartilhado (nesse caso, ele é sinalizado quando a tecla Esc é pressionada):

*HANDLE end\_event;*

Nota-se que o compartilhamento inter processo dos eventos é dado por meio da abertura do *handle* correspondente no processo secundário por meio da função *OpenHandle*. Para tal, os eventos compartilhados são nomeados.

### **3. CONTROLE**

O controle da execução e acesso aos recursos compartilhados das tarefas foi feito por meio de objetos de sincronização que se resumem a semáforos contadores, semáforos binários e eventos. Optou-se por usar mecanismos simples e confiáveis para garantir boa interpretabilidade e desempenho. Observa-se que pela multiplicidade de maneiras possíveis e experimentadas de atender as especificações do projeto as soluções propostas foram semelhantes entre si, baseadas no mecanismo de *timeout* das funções de espera por objetos. Uma breve discussão sobre a motivação dessa escolha é travada na seção 5. Documentação e análise desse tipo de método foi encontrada no material “*Intel Guide For Developing Multithreaded Applications*” [4], principalmente as seções “*Choosing appropriate synchronization primitives to minimize overhead*” e “*Use non blocking locks when possible*”.



### 3.1. SINCRONIZAÇÃO

O problema de sincronização enfrentado na primeira parte do trabalho se resume a coordenar a produção e consumo de informações (por enquanto arbitrárias) na primeira lista circular em memória. Nessa interação há dois produtores (tarefa de leitura do sistema de medição e tarefa de leitura de dados do processo) e um consumidor (tarefa de captura de mensagens). As seguintes subseções descrevem por meio de quais objetos e como seu comportamento foi controlado. Nota-se que a solução descrita é muito semelhante à uma das que foram implementadas para o problema de produtores e consumidores no material didático [5].

#### 3.1.1. Objetos

A função de cada objeto, assim como o nome de sua variável e os valores de inicialização, serão listados nesta subseção. O processo usou duas variáveis globais responsáveis por armazenar a próxima posição livre para depósito de informações por parte dos dois produtores e a próxima posição ocupada para retirada de informações por parte do consumidor:

*int p\_livre = 0;*

*int p\_ocupado = 0;*

Os valores foram assim atribuídos por que inicialmente não há dados no buffer, portanto a próxima posição livre e ocupada é a primeira. Note que a inicialização dos semáforos foi documentada de forma simplificada, omitindo a declaração dos *handles* correspondentes.

Para controle do processo de escrita e leitura foi usado:

- Um semáforo binário, agindo efetivamente como mutex, para permitir apenas um processo a executar sua ação desejada por vez:

*sem\_rw = CreateSemaphore(NULL, 1, 1, NULL);*

O valor máximo do semáforo é 1 porque ele é binário e seu valor inicial é 1 porque a primeira *thread* que desejar escrever no banco deve conseguir (caso contrário ocorreria um *deadlock*);

- Um par de semáforos contadores para controlar a população de informações na lista, impedir que informações sejam sobrescritas ou lidas de maneira repetida e garantir que só sejam consumidas ou depositadas informações quando for necessário. Ao produzir um dado os produtores consomem uma sinalização do semáforo *sem\_livre*, efetivamente comunicando que o número das posições livres para depósito reduziu em 1. De maneira exatamente análoga, o único consumidor da lista, ao consumir um dado, consome uma sinalização do semáforo *sem\_ocupado* comunicando que o número de posições ocupadas, ou em que se encontram dados prontos para leitura, reduziu em 1:

*sem\_livre = CreateSemaphore(NULL, buffer\_size, buffer\_size, NULL);*

*sem\_ocupado = CreateSemaphore(NULL, 0, buffer\_size, NULL);*

Ambos semáforos devem ter valor máximo correspondente ao tamanho do *buffer* porque é possível que ocorram consecutivos depósitos ou consumos até o número de algum desses eventos se igualar a esse valor. O valor inicial de cada um é diferente, no entanto; como o *buffer* é inicializado sem dados, o número inicial de posições livres é igual ao seu tamanho e não há posições ocupadas.

### 3.1.2. Métodos

Semáforos são objetos de sincronização do *kernel* relacionados ao sincronismo entre *threads* e à comunicação entre processos, que implementam uma fila do tipo “*First In, First Out*”, associada a um contador, e são executados de forma atômica. O contador, por sua vez, indica o número de instâncias livres do recurso:

- *contador = 0*: podem existir N *threads* aguardando na fila para tomarem posse do semáforo;
- *contador > 1*: a fila está vazia.

Sendo assim, quando uma *thread* instancia um semáforo (correspondente a *Wait*), seu valor de contagem é decrementado em 1. Analogamente, quando uma *thread* libera o recurso instanciado (correspondente a *Signal*), o valor de contagem é incrementado em 1. Vale ressaltar que semáforos binários possuem valor máximo igual a 1.

Ao ser criado, o S.O. retorna um *handle* através do qual é possível manipular o semáforo. No Windows, a operação de Wait é empregada por *WaitForSingleObject*, que decrementa o valor de contagem ou bloqueia a *thread* que a chama quando for igual a 0, e a operação de Signal por *ReleaseSemaphore*, que acorda a próxima *thread* da fila ou incrementa o valor de contagem. Com isso, é possível perceber que, salvo casos de exclusão mútua, é possível que um semáforo seja liberado por uma *thread* distinta da que o possuiu em primeiro lugar.

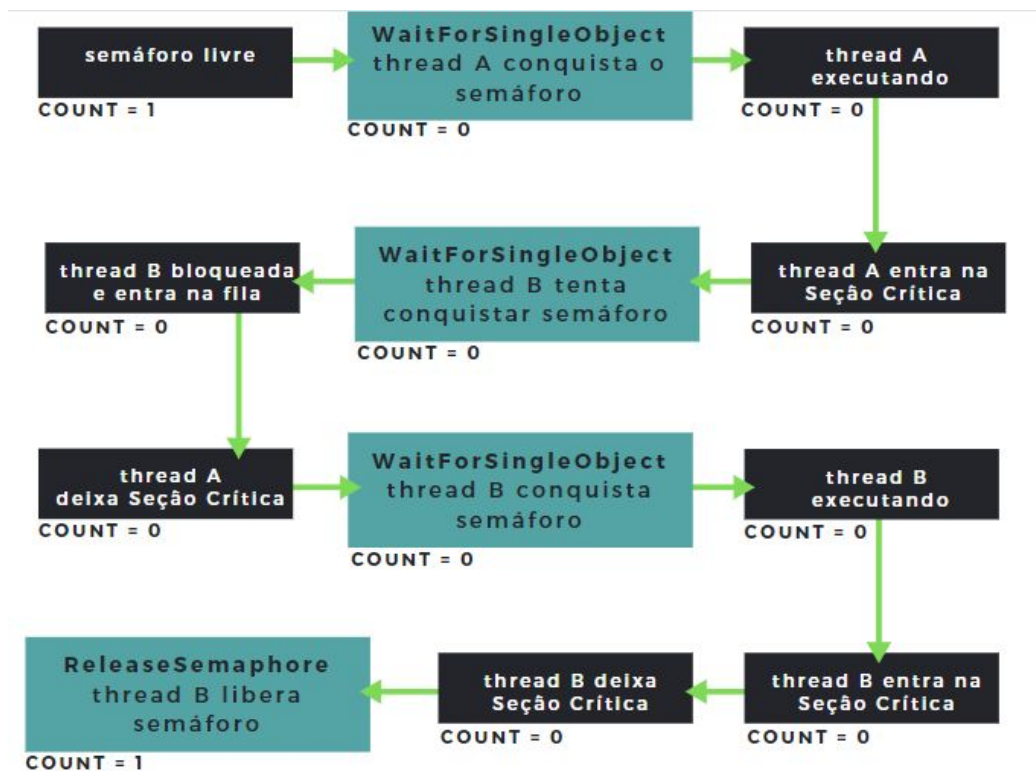


Figura 4 - Exemplo de operação: semáforo binário

## 4. TEMPORIZAÇÃO

### 4.1. MECANISMOS DE TEMPORIZAÇÃO

A temporização das tarefas, na primeira parte do trabalho, é realizada simplesmente pela função *Sleep*, fazendo com que a cada etapa de execução das tarefas a *thread* correspondente pause por 1 segundo, como demandando no enunciado.

## 5. ESPECIFICAÇÕES DE PROJETO

Essa seção aborda a maneira que as especificações de projeto foram atendidas, sendo elas o bloqueio de tarefas produtoras ao tentar depositar dados no *buffer* se ele estiver cheio, o bloqueio e desbloqueio de cada tarefa e o encerramento simultâneo de todas as tarefas por um mesmo sinal. Optou-se por utilizar ao máximo as propriedades de temporização das funções de *WaitForSingleObjects* e *WaitForMultipleObjects*, explorando as possibilidades de controle com o mecanismo de *timeout*, pelos seguintes motivos:

- Como as funções de espera por sinalização são usadas de qualquer maneira, está no interesse de simplicidade e eficiência usá-las para temporização também;
- O mecanismo usado é essencialmente o mesmo, tornando a compreensão do código como um todo mais simples;
- A temporização resultante é mais flexível por ser essencialmente parametrizável em tempo de execução;
- A natureza *soft real time* da aplicação permite que a granularidade e precisão da temporização seja a natural do sistema operacional.

### 5.1. ACESSO A LISTAS CIRCULARES

O acesso dos produtores ao *buffer* circular é mediado, necessariamente, pelo semáforo binário *sem\_livre* (subseção 3.1.1). Na API WIN32 a espera pela sinalização desse semáforo é feita pela função *WaitForSingleObject(x, y)* onde *x* corresponde ao objeto cuja sinalização é aguardada e *y* ao tempo da espera antes da função desistir e retornar um código de erro, sinalizando que o tempo de espera se esgotou. Propõe-se explorar a seguinte propriedade da produção de dados analisada: há dois estados possíveis para o contador do semáforo *sem\_livre* (que indica o número de posições livres no buffer), 0 e diferente de 0. Se o contador for igual a zero, o buffer está necessariamente cheio e o produtor que faz a tentativa de depósito vai esperar por um intervalo de tempo não nulo. Se o contador for diferente de zero, há espaço na lista e a *thread* não aguarda por tempo algum; a função *WaitForSingleObject* simplesmente

consome uma sinalização e a *thread* prossegue com sua execução. Conclui-se portanto que se o tempo de espera da função *WaitForSingleObject* for desprezível para a execução do programa como um todo, o sinal de *timeout* pode ser usado para indicar que o *buffer* estava cheio no momento da chamada da função, porque caso contrário não ocorreria espera alguma.

A *thread* executa uma tentativa de acesso à lista da seguinte maneira:

```
ret = WaitForSingleObject(sem_livre, timeout);
```

Em que o valor de *timeout* é 100 ms. Se o retorno da função informar que ocorreu o *timeout*, isso é, que não haviam posições imediatamente livres no *buffer*, conclui-se que o *buffer* estava cheio no momento da chamada.

No caso em que o *buffer* está cheio a *thread* executa uma chamada de *WaitForMultipleObjects* com tempo de espera infinito aguardando sinalização do evento de término ou do semáforo *sem\_livre* para prosseguir com sua execução, cada caso sendo tratado da maneira apropriada.

No caso em que o *buffer* possui posições livres a *thread* “passa reto” da chamada e deposita o dado na posição adequada.

## 5.2. BLOQUEIO E DESBLOQUEIO DAS THREADS E ENCERRAMENTO

O bloqueio e desbloqueio assim como o encerramento de todas as tarefas é realizada da mesma maneira, semelhante em conceito ao que foi discutido na subseção anterior. Inicializa-se um par de eventos:

```
end_event = CreateEvent(NULL, TRUE, FALSE, TEXT("end_event"));
```

```
leitura_dados_toggle_event = CreateEvent(NULL, FALSE, FALSE, NULL);
```

Observa-se que o segundo desses eventos é criado para cada uma das tarefas, com o intuito de bloquear e desbloquear cada uma individualmente. Uma diferença relevante

entre os eventos é seu tipo, cujo evento de encerramento (primeira linha do exemplo acima) é do tipo *reset* manual justamente porque é desejado que sua sinalização encerre *todas* as tarefas, sendo necessário que o sinal de encerramento não seja consumido em sua recepção. Já o evento de bloqueio e desbloqueio deve ter sua sinalização consumida, uma vez que é desejado que o estado da *thread* receptora seja alterado. Além disso, os eventos que são compartilhados entre os três processos da aplicação (como o evento de encerramento) foram criados de maneira nomeada e os que foram usados apenas pelo processo principal (como o exemplificado acima) foram criados de maneira não nomeada.

Dessa maneira o controle do bloqueio é feito por meio da seguinte chamada:

```
ret = WaitForMultipleObjects(2, Events, FALSE, timeout);
```

Em que *Events* é um vetor que armazena ambos eventos comentados no parágrafo anterior. As *threads* tratam o retorno *ret* de forma a identificar e reagir aos três possíveis casos das seguintes maneiras:

- Se ocorrer *timeout*: Caso normal para a *thread*. Pula para o próximo laço de sua execução, continuando seu comportamento esperado;
- Se for recebido o sinal do evento de bloqueio a *thread* executa uma outra chamada de *WaitForMultipleObjects* que se diferencia da anterior por aguardar os sinais indefinidamente. Se a *thread* se despertar nesse ponto por sinalização do evento de bloqueio e desbloqueio a execução continua normalmente;
- Se for recebido o sinal de encerramento a *thread* escapa do laço infinito e encerra sua execução.

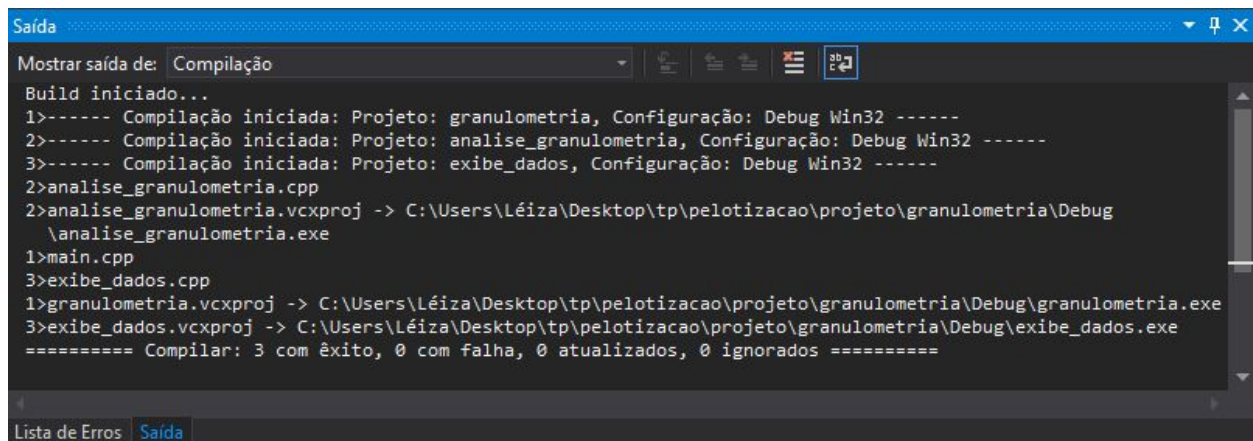
Dessa forma, a função *WaitForMultipleObjects* funciona mais como uma verificação do que uma espera de fato, conferindo se desde a última passagem da execução naquele ponto houve sinalização pelos objetos de interesse. Se não, continua-se como se nada tivesse ocorrido (de fato nada ocorreu!).

O encerramento é realizado pelo evento correspondente, cujo sinal é disparado pela tecla “Esc”. A única observação pertinente sobre seu comportamento é que foi necessário ter o cuidado de incluir uma observação desse evento e consequente reação à sua sinalização sempre que uma função de espera (*WaitFor\**) com tempo de espera infinito for chamada, para garantir que não ocorresse um *deadlock* resultante de uma *thread* aguardando por um sinal que não pode ser disparado devido ao encerramento da *thread* principal.

Recomenda-se que seja recorrido ao código e seus minuciosos comentários para a compreensão dos detalhes mais finos da lógica descrita

## 6. TESTES DE EXECUÇÃO

### 6.1. COMPILAÇÃO



```
Saída
Mostrar saída de: Compilação
Build iniciado...
1>----- Compilação iniciada: Projeto: granulometria, Configuração: Debug Win32 -----
2>----- Compilação iniciada: Projeto: analise_granulometria, Configuração: Debug Win32 -----
3>----- Compilação iniciada: Projeto: exibe_dados, Configuração: Debug Win32 -----
2>analise_granulometria.cpp
2>analise_granulometria.vcxproj -> C:\Users\Léiza\Desktop\tp\pelotizacao\projeto\granulometria\Debug
  \analise_granulometria.exe
1>main.cpp
3>exibe_dados.cpp
1>granulometria.vcxproj -> C:\Users\Léiza\Desktop\tp\pelotizacao\projeto\granulometria\Debug\granulometria.exe
3>exibe_dados.vcxproj -> C:\Users\Léiza\Desktop\tp\pelotizacao\projeto\granulometria\Debug\exibe_dados.exe
===== Compilar: 3 com êxito, 0 com falha, 0 atualizados, 0 ignorados =====
```

Figura 4 - Compilação

### 6.2. DEPURAÇÃO

#### 6.2.1. Inicialização dos processos

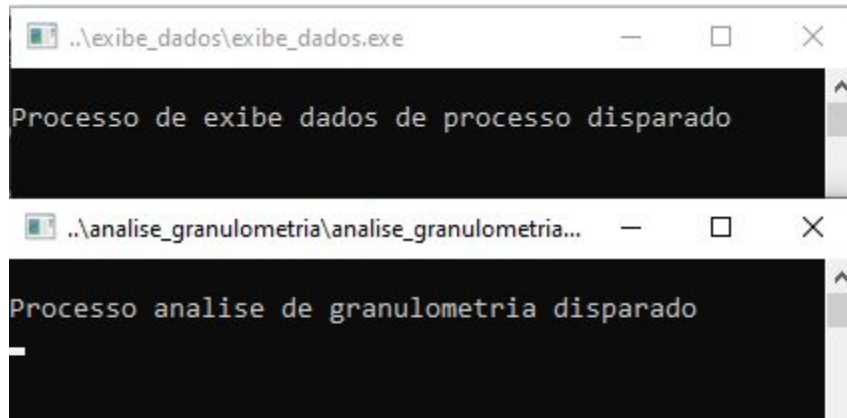


Figura 5 - Processos exibe\_dados e analise\_granulometria

```
Console de Depuração do Microsoft Visual Studio

thread leitura de medicao criada com id = 34e8
thread leitura dados criada com id = 3b0c
thread captura mensagens criada com id = 1258

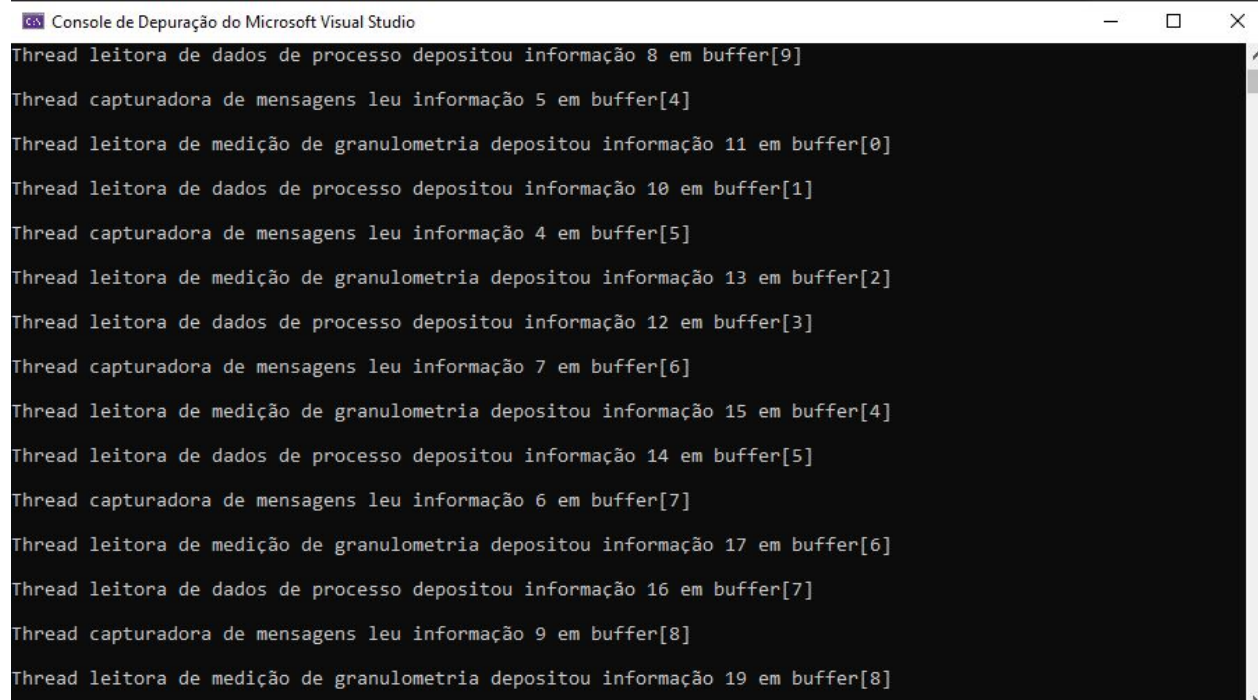
Thread leitora de medição de granulometria depositou informação 1 em buffer[0]

-----
SISTEMA DE CONTROLE E SUPERVISÃO
-----
Processo de pelotização e medição de granulometria
.....
Bloqueio/retomada da execução de tarefas:
Tecla g: Tarefa de leitura de dados de medição
Tecla c: Tarefa de leitura de dados de processo
Tecla r: Tarefa de captura de mensagens
Tecla p: Tarefa de exibição de dados de processo
Tecla a: Tarefa de análise de granulometria
.....
Para encerrar o processo: ESC

Thread leitora de dados de processo depositou informação 0 em buffer[1]
Thread capturadora de mensagens leu informação 1 em buffer[0]
Thread leitora de medição de granulometria depositou informação 3 em buffer[2]
Thread leitora de dados de processo depositou informação 2 em buffer[3]
Thread capturadora de mensagens leu informação 0 em buffer[1]
```

Figura 6 - Processo principal



The image shows a screenshot of the 'Console de Depuração do Microsoft Visual Studio' window. It contains a log of thread operations on a buffer. The threads are categorized into three types: 'Thread leitora de dados de processo' (process data reader), 'Thread capturadora de mensagens' (message capturer), and 'Thread leitora de medição de granulometria' (granulometry measurement reader). Each thread deposits or reads information from specific indices of a buffer (buffer[0] to buffer[9]). The sequence of operations demonstrates a circular buffer behavior where the next thread to read from index 9 wraps around to index 0.

```
Thread leitora de dados de processo depositou informação 8 em buffer[9]
Thread capturadora de mensagens leu informação 5 em buffer[4]
Thread leitora de medição de granulometria depositou informação 11 em buffer[0]
Thread leitora de dados de processo depositou informação 10 em buffer[1]
Thread capturadora de mensagens leu informação 4 em buffer[5]
Thread leitora de medição de granulometria depositou informação 13 em buffer[2]
Thread leitora de dados de processo depositou informação 12 em buffer[3]
Thread capturadora de mensagens leu informação 7 em buffer[6]
Thread leitora de medição de granulometria depositou informação 15 em buffer[4]
Thread leitora de dados de processo depositou informação 14 em buffer[5]
Thread capturadora de mensagens leu informação 6 em buffer[7]
Thread leitora de medição de granulometria depositou informação 17 em buffer[6]
Thread leitora de dados de processo depositou informação 16 em buffer[7]
Thread capturadora de mensagens leu informação 9 em buffer[8]
Thread leitora de medição de granulometria depositou informação 19 em buffer[8]
```

Figura 7 - Dinâmica das threads com o buffer

**Comentário:** Note na imagem acima que o *buffer* é acessado de forma circular. Para simplificar o teste, a capacidade estabelecida para o *buffer* foi igual a 10, sendo assim, após uma *thread* depositar informação na última posição [9], a próxima *thread* irá depositar na posição [0], caso ela esteja livre. Vale ressaltar que a lista circular em memória é implementada de forma FIFO, *First In First Out*, ou seja, a primeira mensagem depositada será a primeira consumida.

```
Console de Depuração do Microsoft Visual Studio

Thread leitora de medição de granulometria depositou informação 19 em buffer[8]
Thread capturadora de mensagens leu informação 8 em buffer[9]
Thread leitora de dados de processo depositou informação 18 em buffer[9]
*****
Capacidade máxima da primeira lista circular em memória atingida.
Thread de leitura do sistema de medição tentou depositar informação e está se bloqueando até livrar posição.
*****
Thread capturadora de mensagens leu informação 11 em buffer[0]
Thread leitora de medição de granulometria depositou informação 21 em buffer[0]
*****
Capacidade máxima da primeira lista circular em memória atingida.
Thread de leitura de dados do processo tentou depositar informação e está se bloqueando até livrar posição.
*****
Thread capturadora de mensagens leu informação 10 em buffer[1]
Thread leitora de dados de processo depositou informação 20 em buffer[1]
*****
Capacidade máxima da primeira lista circular em memória atingida.
Thread de leitura do sistema de medição tentou depositar informação e está se bloqueando até livrar posição.
*****
```

Figura 8 - Dinâmica das threads com o buffer: bloqueio

**Comentário:** Caso uma *thread* tente depositar no *buffer* e ele esteja cheio, ela se bloqueia até que uma posição seja liberada, como mostrado acima.

```
Console de Depuração do Microsoft Visual Studio

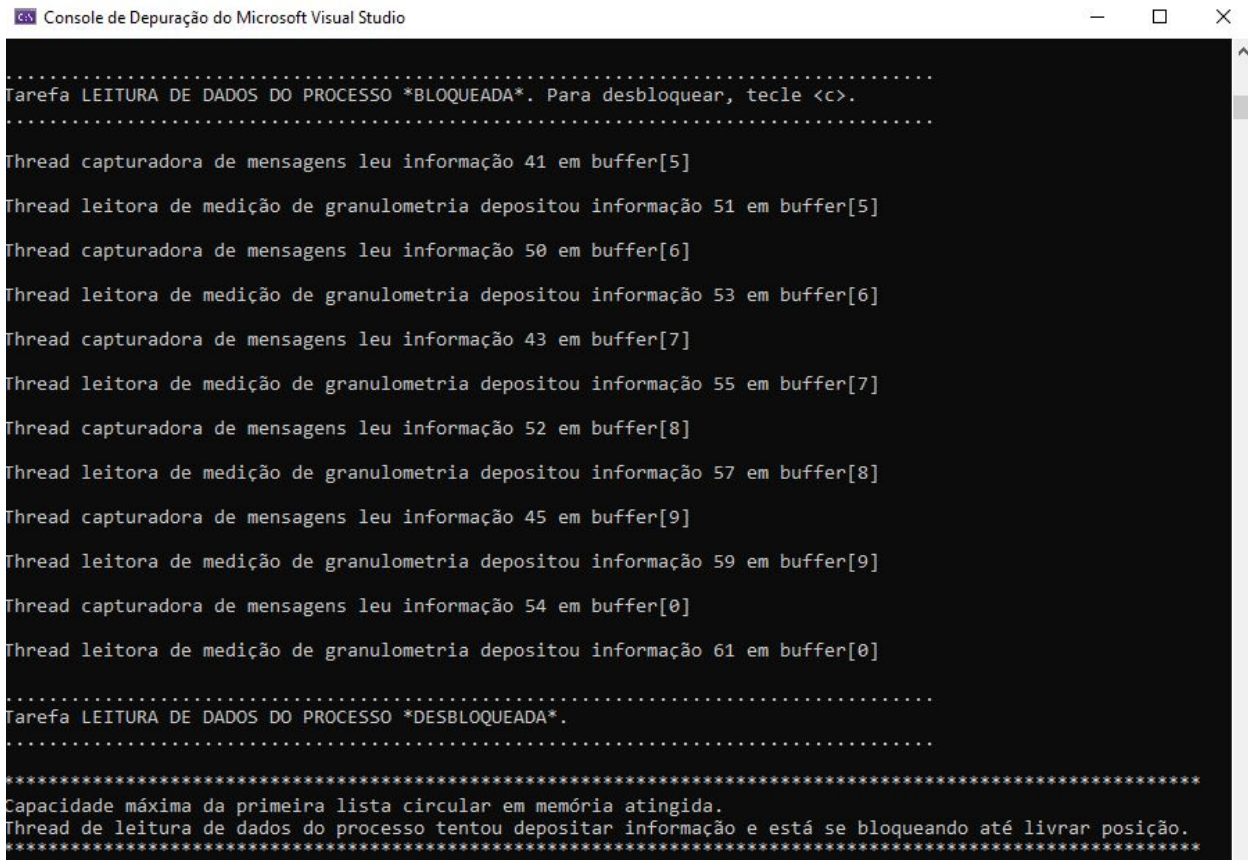
.....
Tarefa LEITURA DO SISTEMA DE MEDIÇÃO *BLOQUEADA*. Para desbloquear, tecle <g>.
.....

Thread capturadora de mensagens leu informação 14 em buffer[5]
Thread leitora de dados de processo depositou informação 24 em buffer[5]
Thread capturadora de mensagens leu informação 17 em buffer[6]
Thread leitora de dados de processo depositou informação 26 em buffer[6]
Thread capturadora de mensagens leu informação 16 em buffer[7]
Thread leitora de dados de processo depositou informação 28 em buffer[7]
Thread capturadora de mensagens leu informação 19 em buffer[8]
Thread leitora de dados de processo depositou informação 30 em buffer[8]
Thread capturadora de mensagens leu informação 18 em buffer[9]
Thread leitora de dados de processo depositou informação 32 em buffer[9]
Thread capturadora de mensagens leu informação 21 em buffer[0]
Thread leitora de dados de processo depositou informação 34 em buffer[0]
.....
Tarefa LEITURA DO SISTEMA DE MEDIÇÃO *DESBLOQUEADA*.
.....

*****
Capacidade máxima da primeira lista circular em memória atingida.
Thread de leitura do sistema de medição tentou depositar informação e está se bloqueando até livrar posição.
*****
```

Figura 9 - Bloqueio/Desbloqueio da tarefa de leitura de medição

**Comentário:** Quando apenas a *tarefa de leitura do sistema de medição* é bloqueada, a proporção de *threads* consumidoras e produtoras é igual a 1. Sendo assim, a taxa de mensagens sendo depositadas no buffer é igual à taxa de mensagens sendo retirada, não há bloqueio e a execução prossegue indefinidamente. Note que, assim que a tarefa foi desbloqueada, a primeira *thread* a tentar acessar a lista circular foi bloqueada.



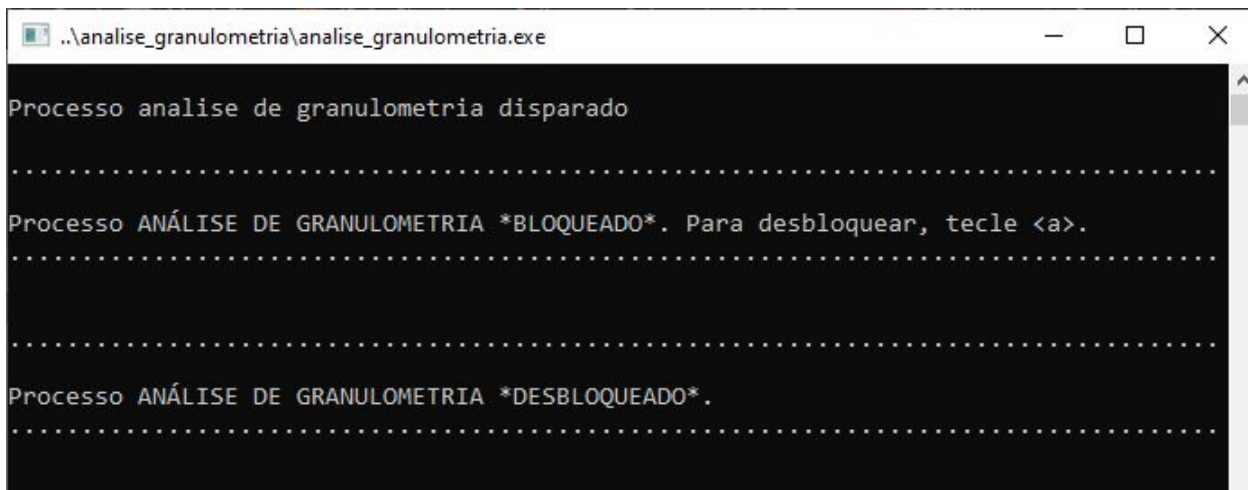
```
.....
Tarefa LEITURA DE DADOS DO PROCESSO *BLOQUEADA*. Para desbloquear, tecle <c>.
.....

Thread capturadora de mensagens leu informação 41 em buffer[5]
Thread leitora de medição de granulometria depositou informação 51 em buffer[5]
Thread capturadora de mensagens leu informação 50 em buffer[6]
Thread leitora de medição de granulometria depositou informação 53 em buffer[6]
Thread capturadora de mensagens leu informação 43 em buffer[7]
Thread leitora de medição de granulometria depositou informação 55 em buffer[7]
Thread capturadora de mensagens leu informação 52 em buffer[8]
Thread leitora de medição de granulometria depositou informação 57 em buffer[8]
Thread capturadora de mensagens leu informação 45 em buffer[9]
Thread leitora de medição de granulometria depositou informação 59 em buffer[9]
Thread capturadora de mensagens leu informação 54 em buffer[0]
Thread leitora de medição de granulometria depositou informação 61 em buffer[0]
.....
Tarefa LEITURA DE DADOS DO PROCESSO *DESBLOQUEADA*.
.....

*****
Capacidade máxima da primeira lista circular em memória atingida.
Thread de leitura de dados do processo tentou depositar informação e está se bloqueando até livrar posição.
*****
```

Figura 10 - Bloqueio/Desbloqueio da tarefa de leitura do processo

**Comentário:** De forma análoga à explicada na Figura 9, aplica-se o bloqueio e desbloqueio *tarefa de leitura de dados do processo*.



```
Processo analise de granulometria disparado

.....

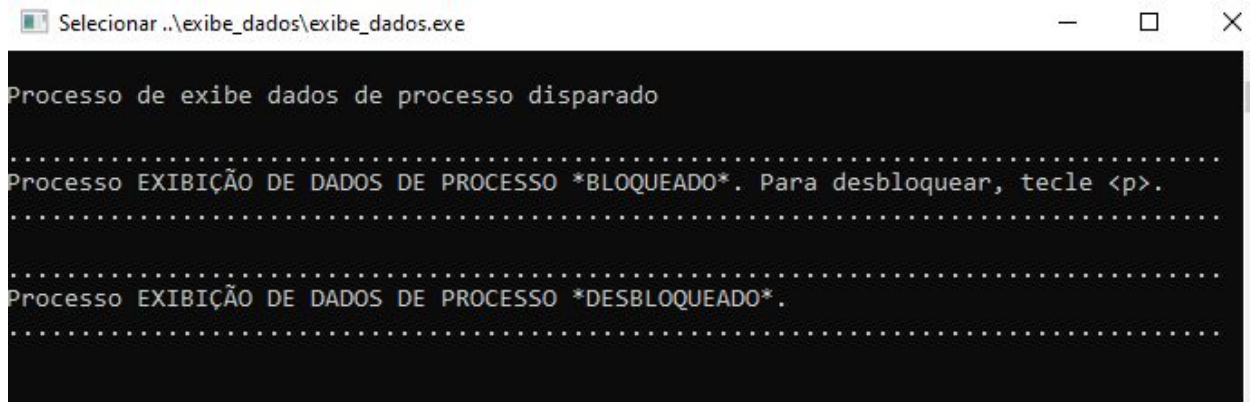
Processo ANÁLISE DE GRANULOMETRIA *BLOQUEADO*. Para desbloquear, tecle <a>.
.....

.....

Processo ANÁLISE DE GRANULOMETRIA *DESBLOQUEADO*.
.....
```

Figura 11 - Bloqueio/Desbloqueio da tarefa de análise de granulometria





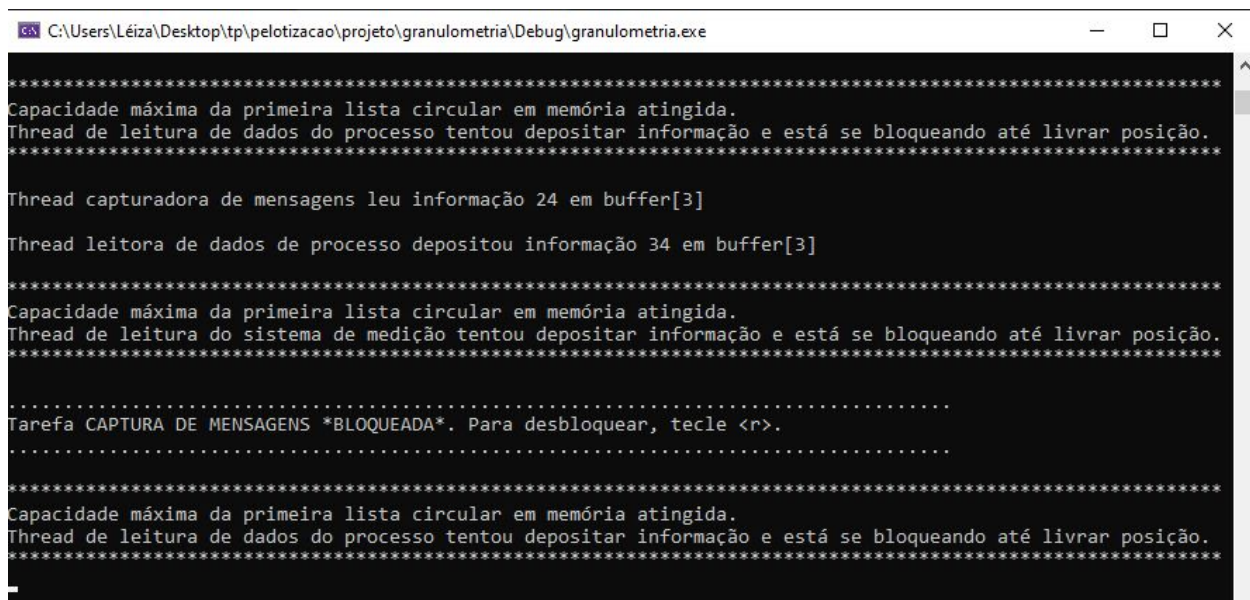
```
Selecionar ..\exibe_dados\exibe_dados.exe

Processo de exibe dados de processo disparado

.....
Processo EXIBIÇÃO DE DADOS DE PROCESSO *BLOQUEADO*. Para desbloquear, tecle <p>.
.....

.....
Processo EXIBIÇÃO DE DADOS DE PROCESSO *DESBLOQUEADO*.
.....
```

Figura 12 - Bloqueio/Desbloqueio da tarefa de exibição de dados de processo



```
C:\Users\Léiza\Desktop\tp\pelotizacao\projeto\granulometria\Debug\granulometria.exe

*****
Capacidade máxima da primeira lista circular em memória atingida.
Thread de leitura de dados do processo tentou depositar informação e está se bloqueando até livrar posição.
*****

Thread capturadora de mensagens leu informação 24 em buffer[3]
Thread leitora de dados de processo depositou informação 34 em buffer[3]

*****
Capacidade máxima da primeira lista circular em memória atingida.
Thread de leitura do sistema de medição tentou depositar informação e está se bloqueando até livrar posição.
*****

.....
Tarefa CAPTURA DE MENSAGENS *BLOQUEADA*. Para desbloquear, tecle <r>.
.....

*****
Capacidade máxima da primeira lista circular em memória atingida.
Thread de leitura de dados do processo tentou depositar informação e está se bloqueando até livrar posição.
*****
```

Figura 13 - Bloqueio da tarefa de captura de mensagens

**Comentário:** Quando a *tarefa de captura de mensagens* é bloqueada, o *buffer* alcança sua capacidade máxima e a aplicação fica parada, aguardando o desbloqueio da tarefa para continuar ou então o encerramento dos processos.

```
C:\Users\Léiza\Desktop\tp\pelotizacao\projeto\granulometria\Debug\granulometria.exe
.....
Tarefa LEITURA DO SISTEMA DE MEDIÇÃO *BLOQUEADA*. Para desbloquear, tecle <g>.
.....
Thread capturadora de mensagens leu informação 56 em buffer[6]
Thread leitora de dados de processo depositou informação 66 em buffer[6]
.....
Tarefa LEITURA DE DADOS DO PROCESSO *BLOQUEADA*. Para desbloquear, tecle <c>.
.....
Thread capturadora de mensagens leu informação 57 em buffer[7]
Thread capturadora de mensagens leu informação 58 em buffer[8]
Thread capturadora de mensagens leu informação 59 em buffer[9]
Thread capturadora de mensagens leu informação 60 em buffer[0]
Thread capturadora de mensagens leu informação 61 em buffer[1]
Thread capturadora de mensagens leu informação 62 em buffer[2]
Thread capturadora de mensagens leu informação 63 em buffer[3]
Thread capturadora de mensagens leu informação 64 em buffer[4]
Thread capturadora de mensagens leu informação 65 em buffer[5]
Thread capturadora de mensagens leu informação 66 em buffer[6]
```

Figura 14 - Bloqueio da tarefa de captura de mensagens

**Comentário:** Quando as duas tarefas produtoras são bloqueadas, não há inserção de novas informações no *buffer* e a *thread* de captura de mensagens finaliza de consumir as mensagens. A aplicação então aguarda o desbloqueio de uma ou mais *threads* produtoras para continuar a execução ou o encerramento dos processos, representado na Figura 15.

```
Console de Depuração do Microsoft Visual Studio

Thread capturadora de mensagens leu informação 58 em buffer[8]
Thread capturadora de mensagens leu informação 59 em buffer[9]
Thread capturadora de mensagens leu informação 60 em buffer[0]
Thread capturadora de mensagens leu informação 61 em buffer[1]
Thread capturadora de mensagens leu informação 62 em buffer[2]
Thread capturadora de mensagens leu informação 63 em buffer[3]
Thread capturadora de mensagens leu informação 64 em buffer[4]
Thread capturadora de mensagens leu informação 65 em buffer[5]
Thread capturadora de mensagens leu informação 66 em buffer[6]

Tarefa de leitura de dados de processo encerrando.
Tarefa de captura de dados encerrando.
Tarefa de leitura do sistema de medição encerrando.

Processo encerrando.

O C:\Users\Léiza\Desktop\tp\pelotizacao\projeto\granulometria\Debug\granulometria.exe (processo 11552) foi encerrado com o código 0.
Para fechar o console automaticamente quando a depuração parar, habilite Ferramentas -> Opções -> Depuração -> Fechar o console automaticamente quando a depuração parar.
Pressione qualquer tecla para fechar esta janela...
```

Figura 15 - Encerramento dos processos após bloqueio

```
Console de Depuração do Microsoft Visual Studio

Thread capturadora de mensagens leu informação 6 em buffer[7]

.....
Tarefa LEITURA DO SISTEMA DE MEDIÇÃO *BLOQUEADA*. Para desbloquear, tecle <g>.
.....

Tarefa LEITURA DE DADOS DO PROCESSO *BLOQUEADA*. Para desbloquear, tecle <c>.
.....

Thread capturadora de mensagens leu informação 9 em buffer[8]
Thread capturadora de mensagens leu informação 8 em buffer[9]

.....
Tarefa CAPTURA DE MENSAGENS *BLOQUEADA*. Para desbloquear, tecle <r>.
.....

Tarefa de leitura de dados de processo encerrando.
Tarefa de leitura do sistema de medição encerrando.
Tarefa de captura de dados encerrando.

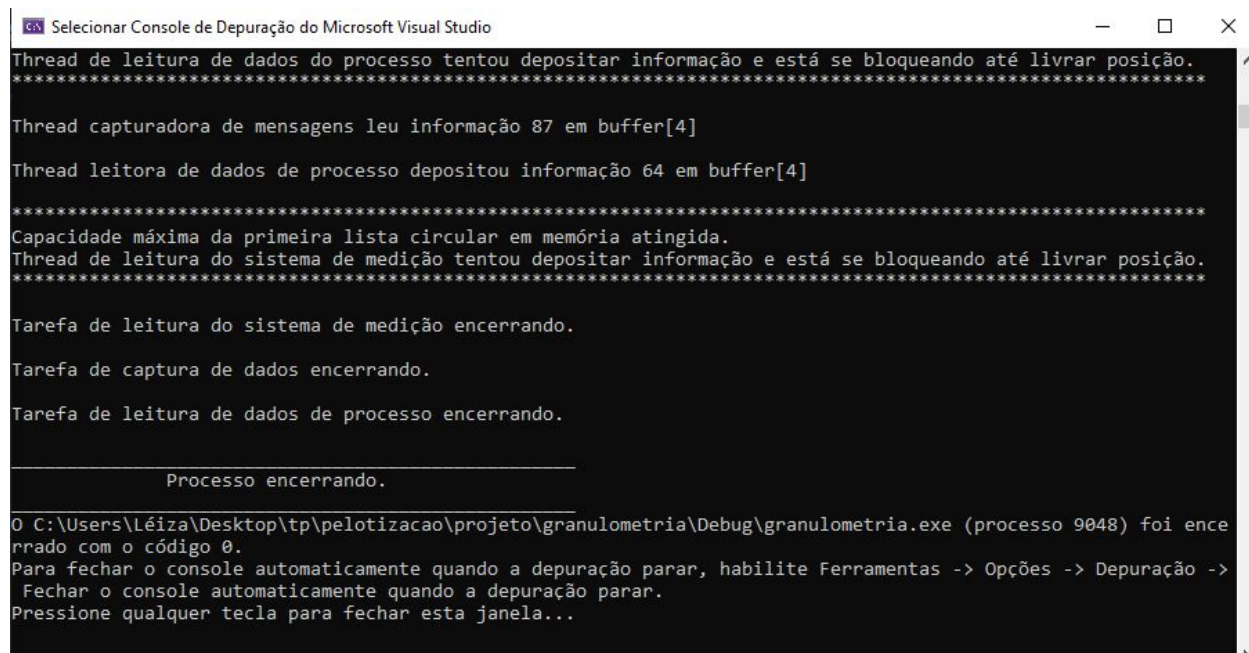
Processo encerrando.

O C:\Users\Léiza\Desktop\tp\pelotizacao\projeto\granulometria\Debug\granulometria.exe (processo 4952) foi encerrado com o código 0.
Para fechar o console automaticamente quando a depuração parar, habilite Ferramentas -> Opções -> Depuração -> Fechar o console automaticamente quando a depuração parar.
Pressione qualquer tecla para fechar esta janela...
```

Figura 16 - Encerramento dos processos após bloqueio



**Comentário:** Foi aplicado um teste em que, todas as tarefas foram bloqueadas, Figuras 16, 11 e 12. Após isso, a tecla “Esc” foi pressionada e todos os processos encerrados.



```
Selecionar Console de Depuração do Microsoft Visual Studio

Thread de leitura de dados do processo tentou depositar informação e está se bloqueando até livrar posição.
*****

Thread capturadora de mensagens leu informação 87 em buffer[4]

Thread leitora de dados de processo depositou informação 64 em buffer[4]

*****
Capacidade máxima da primeira lista circular em memória atingida.
Thread de leitura do sistema de medição tentou depositar informação e está se bloqueando até livrar posição.
*****

Tarefa de leitura do sistema de medição encerrando.

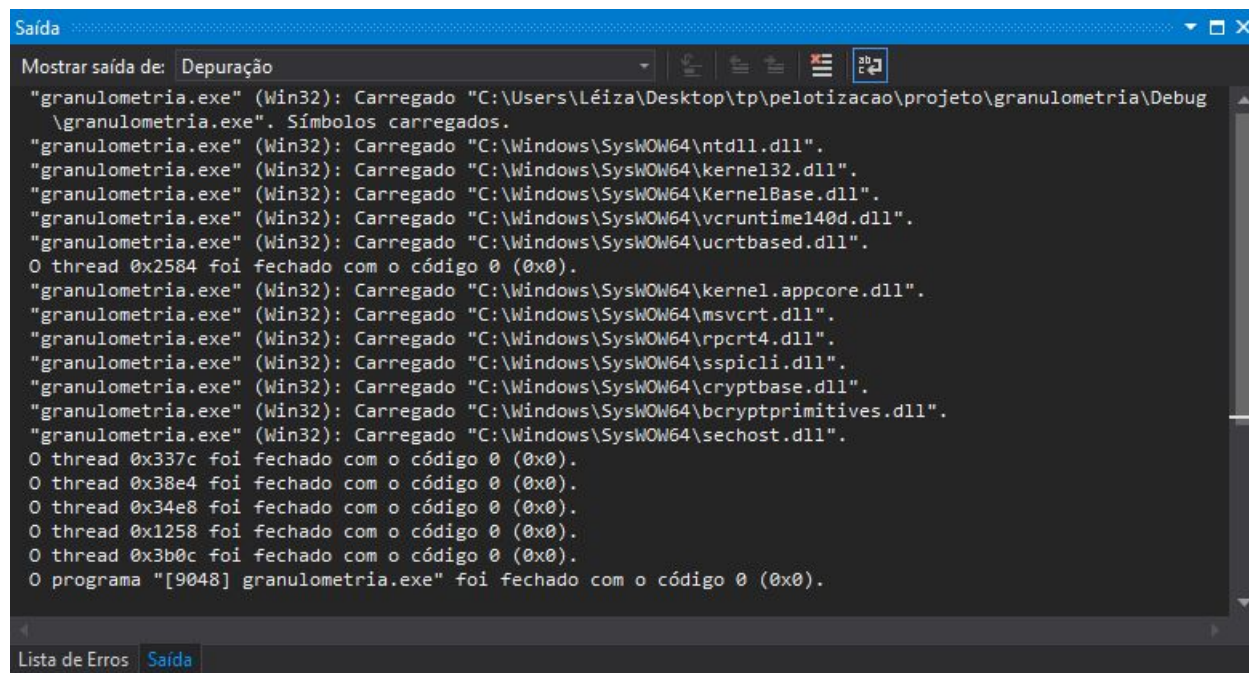
Tarefa de captura de dados encerrando.

Tarefa de leitura de dados de processo encerrando.

-----
Processo encerrando.

O C:\Users\Léiza\Desktop\tp\pelotizacao\projeto\granulometria\Debug\granulometria.exe (processo 9048) foi encerrado com o código 0.
Para fechar o console automaticamente quando a depuração parar, habilite Ferramentas -> Opções -> Depuração -> Fechar o console automaticamente quando a depuração parar.
Pressione qualquer tecla para fechar esta janela...
```

Figura 17 - Encerramento dos processos, sem condição especial



```
Saída

Mostrar saída de: Depuração

"granulometria.exe" (Win32): Carregado "C:\Users\Léiza\Desktop\tp\pelotizacao\projeto\granulometria\Debug\granulometria.exe". Símbolos carregados.
"granulometria.exe" (Win32): Carregado "C:\Windows\SysWOW64\ntdll.dll".
"granulometria.exe" (Win32): Carregado "C:\Windows\SysWOW64\kernel32.dll".
"granulometria.exe" (Win32): Carregado "C:\Windows\SysWOW64\KernelBase.dll".
"granulometria.exe" (Win32): Carregado "C:\Windows\SysWOW64\vcruntime140d.dll".
"granulometria.exe" (Win32): Carregado "C:\Windows\SysWOW64\ucrtbased.dll".
O thread 0x2584 foi fechado com o código 0 (0x0).
"granulometria.exe" (Win32): Carregado "C:\Windows\SysWOW64\kernel.appcore.dll".
"granulometria.exe" (Win32): Carregado "C:\Windows\SysWOW64\msvcrt.dll".
"granulometria.exe" (Win32): Carregado "C:\Windows\SysWOW64\rpcrt4.dll".
"granulometria.exe" (Win32): Carregado "C:\Windows\SysWOW64\sspicli.dll".
"granulometria.exe" (Win32): Carregado "C:\Windows\SysWOW64\cryptbase.dll".
"granulometria.exe" (Win32): Carregado "C:\Windows\SysWOW64\bcryptprimitives.dll".
"granulometria.exe" (Win32): Carregado "C:\Windows\SysWOW64\sechost.dll".
O thread 0x337c foi fechado com o código 0 (0x0).
O thread 0x38e4 foi fechado com o código 0 (0x0).
O thread 0x34e8 foi fechado com o código 0 (0x0).
O thread 0x1258 foi fechado com o código 0 (0x0).
O thread 0x3b0c foi fechado com o código 0 (0x0).
O programa "[9048] granulometria.exe" foi fechado com o código 0 (0x0).
```

Figura 18 - Processos encerrados



## 7. REFERÊNCIAS

[1]

POLICARPO, Flávio F. **Minério de ferro: desafios para as indústrias mineral e siderúrgica**. 2012. 64 páginas. Universidade Federal de Minas Gerais - UFMG, Belo Horizonte. Disponível em:

<[https://repositorio.ufmg.br/bitstream/1843/BUOS-9CAH4A/1/monografia\\_\\_\\_vers\\_o\\_final\\_\\_\\_fl\\_vio\\_ferreira\\_policarpo.pdf](https://repositorio.ufmg.br/bitstream/1843/BUOS-9CAH4A/1/monografia___vers_o_final___fl_vio_ferreira_policarpo.pdf)>. Acesso em: 20 de fev. de 2021.

[2]

VOCÊ sabe o que é pelotização? **Vale**. Disponível em:

<<http://www.vale.com/brasil/PT/aboutvale/news/Paginas/voce-sabe-o-que-e-pelotizacao.aspx#:~:text=Eles%20são%20todos%20feitos%20de,dá%20nas%20usinas%20de%20pelotização>>. Acesso em: 20 de fev. de 2021.

[3]

ENTENDA como funciona o processo de pelotização. **Vale**. Disponível em:

<<http://www.vale.com/brasil/pt/aboutvale/news/paginas/entenda-funciona-processo-pelotizacao-usinas.aspx>>. Acesso em: 20 de fev. de 2021.

[4]

INTEL Guide for Developing Multithreaded Applications. **Intel Software Dispatch**.

Disponível em:

<[http://runge.math.smu.edu/Courses/Math6370\\_Spring11/intel\\_multithreading\\_guide1.pdf](http://runge.math.smu.edu/Courses/Math6370_Spring11/intel_multithreading_guide1.pdf)>. Acesso em: 22 de fev. de 2021.

[5]

MENDES, Luiz Themystokliz. **Sincronismo entre Threads - Parte II**. 35 slides.

[6]

MENDES, Luiz Themystokliz. **Multithreading na plataforma Windows I**. slides.