UNIVERSIDADE FEDERAL DE MINAS GERAIS

DEPARTAMENTO DE ENGENHARIA ELETRÔNICA

Letícia Maria Resende

Pedro Luís Silva Giacomin

Universidade Federal de Minas Gerais – UFMG

TRABALHO DE AUTOMAÇÃO EM TEMPO REAL

PARTE II

Belo Horizonte

03 de dezembro de 2023

Sumário

1- Introdução

Este trabalho se concentra em aprimorar o controle e monitoramento do processo de dessulfuração em uma indústria siderúrgica. A dessulfuração é um procedimento crucial para melhorar as propriedades mecânicas do aço, essenciais para diversas aplicações industriais.

Nesse viés, o objetivo do trabalho é o desenvolvimento de uma aplicação de software multithread para integrar as informações provenientes de três Controladores Lógicos Programáveis (CLPs). Os CLPs 1 e 2 controlam sistemas independentes de dessulfuração, enquanto o CLP 3 monitora eventos críticos durante o processo. O objetivo é apresentar esses dados de maneira integrada em terminais de vídeo na cabine de operação, proporcionando uma visão abrangente dos dados de processo e alarmes críticos aos operadores.

2- Arquitetura

A arquitetura implementada consiste em uma solução central composta por três processos distintos. Essa estrutura modular e bem definida visa otimizar o controle e monitoramento do processo de dessulfuração na indústria siderúrgica. Dessa forma, a descrição do processo assim como das tarefas pode ser vista a seguir:

2.1. Processo principal

O Processo Principal é responsável não só pela criação dos outros dois processos presentes na aplicação desenvolvida, como também pela criação dos eventos que serão disparados quando o operador der algum comando. Dessa forma, esse processo atua como núcleo central do sistema e irá coordenar e gerenciar as demais tarefas. Ele é composto por três tarefas, sendo elas:

- **Thread Primária (main):** Responsável não só por criar um evento para cada comando do operador, como também por criar os demais processos.
- **Thread LeituraTeclado:** Recebe os comandos do operador, via a função *_getch*, e executa o tratamento adequado por meio do disparo de eventos.
- **Thread MonitoraListas:** Fica aguardando a sinalização de um evento indicando que alguma das listas em memória ficou cheia para que, quando o evento ocorrer, consiga sinalizar no seu console.

2.2. Processo exibirAlarmes

Este processo tem como foco exclusivo a exibição de alarmes críticos gerados durante o processo de dessulfuração. As mensagens de alarme chegam por meio de um mailslot.

O processo é formado pelas threads:

- **Thread Primária (main):** Responsável pela abertura dos eventos que serão utilizados no processo exibirAlarmes e pela criação das demais threads do processo.
- **Thread ExibirAlarme:** Verifica se existem mensagens no mailslot por meio da função *GetMailslotInfo* e em caso afirmativo, exibe esses alarmes no console.
- **Thread MonitoraEvento:** Esta thread monitora a ocorrência do evento relacionado ao comando do operador para bloquear/desbloquear a thread ExibirAlarme ou para encerrar a aplicação. Uma vez detectada a ocorrência desses comandos, ela é sinalizada à thread ExibirAlarme por meio de variáveis globais.

2.3. Processo mensagensCLP

Este processo abriga as tarefas que estão relacionadas com a leitura do CLP, retirada de mensagens da lista e exibição de dados do processo. No console desse processo, são exibidos os dados recebidos dos CLPs 1 e 2 cujo campo de diagnóstico de entrada e saída é diferente de 55 (entrada/saída sem falha).

A descrição mais detalhada das tarefas listadas é feita a seguir:

- **Thread Primária (main):** Cria as demais threads e também os semáforos e mutexes que serão utilizados nesse processo. Além disso, obtém os HANDLES para os eventos que haviam sido criados no processo principal e que alguma thread do processo mensagensCLP deverá usar.
- **Threads LeituraCLP:** Simulam a leitura das mensagens dos CLPs de processo (1 e 2), depositando os dados na primeira lista circular em memória.
- **Thread RetiraMensagem:** Retira mensagens da primeira lista circular em memória, redirecionando-as conforme sua tipologia para uma segunda lista circular em memória ou para a thread de Exibição de Alarmes, que se encontra em outro processo, por meio de um mailslot.
- **Thread ExibeDadosProcesso:** Retira as mensagens da segunda lista circular em memória e as exibe no console .
- **Thread MonitoraAlarme:** Simula a leitura do CLP 3, de monitoração, gerando alarmes com periodicidade aleatória entre 1 e 5 segundos.

3- Decisões de projeto

- Os eventos são todos criados no processo principal, pois são usados para sincronizar processos diferentes.
- Para a implementação das mensagens e alarmes, foram criadas duas structs que salvam as informações necessárias em cada um desses dois tipos. Os nomes são, respectivamente, **msgType** e **almType**.
- A implementação das listas circulares foi feita baseada no Problema dos Produtores e Consumidores. Nessa perspectiva, para simplificação do projeto, todas as threads que fossem produtoras ou consumidoras foram colocadas em um mesmo processo (MensagemCLP).

- Tendo em vista que a thread LeituraTeclado, do processo principal, utiliza a função bloqueante *_getch*, foi necessário criar uma thread adicional (MonitoraEvento) no processo principal para monitorar a ocorrência dos eventos de lista cheia.

4- Biblioteca Tp.h

Com o intuito de melhorar a legibilidade do código foram implementadas, em um arquivo .h, algumas funções auxiliares que serão descritas a seguir:

- `int setId()`: Sorteia aleatoriamente o ID do alarme e retorna esse valor.
- `float setTemp()`: Sorteia aleatoriamente o valor da temperatura do processo em uma faixa de valores entre 1000 a 2000 com precisão de 1 casa decimal e retorna esse valor.
- `float setPress()`: Sorteia aleatoriamente o valor da pressão em uma faixa entre 100 a 200 com precisão de uma casa decimal e retorna esse valor.
- `int setDiag()`: Sorteia aleatoriamente o valor de DIAG (diagnóstico dos cartões E/S do CLP) em uma faixa de 0 a 70 e retorna esse valor.
- `string getTime(SYSTEMTIME tempo)`: Retorna uma string contendo a hora , o minuto e o segundo do momento da chamada da função.
- `void produzMensagem(msgType &mensagem , int ID, int NSEQ_aux)`: Recebe como parâmetro o ID da thread, o campo NSEQ da mensagem a ser produzida e uma struct onde será armazenada a mensagem. E os demais campos são sorteados.
- `void produzAlarme(almType &alarme, int NSEQ_aux)`: Recebe como parâmetro o campo NSEQ do alarme que é produzido e armazenado na estrutura alarme que também é recebida como parâmetro.
- `int randTime1a5s()`: Retorna um tempo aleatório entre 1s e 5s em pacotes de 100ns.
- `void printMensagemAlarme(int id)`: Recebe como parâmetro o identificador do alarme e exibe a mensagem correspondente a esse Id.

Além das funções auxiliares, nesse arquivo também foi definido duas estruturas e um novo tipo de dados, que serão descritos a seguir:

- `enum estado`: Esse novo tipo de dado permite que variáveis do tipo estado assumam dois valores, bloqueado ou desbloqueado.
- `struct almType`: Representa a estrutura dos alarmes que são compostos por:
  - número sequencial (nSeq);
  - um identificador (Id);
  - um horário (timestamp).
- `struct msgType`: Representa a estrutura das mensagens do processo que são compostas por:
  - número sequencial (nSeq);
  - um inteiro que representa o diagnóstico dos cartões E/S do CLP (diag);
  - um identificador (Id);
  - um horário (timestamp)
  - a pressão interna na panela de gusa (pressInt)
  - a pressão de injeção do nitrogênio(pressInj)
  - a temperatura no interior da panela de gusa (temp).

5- Princípios de Funcionamento

A seguir está descrito de maneira geral como foram implementadas as funcionalidades do sistema.

5.1. Estruturas de bloqueio/desbloqueio e encerramento

Cada thread executa em um loop, cuja condição de saída é a ocorrência do evento **EventoESC** (com reset manual, mais detalhado na seção 6.1). Ao pressionar a tecla ESC no processo Principal, esse evento é disparado e todas as threads devem se encerrar.

Ademais, com exceção da tarefa LeituraTeclado, todas as outras 6 tarefas podem ser bloqueadas e desbloqueadas ao pressionar-se uma tecla específica. A implementação dessa estrutura de bloqueio/desbloqueio foi feita da seguinte forma:

1. Foi definido um evento para cada uma dessas teclas **Evento_x** (**x** é um valor que depende da tecla, mais detalhado na seção 6.1), que corresponde ao Bloqueio/Desbloqueio de cada thread. O evento é disparado pelo processo Principal ao pressionar da respectiva tecla.
2. Foi definido um tipo enumerado chamado estado (detalhado na seção 4), cujos valores possíveis são BLOQUEADO e DESBLOQUEADO.
3. Cada thread tem uma variável local desse tipo, a qual é inicializada com o valor DESBLOQUEADO.
4. A cada iteração do loop, a thread testa o valor dessa variável.
   a. Se for DESBLOQUEADO, prossegue com sua execução normal.
   b. Se for BLOQUEADO, se bloqueia até receber outra sinalização de **Evento_x**.

Assim, o funcionamento básico de cada thread é descrito pelo pseudocódigo:

```
estado estadoTarefa = DESBLOQUEADO;
do {
    if (estadoTarefa == DESBLOQUEADO) {
       // Execução normal
    }
    else if (estadoTarefa == BLOQUEADO){
        tipoEvento = WaitForMultipleObjects([Evento_x, EventoESC]);
        if (tipoEvento == Evento_x) {
            estadoTarefa =  DESBLOQUEADO;
            continue;
        }
        else if (tipoEvento == EventoESC) {
            printf("Tecla ESC digitada, encerrando o programa... \n");
            continue;
        }
    }
} while(tipoEvento != EventoESC);
```

Todavia, pelo código acima não fica claro quando a thread espera pelo **Evento_x**, para de fato entrar no estado BLOQUEADO. A resposta é que ela espera por esse evento, **todas as vezes que espera pela sinalização de algum objeto do kernel** (evento, semáforo, mutex ou temporizador). Assim, no estado DESBLOQUEADO:

```
if (estadoTarefa == DESBLOQUEADO) {
    . . .
    // Precisou usar um objeto do kernel
    tipoEvento = WaitForMultipleObjects([Evento_x, EventoESC, Objeto]);
    if (tipoEvento == Evento_x) {
        estadoTarefa = BLOQUEADO;
        continue;
    }
    else if(tipoEvento == EventoESC) {
        printf("Tecla ESC digitada, encerrando o programa... \n");
        continue;
    }
    else if (tipoEvento == Objeto) {
        // Executa o código que usa esse objeto
    }
    . . .
}
```

5.2. Temporização

As threads de LeituraCLP são executadas em ciclos de 500 ms, enquanto a thread MonitoraAlarmes é executada em intervalos de 1 a 5 s. Para implementar a temporização foram usados **Waitable Timers** exclusivos para cada thread.

No início da execução de cada uma dessas threads, elas esperam por um objeto Timer usando *WaitForMultipleObjects* (como mostrado na seção 5.1) e só seguem com sua execução normal quando seu respectivo temporizador é sinalizado.

Como todas essas threads fazem parte do mesmo processo (mensagensCLP), os temporizadores foram declarados e inicializados na thread primária antes da inicialização dessas threads. Para as threads de LeituraCLP, bastou inicializar o temporizador com um período de 500 ms; já para a thread MonitoraAlarme, como o período de temporização não é fixo, foi feito o seguinte:

1. Declarada a função *randTime1a5s* (seção 4).
2. O temporizador foi inicializado com um período aleatório, definido por essa função.
3. A cada disparo do temporizador, a thread MonitoraAlarme:
   a. Executa seu código.
   b. Cancela o temporizador com *CancelWaitableTimer*.
   c. Calcula um novo tempo aleatório com a função descrita em (1).

d. Reinicializa o temporizador com *SetWaitableTimer,* passando como parâmetro o tempo calculado.

Assim, foi possível executar o código de MonitoraAlarme com periodicidade aleatória entre 1 e 5 segundos.

5.3. Escrita e leitura na lista circular 1

A lista circular 1 tem as seguintes características:

- A lista circular é um array de structs do tipo msgType com 100 posições.
- As 2 threads LeituraCLP depositam mensagens na lista.
- A thread RetiraMensagem retira mensagens da lista.
- A variável global **pLivre1** indica a próxima posição livre para escrita na lista 1.
- A variável global **pOcup1** indica a posição da próxima mensagem que deve ser lida.
- São usados dois semáforos para sincronizar o depósito e retirada de mensagens da lista: **Lista1Livre** (inicializado com 100) e **Lista1Ocup** (inicializado com 0).

Como dito anteriormente, foi implementada uma estrutura baseada na solução do Problema dos Produtores e Consumidores. O procedimento de escrita é o seguinte:

1. Antes de depositar uma mensagem, a thread LeituraCLP usa *WaitForMultipleObjects* no semáforo **Lista1Livre** (como mostrado na seção 5.1)
   a. Se o semáforo está em 0, significa que não há posições livres na lista, e a thread se bloqueia por um tempo máximo de 10 ms. Passado esse tempo, assume-se que a lista está cheia, e o evento **Evento_L1** é sinalizado, então a thread volta a esperar indefinidamente por uma sinalização do semáforo **Lista1Livre**.
   b. Se o semáforo tem um valor maior que 0, significa que há posições livres para escrita e a thread não se bloqueia.
2. Tendo posições livres, a thread espera pelo mutex **hMutexpLivre1.**
3. Conquistando o mutex, deposita a mensagem na posição **pLivre1** e incrementa essa variável em uma unidade.
4. Libera o mutex **hMutexpLivre1.**
5. Então, incrementa em 1 o semáforo **Lista1Ocup**, para alertar a thread RetiraMensagem de que há um novo dado a ser lido.

O uso do mutex só foi necessário porque são duas threads tentando escrever, fazendo com que a variável **pLivre1** precise ser acessada com exclusão mútua.

A operação de leitura é similar, sendo que:

1. Antes de ler uma mensagem, a thread RetiraMensagem usa *WaitForMultipleObjects* no semáforo **Lista1Ocup**.
   a. Caso o semáforo esteja em 0, significa que não há mensagens a serem lidas e a thread se bloqueia até o semáforo ser sinalizado.

   b. Caso o semáforo tenha um valor maior que 0, significa que há mensagens a serem lidos e a thread não se bloqueia
2. Tendo mensagens a serem lidas, a thread lê o dado da posição **pOcup1** e incrementa essa variável em uma unidade.
3. Então, incrementa em 1 o semáforo **Lista1Livre**, para alertar às threads LeituraCLP que há uma nova posição livre para escrita.

Como foi dito anteriormente, quando a lista está cheia, é sinalizado o evento com reset automático **Evento_L1**. Esse evento é recebido pela thread LeituraTeclado, que imprime para o usuário a informação de que a lista 1 está cheia.

Como são duas threads operando a lista 1, para evitar que o evento seja sinalizado duas vezes quando a lista estiver cheia, foi criada a variável de controle **ListaCheiaNotificada**. Assim, antes de sinalizar o evento, a thread:

1. Conquista o mutex **hMutexListaCheia**
2. Verifica se **ListaCheiaNotificada** está em TRUE
   a. Se sim, significa que o evento já foi sinalizado pela outra thread, então ela não sinaliza o evento **Evento_L1**.
   b. Se não, significa que o evento não foi sinalizado, então sinaliza **Evento_L1.**
3. Libera o mutex **hMutexListaCheia** e se bloqueia

5.4. Escrita e leitura na lista circular 2

A estrutura da lista 2 é idêntica à da lista 1, sendo que:

- A lista circular é um array de structs do tipo msgType com 50 posições.
- A thread RetiraMensagem depositam mensagens na lista.
- A thread ExibeDadosProcesso retira mensagens da lista.
- A variável global **pLivre2** indica a próxima posição livre para escrita na lista 2.
- A variável global **pOcup2** indica a posição da próxima mensagem que deve ser lida.
- São usados dois semáforos para sincronizar o depósito e retirada de mensagens da lista: **Lista2Livre** (inicializado com 50) e **Lista2Ocup** (inicializado com 0).

A principal diferença é que nem todas as mensagens lidas da lista 1 por RetiraMensagem são escritas na lista 2. A tarefa RetiraMensagem testa o campo DIAG da mensagem e:

1. Se for diferente de 55, deposita na lista 2 normalmente, da forma descrita na seção 5.3.
2. Se for igual a 55, escreve as informações da mensagem em um mailslot e envia para a tarefa ExibirAlarmes, que se encontra em um outro processo. Essa operação é detalhada na seção seguinte.

5.5. Inter Process Communication (IPC)

Com a arquitetura implementada foi necessário fazer apenas duas comunicações entre processos:

1. Entre a thread RetiraMensagens e ExibirAlarme
2. Entre a thread MonitoraAlarme e ExibirAlarme

O mecanismo utilizado foi o **mailslot**. Dessa forma, tendo em vista que as duas threads clientes estão no mesmo processo e que ambas irão enviar mensagem para o mesmo servidor, os clientes utilizam o mesmo handle do mailslot que foi declarado globalmente no processo MensagensCLP e que é obtido na thread primária desse processo. É válido ressaltar que o handle só é obtido após o servidor notificar que criou o mailslot como pode ser visto nos trechos de código a seguir:

```
hMailslot = CreateMailslot(
      L"\\\\.\\mailslot\\MyMailslot",
      0,
      MAILSLOT_WAIT_FOREVER,
      NULL);
CheckForError(hMailslot != INVALID_HANDLE_VALUE);
SetEvent(event_mailslotCriado);
```

Além disso, para garantir que não haja nenhuma tentativa de escrita ao mesmo tempo, ambas as threads, antes de escreverem no mailslot, devem conquistar o mutex **hMutexMailslot** e depois de realizarem a escrita, liberá-lo.

A leitura do mailslot pela thread ExibirAlarme (do processo exibirAlarmes) é feita com o auxílio da função *GetMailslotInfo*, que fornece a informação de quantas mensagens estão pendentes no mailslot. Assim, a thread lê o mailslot e imprime os alarmes enquanto o número de mensagens nele for maior que 0, como mostrado no pseudocódigo a seguir:

```
int messages = 0;
do {
    GetMailslotInfo(&messages, &...);
    // retorna o número de mensagens na variável messages
    if (messages > 0) { // Existem mensagens no mailslot a serem lidas
        do {
            bStatus = ReadFile(hMailslot, &...);
            . . . // Imprime a mensagem lida
            messages--;
        } while (messages != 0);
    }
} while (ESC==FALSE);
```

5.6. Número sequencial das mensagens (NSEQ)

As duas threads de LeituraCLP devem produzir mensagens de processo, as quais têm um número de sequência que deve ser sempre crescente e único (até um máximo de 99999). Essa funcionalidade foi implementada usando-se uma variável global **NSEQ** (inicializada em 0) e garantindo exclusão mútua na leitura e no incremento dela, utilizando o mutex **hMutexNSEQ**.

A função *produzMensagem* (usada em LeituraCLP, e detalhada na seção 4) recebe o valor de NSEQ como parâmetro, mas antes desse valor ser passado para essa função:

1. A thread conquista o mutex **hMutexNSEQ**.
2. Salva o valor de NSEQ em uma variável auxiliar (local).
3. Incrementa NSEQ.
4. Libera o mutex **hMutexNSEQ**.
5. Passa o valor salvo na variável auxiliar para a função.

Assim, é garantido que o número de sequência será sempre incrementado corretamente e que seja sempre único (até, claro, o limite de 99999).

5.7. Número sequencial dos alarmes (NSEQAlarme)

Os alarmes têm um número de sequência independente do NSEQ das mensagens. Para isso foi criada a variável **NSEQAlarme**, que é incrementada todas as vezes que se usa a função *produzAlarme*. Todavia, como apenas a thread MonitoraAlarme tem acesso a essa variável, não é necessária exclusão mútua.

6- Objetos do Kernel

É apresentado a seguir um sumário dos objetos de kernel utilizados, bem como seus respectivos objetivos.

6.1. Eventos

Os eventos foram todos criados no processo Principal e são usados para sincronização entre processos. A seguir estão os eventos e suas funções:

- **Evento_0:** Bloqueio e desbloqueio da tarefa LeituraCLP correspondente ao CLP 1. É um evento de reset automático sinalizado pelo processo Principal quando é pressionada a ***tecla 1***.
- **Evento_1:** Bloqueio e desbloqueio da tarefa LeituraCLP correspondente ao CLP 2. É um evento de reset automático sinalizado pelo processo Principal quando é pressionada a ***tecla 2***.
- **Evento_2:** Bloqueio e desbloqueio da tarefa MonitoraAlarmes. É um evento de reset automático sinalizado pelo processo Principal quando é pressionada a ***tecla m***.

- **Evento_3:** Bloqueio e desbloqueio da tarefa RetiraMensagem. É um evento de reset automático sinalizado pelo processo Principal quando é pressionada a ***tecla r***.
- **Evento_4:** Bloqueio e desbloqueio da tarefa ExibeDadosProcesso. É um evento de reset automático sinalizado pelo processo Principal quando é pressionada a ***tecla p***.
- **Evento_5:** Bloqueio e desbloqueio da tarefa ExibeAlarmes. É um evento de reset automático sinalizado pelo processo Principal quando é pressionada a ***tecla a***.
- **Evento_L1:** Sinaliza que a lista circular 1 está cheia. É um evento com reset automático sinalizado pelas duas threads de LeituraCLP após aguardarem mais do que 10ms por uma posição na lista.
- **Evento_L2:** Sinaliza que a lista circular 2 está cheia. É um evento com reset automático sinalizado pela thread RetiraMensagem após aguardar mais do que 10ms por uma posição na segunda lista circular.
- **EventoESC:** Sinaliza o término da execução do programa. É um evento de reset manual sinalizado pelo processo Principal ao pressionar-se a ***tecla ESC***.

6.2 Semáforos

Em toda a aplicação foram usados apenas 4 semáforos, sendo todos criados na tarefa principal do processo MensagensCLP.

- **hLista1Livre:** O intuito desse semáforo é controlar o número de posições livres na primeira lista circular em memória. Como inicialmente considera-se que todas as posições estão disponíveis, seu valor inicial e máximo de contagem será 100.
- **hLista2Livre:** O intuito desse semáforo é controlar o número de posições livres na segunda lista circular em memória. Como inicialmente considera-se que todas as posições estão disponíveis, seu valor inicial e máximo de contagem será 50.
- **hLista1Ocup:** O intuito desse semáforo é sinalizar que existem mensagens a serem lidas na primeira lista circular em memória. Como inicialmente não há nenhuma mensagem disponível, seu valor inicial é 0 e o valor máximo de contagem é 100.
- **hLista2Ocup:** O intuito desse semáforo é sinalizar que existem mensagens a serem lidas na segunda lista circular em memória. Como inicialmente não há nenhuma mensagem disponível, seu valor inicial é 0 e o valor máximo de contagem é 50.

6.3 Mutexes

Os mutexes foram utilizados exclusivamente para proteger variáveis globais que são utilizadas por mais de uma thread. Nessa perspectiva, foram utilizados apenas dois mutexes em toda a aplicação, sendo eles:

- **hMutexNSEQ**: Mutex utilizado para proteger a variável NSEQ que indica o número sequencial das mensagens produzidas pelas tarefas de leitura do CLP. Como esse número deve ser único, toda thread de leitura do CLP deve conquistar esse mutex antes de produzir a sua mensagem e só liberá-lo depois que salvar o seu valor em uma variável local e incrementá-lo.
- **hMutexpLivre1**: Mutex utilizado para proteger a variável pLivre1. Isso se evidencia pois na aplicação desenvolvida, as duas threads de leitura do CLP são produtoras e

irão produzir mensagens que serão depositadas na primeira lista circular em memória. Dessa forma, para evitar que essas duas threads escrevam na mesma posição foi criado esse mutex.

- **hMutexMailslot**: Mutex utilizado para proteger a escrita no mailslot, uma vez que o mesmo mailslot é utilizado pelas threads RetiraMensagem e MonitoraAlarme para enviar mensagem ao servidor.
- **hMutexListaCheia:** Mutex utilizado para garantir que apenas uma thread por vez acesse a variável ListaCheiaNotificada que é uma variável de controle utilizada para que as threads saibam se o console principal já foi notificado que a lista está cheia.

6.4 Temporizadores

Os temporizadores foram utilizados para coordenar a execução das tarefas que fazem aquisição de dados do CLP de acordo com a sua periodicidade.

- **hTimerCLP[0]:** Temporizador periódico do CLP 1 que irá disparar a cada 500ms.
- **hTimerCLP[1]:** Temporizador periódico do CLP 2 que irá disparar a cada 500ms.
- **hTimerCLP[2]:** Temporizador aperiódico do CLP 3 que irá disparar em um intervalo de tempo entre 1 e 5s.

7- Referências

Referências para a criação dos alarmes

1. https://ppgem.eng.ufmg.br/defesas/1532M.PDF

2. https://monografias.ufop.br/bitstream/35400000/5435/6/MONOGRAFIA_Modelagem F%C3%ADsicaDessulfura%C3%A7%C3%A3o.pdf