Desenvolvimento de Software de Automação em Tempo Real para Dessulfuração na Indústria Siderúrgica: Uma Abordagem Baseada em Programação Concorrente

Marco Thulio Alves Maciel

December 4, 2023

1 Introdução

A automação industrial, especialmente em contextos que exigem alta precisão e resposta rápida, como é o caso das operações de dessulfuração na indústria siderúrgica, tem enfrentado desafios crescentes e demandas complexas. Neste cenário, a programação concorrente emerge como uma ferramenta crucial, proporcionando a base para o desenvolvimento de aplicações em *soft-real time*, onde a precisão temporal não é absolutamente crítica, mas ainda assim de vital importância.

1.1 Importância da Programação Concorrente em Aplicações Soft-Real Time

A programação concorrente, que envolve a execução de múltiplas sequências de operações ao mesmo tempo, é essencial para sistemas de automação que demandam monitoramento e controle contínuos de processos. Em uma aplicação *soft-real time*, como a automação do processo de dessulfuração, a capacidade de gerenciar múltiplas tarefas simultaneamente - como leitura de sensores, controle de atuadores e interfaces de usuário - é crucial. A eficácia do multithreading permite que o sistema responda a eventos em um tempo considerado aceitável, garantindo a fluidez e a precisão das operações industriais.

1.2 Processos Físico-Químicos na Dessulfuração

O processo de dessulfuração é uma operação crítica na produção de aço, onde o enxofre é removido para melhorar a qualidade do metal. Este processo envolve reações físico-químicas complexas que devem ser monitoradas e controladas com precisão. A falha em manter os parâmetros ideais pode levar a produtos de qualidade inferior e problemas operacionais. Assim, a precisão e a confiabilidade

do software de controle são de extrema importância, reforçando a necessidade de um sistema robusto e eficiente de automação em tempo real.

1.3 Escolha do Projeto CMake e Ambiente Microsoft Visual Studio

Para enfrentar estes desafios, a escolha da ferramenta de construção de software CMake e do ambiente de desenvolvimento Microsoft Visual Studio foi estratégica. O CMake oferece uma abordagem sofisticada e flexível para a gestão de projetos de software, permitindo uma configuração eficiente e portátil. O Microsoft Visual Studio, por sua vez, proporciona um ambiente de desenvolvimento integrado (IDE) robusto e repleto de recursos, facilitando o desenvolvimento, o teste e a depuração de aplicações complexas.

1.4 A Linguagem C++ e a API Win32 no Ambiente Windows

A escolha da linguagem C++ para este projeto não foi por acaso. Sua capacidade de manipulação de baixo nível, juntamente com recursos de programação orientada a objetos, torna-a ideal para desenvolver sistemas que exigem alto desempenho e controle preciso. Além disso, a utilização da API Win32 no ambiente Windows permite uma interação mais direta e eficiente com o hardware do sistema, uma consideração crítica para aplicações em tempo real que demandam respostas rápidas e precisas.

2 Arquitetura

2.1 Visão Geral

O sistema de automação em tempo real é projetado para otimizar o processo de dessulfuração na indústria siderúrgica, melhorando a eficiência e a precisão das operações. A arquitetura do software é estruturada em torno de processos concorrentes que facilitam a leitura de dados de Controladores Lógicos Programáveis (CLPs), a tomada de decisões com base nesses dados, e a subsequente exibição de informações críticas para os operadores.

2.2 Descrição da Arquitetura

Conforme ilustrado na Figura 1, a arquitetura é composta pelas seguintes partes principais:

2.3 Leitura de CLPs

Duas threads principais são responsáveis pela leitura de dados: uma para processos e outra para alarmes. Estas threads leem os dados dos CLPs e os ar-

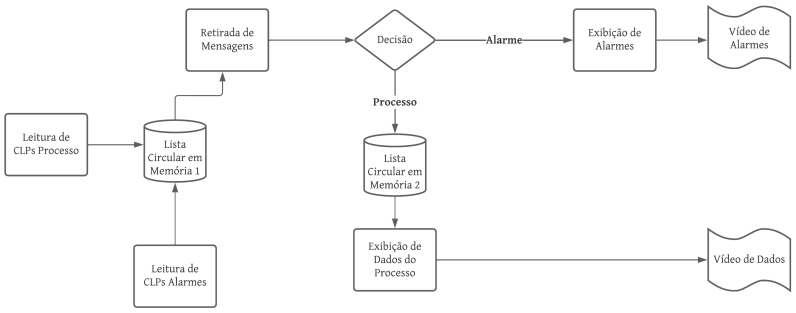

Figure 1: Diagrama Lógico do Software de Automação

mazenam em listas circulares em memória, isolando a coleta de dados da lógica de processamento.

2.4 Listas Circulares em Memória

Duas listas circulares em memória servem como buffers: uma para armazenar mensagens de processos e outra para mensagens de alarmes. Essas estruturas de dados permitem um acesso eficiente e gerenciamento de dados em um ambiente multithread.

2.5 Retirada e Decisão de Mensagens

Uma thread de retirada de mensagens consome dados das listas circulares e toma decisões baseadas no tipo de mensagem. Em caso de mensagens de alarme, a thread encaminha esses dados para o módulo de exibição de alarmes.

2.6 Exibição de Dados

Dois módulos de exibição operam de forma independente: um para exibir alarmes e outro para exibir dados do processo. Estes módulos são responsáveis por apresentar as informações em terminais de vídeo dedicados, permitindo que os operadores monitorem o estado atual do processo de dessulfuração.

3 Estrutura de Dados: Lista Circular em Memória RAM

3.1 BufferCircular: Uma Estrutura de Dados de Alto Desempenho

O `BufferCircular` é uma estrutura de dados essencial para o software de automação em tempo real, permitindo um gerenciamento de dados eficiente e

rápido sem as latências associadas ao uso de dispositivos de armazenamento secundário como HDs ou SSDs.

3.2 Definição e Implementação

O `BufferCircular` é implementado como uma classe em C++ que encapsula um vetor de strings, juntamente com indicadores para a cabeça e a cauda do buffer, um contador de elementos e uma seção crítica para gerenciamento de concorrência.

```
class BufferCircular {
private:
    std::vector<std::string> buffer;
    int cabeca = 0;
    int cauda = 0;
    int contador = 0;
    int tamanho; // Variável adicionada para armazenar o tamanho do buffer
    CRITICAL_SECTION secaoCritica;

public:
    // Construtor que aceita o tamanho do buffer como parâmetro
    BufferCircular(int tamanho) : tamanho(tamanho), buffer(tamanho) {
        InitializeCriticalSection(&secaoCritica);
    }

    ~BufferCircular() {
        DeleteCriticalSection(&secaoCritica);
    }

    bool adicionarDado(const std::string& dado) {
        EnterCriticalSection(&secaoCritica);
        if (contador == tamanho) { // Usa a variável tamanho aqui
            LeaveCriticalSection(&secaoCritica);
            return false;
        }
        buffer[cauda] = dado;
        cauda = (cauda + 1) % tamanho; // Usa a variável tamanho aqui
        contador++;
        LeaveCriticalSection(&secaoCritica);
        return true;
    }

    bool removerDado(std::string& dado) {
        EnterCriticalSection(&secaoCritica);
        if (contador == 0) {
            LeaveCriticalSection(&secaoCritica);
```

```
            return false;
        }
        dado = buffer[cabeca];
        cabeca = (cabeca + 1) % tamanho; // Usa a variável tamanho aqui
        contador--;
        LeaveCriticalSection(&secaoCritica);
        return true;
    }

    bool estaCheio() const {
        return contador == tamanho; // Usa a variável tamanho aqui
    }

    bool estaVazio() const {
        return contador == 0;
    }
};
```

3.3 Importância da Memória RAM Sobre o Armazenamento Secundário

A escolha de implementar o buffer em memória RAM é estratégica, aproveitando a alta velocidade de acesso da RAM em comparação com dispositivos de armazenamento secundário. Operações de leitura e escrita em memória RAM são significativamente mais rápidas do que aquelas realizadas em HDs ou SSDs, o que é crucial para sistemas que requerem processamento em tempo real.

3.4 Gerenciamento de Concorrência

A classe `BufferCircular` utiliza seções críticas para garantir que múltiplas threads possam operar no buffer de forma segura sem corromper os dados. Isso é crucial em um ambiente multithread onde a concorrência pode levar a condições de corrida.

3.5 Operações do Buffer

As operações principais do buffer incluem:

- `adicionarDado`: Insere um novo dado no buffer, se houver espaço disponível.
- `removerDado`: Remove e retorna um dado do buffer, se não estiver vazio.
- `estaCheio` e `estaVazio`: Métodos para verificar o estado do buffer.

3.6 Benefícios para o Sistema

A utilização do `BufferCircular` proporciona ao sistema uma capacidade de resposta rápida, essencial para manter a integridade dos processos de automação e a precisão dos dados apresentados aos operadores.

4 Funções de Exibição e Comunicação

O software de automação utiliza funções especializadas para gerar novas janelas de console e exibir dados de processo e alarmes, melhorando a organização visual e a usabilidade. Além disso, utiliza pipes nomeados para o encaminhamento eficiente das mensagens.

4.1 Processamento e Exibição de Mensagens de Processo

A função `processaMensagemProcesso` recebe uma mensagem bruta como entrada, a divide em tokens, verifica sua validade e reformata para exibição.

```
std::string processaMensagemProcesso(const std::string& message) {
    std::stringstream ss(message);
    std::string item;
    std::vector<std::string> tokens;

    // Dividindo a string com base no delimitador ';'
    while (std::getline(ss, item, ';')) {
        tokens.push_back(item);
    }

    // Verificando se a mensagem tem o número correto de tokens
    if (tokens.size() != 7) {
        return "Formato de mensagem inválido";
    }

    // Reformatando a mensagem
    std::string formattedMessage = tokens[6] + " NSEQ: " + tokens[0] + " PR INT: " + tokens

    return formattedMessage;
}
}
```

Esta função é fundamental para assegurar que os dados exibidos estejam em um formato consistente e de fácil interpretação pelos operadores.

4.2 Criação de Janelas de Console e Pipes Nomeados

As funções `ConsoleProcesso` e `ConsoleAlarmes` são responsáveis por criar janelas de console separadas e pipes nomeados para exibição de dados de pro-

cesso e alarmes, respectivamente.

```
void ConsoleProcesso() {
    setlocale(LC_ALL, "");// Adapta as saídas do console ao nosso alfabeto
    // Cria um pipe nomeado
    hNamedPipe2 = CreateNamedPipe(
        TEXT("\\\\.\\pipe\\MyNamedPipe4"),
        PIPE_ACCESS_OUTBOUND,
        PIPE_TYPE_BYTE | PIPE_READMODE_BYTE | PIPE_WAIT,
        1,
        1024 * 16,
        1024 * 16,
        NMPWAIT_USE_DEFAULT_WAIT,
        NULL);

    if (hNamedPipe2 == INVALID_HANDLE_VALUE) {
        std::cerr << "Erro ao criar pipe nomeado: " << GetLastError() << std::endl;
        return;
    }


    system("start cmd.exe /K \"more < \\\\.\\pipe\\MyNamedPipe4\"");

    // Aguarda a conexão ao pipe
    if (!ConnectNamedPipe(hNamedPipe2, NULL)) {
        if (GetLastError() != ERROR_PIPE_CONNECTED) {
            std::cerr << "Erro ao conectar no pipe nomeado: " << GetLastError() << std::endl
            CloseHandle(hNamedPipe2);
            return;
        }
    }
}

void ConsoleAlarmes() {
    setlocale(LC_ALL, "");// Adapta as saídas do console ao nosso alfabeto
    // Cria um pipe nomeado
    hNamedPipe = CreateNamedPipe(
        TEXT("\\\\.\\pipe\\MyNamedPipe3"),
        PIPE_ACCESS_OUTBOUND,
        PIPE_TYPE_BYTE | PIPE_READMODE_BYTE | PIPE_WAIT,
        1,
        1024 * 16,
        1024 * 16,
        NMPWAIT_USE_DEFAULT_WAIT,
        NULL);
```

```
    if (hNamedPipe == INVALID_HANDLE_VALUE) {
        std::cerr << "Erro ao criar pipe nomeado: " << GetLastError() << std::endl;
        return;
    }


    system("start cmd.exe /K \"more < \\\\.\\pipe\\MyNamedPipe3\"");

    // Aguarda a conexão ao pipe
    if (!ConnectNamedPipe(hNamedPipe, NULL)) {
        if (GetLastError() != ERROR_PIPE_CONNECTED) {
            std::cerr << "Erro ao conectar no pipe nomeado: " << GetLastError() << std::endl
            CloseHandle(hNamedPipe);
            return;
        }
    }
}
```

Estas funções inicializam pipes nomeados, que são canais de comunicação que permitem a transferência de dados entre processos. A utilização de pipes nomeados é uma solução eficaz para o encaminhamento de mensagens, pois permite comunicações entre diferentes processos do sistema operacional de forma segura e confiável.

4.3 Escrita de Dados nos Pipes Nomeados

As funções `PrintProcesso` e `PrintAlarme` escrevem mensagens nos pipes nomeados correspondentes, que serão exibidos nas janelas de console.

```
void PrintProcesso(const std::string& message) {
    if (hNamedPipe2 == INVALID_HANDLE_VALUE) {
        std::cerr << "Pipe não está pronto para escrita." << std::endl;
        return;
    }

    DWORD bytesWritten;
    WriteFile(hNamedPipe2, message.c_str(), message.size(), &bytesWritten, NULL);

}

void PrintAlarme(const std::string& message) {
    if (hNamedPipe == INVALID_HANDLE_VALUE) {
        std::cerr << "Pipe não está pronto para escrita." << std::endl;
        return;
    }
```

```
    DWORD bytesWritten;
    WriteFile(hNamedPipe, message.c_str(), message.size(), &bytesWritten, NULL);

}
```

5 Funções de Mensageria

As funções de mensageria desempenham um papel vital no software de automação, gerando e processando mensagens que simulam dados de processos industriais e alarmes.

5.1 Geração de Mensagens de Alarme

A função `geraMensagemAlarme` é responsável por criar mensagens de alarme simuladas. A aleatoriedade é introduzida para simular a ocorrência de eventos imprevisíveis.

```
std::string geraMensagemAlarme(int& counter) {
    counter++;

    if (counter > 99999) {
        counter = 1;
    }

    // Convertendo o contador para string e preenchendo com zeros à esquerda.
    std::string countStr = std::to_string(counter);
    countStr = std::string(5 - countStr.length(), '0') + countStr;

    std::string randomString = countStr + ";"; // Incluindo o contador e um ponto e vírgula.

    // Gerando um número inteiro aleatório de dois dígitos.
    int randomNum = rand() % 100;
    randomString += (randomNum < 10) ? "0" + std::to_string(randomNum) : std::to_string(rand
    randomString += ";"; // Adicionando um ponto e vírgula após o número inteiro.

    // Adicionando o horário atual no final da string.
    randomString += getCurrentTime();

    // O comprimento total da string será 40 caracteres.
    return randomString;
}
```

5.2 Uso de Sementes Aleatórias Distintas

A fim de garantir que as mensagens geradas sejam distintas e simulem com precisão um ambiente real, sementes aleatórias diferentes são usadas para cada

thread de leitura dos CLPs. Isso é feito para evitar a geração de sequências idênticas de números pseudoaleatórios, o que poderia comprometer a integridade da simulação.

A função `generateRandomReal` utiliza a biblioteca de números aleatórios moderna do C++ para gerar valores de ponto flutuante que simulam leituras de sensores.

```
std::string generateRandomReal() {
    // Inicializa std::random_device e std::mt19937 para obter números aleatórios de alta qu
    std::random_device rd;
    std::mt19937 gen(rd()); // Usa std::random_device para gerar uma semente aleatória

    // Configura a distribuição para números entre 0.0 e 10000.0
    std::uniform_real_distribution<float> dist(0.0f, 10000.0f);

    // Gera o número aleatório real
    float randomReal = dist(gen);

    // Converte o número para string com uma casa decimal
    std::stringstream ss;
    ss << std::fixed << std::setprecision(1) << randomReal;
    return ss.str();
}
```

5.3 Processamento de Mensagens

As funções `processMessage` e `processHardwareFailureMessage` lidam com o processamento das mensagens recebidas, extraindo informações relevantes e reformatando-as para exibição.

```
std::string processMessage(const std::string& inputMessage) {
    std::istringstream iss(inputMessage);
    std::string nseq, id, horario;

    // Divide a string de entrada em NSEQ, ID e HORARIO
    std::getline(iss, nseq, ';');
    std::getline(iss, id, ';');
    std::getline(iss, horario);

    // Obtém o texto do alarme com base no ID
    std::string alarmText = getAlarmText(id);

    // Constrói a mensagem de saída
    std::string outputMessage = horario + " NSEQ: " + nseq + " ID: " + id + " " + alarmText;

    return outputMessage;
```

```
}

std::string processHardwareFailureMessage(const std::string& inputMessage) {
    std::istringstream iss(inputMessage);
    std::string nseq, clpNum, horario;
    std::string dummy;  // para campos que não serão usados

    // Divide a string de entrada
    std::getline(iss, nseq, ';');      // NSEQ
    std::getline(iss, clpNum, ';');    // Número do CLP
    // Pulando os próximos três campos (valores que não são usados)
    for (int i = 0; i < 4; ++i) {
        std::getline(iss, dummy, ';');
    }
    std::getline(iss, horario);  // HORARIO

    // Constrói a mensagem de saída
    std::string outputMessage = horario + " NSEQ: " + nseq + " FALHA DE HARDWARE CLP No. " +

    return outputMessage;
}
```

5.4 Mapeamento de Textos de Alarme

A função `getAlarmText` mapeia códigos de alarme para descrições textuais, permitindo que os operadores entendam rapidamente a natureza de um alarme.

```
std::string getAlarmText(const std::string& id) {
    static std::map<std::string, std::string> alarmTexts = {
    {"00", "Falha no sensor de temperatura do reator"},
    {"01", "Falha no sistema de injecao de calcario"},
    {"02", "Temperatura da escoria fora dos limites"},
    {"03", "Baixa eficiencia de remocao de enxofre"},
    {"04", "Nivel de oxigenio no gas de saida fora dos limites"},
    {"05", "Falha no sistema de recirculacao de gas"},
    {"06", "Pressao anormal no reator"},
    {"07", "Falha no sistema de controle de fluxo de gas"},
    {"08", "Vazamento detectado no sistema de injecao"},
    {"09", "Temperatura do gusa acima do limite seguro"},
    {"10", "Interferencia no sistema de medicao de fluxo"},
    {"11", "Falha no sistema de alimentacao de energia"},
    {"12", "Sensor de pH fora de calibracao"},
    {"13", "Deteccao de vazamento de gas toxico"},
    {"14", "Sobrecarga no motor do agitador"},
    {"15", "Falha no sistema de refrigeracao do reator"},
    {"16", "Desbalanceamento no misturador de reagentes"},
```

```
{"17", "Sensor de nivel de liquido defeituoso"},
{"18", "Falha no sistema de exaustao"},
{"19", "Contaminacao do material de injecao detectada"},
{"20", "Falha no sistema de aquecimento auxiliar"},
{"21", "Erro no controlador logico programavel (PLC)"},
{"22", "Falha no mecanismo de descarga de escoria"},
{"23", "Inconsistencia nos dados de monitoramento de qualidade"},
{"24", "Sinal de alarme do detector de metais"},
{"25", "Variacao anormal na composicao do gas"},
{"26", "Desvio na temperatura do condensador"},
{"27", "Anomalia detectada no fluxo de agua de resfriamento"},
{"28", "Falha no sensor de pressao atmosferica"},
{"29", "Vibracao excessiva detectada no compressor"},
{"30", "Falha na valvula de controle de gas"},
{"31", "Sistema de filtragem de ar obstruido"},
{"32", "Baixa qualidade do calcario detectada"},
{"33", "Sobrecarga no sistema de controle de emissoes"},
{"34", "Falha no sistema de medicao de massa"},
{"35", "Detector de chama desativado"},
{"36", "Falha na bomba de lama de calcario"},
{"37", "Desalinhamento no transportador"},
{"38", "Falha no sistema de isolamento termico"},
{"39", "Interferencia no sistema de comunicacao"},
{"40", "Sensor de umidade fora do limite operacional"},
{"41", "Anomalia na composicao quimica do reagente"},
{"42", "Falha no mecanismo de travamento de seguranca"},
{"43", "Fluxo anormal no duto de gas de saida"},
{"44", "Aumento inesperado na concentracao de CO2"},
{"45", "Sistema de alimentacao de reagente interrompido"},
{"46", "Falha na calibracao do sensor de CO"},
{"47", "Temperatura do forno alem do limite maximo"},
{"48", "Inconsistencia nos parametros do processo"},
{"49", "Falha no sistema de controle de oxidacao"},
{"50", "Falha no equipamento de medicao de densidade"},
{"51", "Desvio na pressao do sistema de vapor"},
{"52", "Vazamento de oleo no sistema hidraulico"},
{"53", "Falha no motor do sistema de agitacao"},
{"54", "Anomalia no circuito de controle principal"},
{"55", "Falha no sensor de nivel de liquido refrigerante"},
{"56", "Falha no sistema de recuperacao de calor"},
{"57", "Sensor de fluxo de ar com leitura irregular"},
{"58", "Sobreaquecimento detectado no trocador de calor"},
{"59", "Falha no sistema de controle de pressao do gas"},
{"60", "Erro no sistema de balanceamento de carga"},
{"61", "Inconsistencia na leitura do medidor de vazao"},
{"62", "Falha no sistema de ignicao do forno"},
```

```
{"63", "Sobretensao detectada no sistema eletrico"},
{"64", "Falha no sistema de monitoramento de pH"},
{"65", "Baixa eficiencia do filtro de particulas"},
{"66", "Falha no sistema de deteccao de chamas"},
{"67", "Desgaste anormal detectado nas bombas"},
{"68", "Falha no sistema de medicao de condutividade"},
{"69", "Bloqueio no sistema de distribuicao de reagentes"},
{"70", "Sinal de alerta do sistema de seguranca"},
{"71", "Falha no sensor de deteccao de corrosao"},
{"72", "Falha no sistema de controle de velocidade do ventilador"},
{"73", "Inconsistencia na leitura do sensor de temperatura"},
{"74", "Anomalia na bomba de circulacao de gas"},
{"75", "Falha no sistema de controle de emissoes de enxofre"},
{"76", "Falha no sistema de controle de turbidez"},
{"77", "Vazamento no sistema de refrigeracao"},
{"78", "Falha no sistema de medicao de pressao"},
{"79", "Falha no sensor de deteccao de metais pesados"},
{"80", "Falha na valvula de controle de ar"},
{"81", "Erro no sistema de controle automatico"},
{"82", "Falha no sistema de aquecimento de emergencia"},
{"83", "Sobrecarga no sistema de alimentacao de reagentes"},
{"84", "Falha no sistema de monitoramento de gases"},
{"85", "Falha no sistema de controle de drenagem"},
{"86", "Falha no sensor de qualidade do ar"},
{"87", "Inconsistencia no sistema de medicao de peso"},
{"88", "Falha na valvula de controle de liquidos"},
{"89", "Falha no sistema de deteccao de vazamento de gas"},
{"90", "Falha no sistema de bombeamento de reagentes"},
{"91", "Falha no sistema de controle de temperatura"},
{"92", "Erro de comunicacao com o controlador logico"},
{"93", "Falha no sistema de analise de gas de combustao"},
{"94", "Falha no sistema de controle de condensacao"},
{"95", "Falha no sensor de pressao do gas de saida"},
{"96", "Falha no sistema de monitoramento de vibracao"},
{"97", "Falha no sistema de controle de fluxo de calcario"},
{"98", "Falha no sistema de monitoramento de oxidacao"},
{"99", "Falha na bomba de recirculacao de agua"}
};



auto it = alarmTexts.find(id);
if (it != alarmTexts.end()) {
    return it->second;
}
```

```
    return "Texto de alarme desconhecido";
}
```

6 Thread de Leitura de Dados de Processo dos CLPs

A thread `Thread_CLP_Processo` é responsável pela leitura dos dados de processo dos CLPs. Ela utiliza um temporizador waitable para controlar a frequência de leitura e assegura que os dados sejam lidos em um intervalo regular de 500ms.

```
unsigned  WINAPI Thread_CLP_Processo(LPVOID param) {
    std::random_device rd;

    ThreadParam* p = (ThreadParam*)param;
    BufferCircular* buffer = p->buffer;
    char identificador = p->identificador;
    int& counter = p->counter;

    // Cria um temporizador waitable.
    HANDLE hTimer = CreateWaitableTimer(NULL, TRUE, NULL);
    if (hTimer == NULL) {
        // Tratar erro de criação do temporizador
        return 1;
    }

    LARGE_INTEGER liDueTime;
    liDueTime.QuadPart = -5000000LL;  // 500 ms em unidades de 100 nanossegundos

    while (true) {


        if (WaitForSingleObject(hFinalizaTudo, 0) == WAIT_OBJECT_0) {
            break;  // Encerra a thread
        }

        if (identificador == '1') {
            WaitForSingleObject(hControlaLeituraPLC1, INFINITE);
        }
        else {
            WaitForSingleObject(hControlaLeituraPLC2, INFINITE);
        }

        if (!SetWaitableTimer(hTimer, &liDueTime, 0, NULL, NULL, 0)) {
            // Tratar erro na configuração do temporizador
            return 2;
```

```
        }

        // Espera pelo temporizador
        WaitForSingleObject(hTimer, INFINITE);

        if (!buffer->estaCheio()) {
            std::string dado = geraMensagemProcesso(counter, identificador);  // Passa count
            buffer->adicionarDado(dado);
            SetEvent(dadoProduzido);
        }
        else {
            WaitForSingleObject(espacoDisponivel, INFINITE);
        }
    }

    CloseHandle(hTimer);
    _endthreadex(0);
    return 0;
}..
}
```

6.1 Estruturas de Exclusão Mútua e Sincronização

Durante a operação, esta thread utiliza eventos e estruturas de exclusão mútua para garantir que os dados sejam acessados e modificados de forma segura e coordenada, evitando condições de corrida e garantindo a integridade dos dados em um ambiente multithread.

7 Thread de Leitura de Alarmes dos CLPs

A thread `Thread_CLP_Alarmes` gerencia a leitura e o envio de alarmes. Ela cria um pipe nomeado para IPC, que permite a comunicação segura e isolada com a tarefa de exibição de alarmes.

```
unsigned WINAPI Thread_CLP_Alarmes(void* parametro) {
    HANDLE hConsole = GetStdHandle(STD_OUTPUT_HANDLE); // Handle para o console
    // Abre ou cria o pipe nomeado
    HANDLE hPipe = CreateNamedPipe(
        TEXT("\\\\.\\pipe\\AlarmPipe"),   // nome do pipe
        PIPE_ACCESS_OUTBOUND,             // escrita apenas no lado do servidor
        PIPE_TYPE_MESSAGE |               // dados transmitidos como mensagens
        PIPE_READMODE_MESSAGE |           // dados lidos como mensagens
        PIPE_WAIT,                        // operações de bloqueio
        1,                                // número máximo de instâncias
        0,                                // tamanho do buffer de saída (0 usa padrão)
```

```
    0,                                  // tamanho do buffer de entrada (0 usa padrão)
    NMPWAIT_USE_DEFAULT_WAIT,           // tempo de espera
    NULL);                              // atributos de segurança padrão

if (hPipe == INVALID_HANDLE_VALUE) {
    printf("Erro: Falha ao criar o pipe 1. Código: %lu\n", GetLastError());
    return 1;
}

EnterCriticalSection(&consoleSection);
SetConsoleTextAttribute(hConsole, FOREGROUND_GREEN);
printf("Thread_CLP_Alarmes: Pipe 1 criado com SUCESSO e esperando por conexão...\n");
SetConsoleTextAttribute(hConsole, FOREGROUND_RED | FOREGROUND_GREEN | FOREGROUND_BLUE);
LeaveCriticalSection(&consoleSection);
;

BOOL fConnected = ConnectNamedPipe(hPipe, NULL) ?
    TRUE : (GetLastError() == ERROR_PIPE_CONNECTED);

if (!fConnected) {
    printf("Erro: Falha ao conectar no pipe 1. Código: %lu\n", GetLastError());
    CloseHandle(hPipe);
    return 2;
}

EnterCriticalSection(&consoleSection);
SetConsoleTextAttribute(hConsole, FOREGROUND_GREEN);
printf("Thread_CLP_Alarmes: Conectado ao pipe 1 com SUCESSO.\n");
SetConsoleTextAttribute(hConsole, FOREGROUND_RED | FOREGROUND_GREEN | FOREGROUND_BLUE);
LeaveCriticalSection(&consoleSection);



// Cria um temporizador waitable.
HANDLE hTimer = CreateWaitableTimer(NULL, FALSE, NULL);
if (hTimer == NULL) {
    printf("Erro: Falha ao criar o temporizador waitable. Código: %lu\n", GetLastError()
    CloseHandle(hPipe);
    return 3;
}

int contador = 0;
std::random_device rd;
std::mt19937 gen(rd());
std::uniform_int_distribution<> dis(1, 5); // Distribuição para intervalos de 1 a 5 segu
```

```
while (true) {

    if (WaitForSingleObject(hFinalizaTudo, 0) == WAIT_OBJECT_0) {
        break;  // Encerra a thread
    }

    WaitForSingleObject(hControlaLeituraAlarmesPLC, INFINITE);


    // Define o intervalo aleatório
    int intervalo = dis(gen);
    LARGE_INTEGER liDueTime;
    liDueTime.QuadPart = static_cast<LONGLONG>(-10000000) * intervalo; // Intervalo em u

    // Configura o temporizador
    if (!SetWaitableTimer(hTimer, &liDueTime, 0, NULL, NULL, 0)) {
        printf("Erro: Falha ao configurar o temporizador waitable. Código: %lu\n", GetLa
        break;
    }

    // Espera pelo temporizador
    WaitForSingleObject(hTimer, INFINITE);

    // Gera e envia a mensagem de alarme
    std::string dado = geraMensagemAlarme(contador); // Gera a mensagem de alarme
    DWORD bytesWritten;


    BOOL fSuccess = WriteFile(
        hPipe,                          // handle para o pipe
        dado.c_str(),                   // buffer para escrever
        (DWORD)dado.size(),             // número de bytes a escrever
        &bytesWritten,                  // número de bytes escritos
        NULL);                          // não usando overlapped I/O

    if (!fSuccess || dado.size() != bytesWritten) {
        printf("Erro: Falha ao escrever no pipe 1. Código: %lu\n", GetLastError());
        break;
    }
}

printf("Thread_CLP_Alarmes: Encerrando thread e fechando recursos.\n");
CloseHandle(hTimer);
CloseHandle(hPipe);
_endthreadex(0);
return 0;
```

```
}
```

7.1 Comunicação Interprocessos (IPC)

A escolha de usar pipes nomeados para enviar dados para a tarefa de exibição de alarmes é devido à sua eficiência e confiabilidade em ambientes Windows. O IPC por meio de pipes nomeados fornece um canal direto e seguro para a transferência de mensagens entre processos, que é ideal para atualizações em tempo real necessárias em sistemas de automação.

7.2 Mecanismos de Sincronização

Ambas as threads utilizam mecanismos de sincronização, como eventos (WaitForSingleObject) e seções críticas (EnterCriticalSection), para gerenciar o acesso a recursos compartilhados e coordenar a execução. Nesse caso especificamente, os buffers são estruturas de dados compartilhadas por diferentes threads, assim como o console principal e outros recursos compartilhados.

8 Thread de Retirada de Mensagens

A `Thread_Retirada` executa uma função crítica no sistema de automação, atuando como um consumidor de mensagens do buffer de dados de processo e um distribuidor para o segundo buffer ou para a exibição de alarmes, baseando-se no conteúdo específico das mensagens.

```
unsigned WINAPI Thread_Retirada(void* parametro) {
    Thread_Retirada_Params* params = static_cast<Thread_Retirada_Params*>(parametro);
    BufferCircular* buffer1 = params->buffer1;
    BufferCircular* buffer2 = params->buffer2;
    HANDLE hConsole = GetStdHandle(STD_OUTPUT_HANDLE);
    std::string diag;

    EnterCriticalSection(&consoleSection);
    SetConsoleTextAttribute(hConsole, FOREGROUND_RED | FOREGROUND_GREEN | FOREGROUND_BLUE);
    printf("Thread_Retirada: Criando o pipe 2 para envio de alarmes de Falha de Hardware...\
    LeaveCriticalSection(&consoleSection);

    HANDLE hPipe2 = CreateNamedPipe(
        TEXT("\\\\.\\pipe\\AlarmPipe2"),   // nome do pipe
        PIPE_ACCESS_OUTBOUND,             // escrita apenas no lado do servidor
        PIPE_TYPE_MESSAGE |               // dados transmitidos como mensagens
        PIPE_READMODE_MESSAGE |           // dados lidos como mensagens
        PIPE_WAIT,                        // operações de bloqueio
        1,                                // número máximo de instâncias
        0,                                // tamanho do buffer de saída (0 usa padrão)
        0,                                // tamanho do buffer de entrada (0 usa padrão)
```

```
    NMPWAIT_USE_DEFAULT_WAIT,         // tempo de espera
    NULL);                            // atributos de segurança padrão

if (hPipe2 == INVALID_HANDLE_VALUE) {
    printf("Erro: Falha ao criar o pipe 2 para envio de alarmes de Falha de Hardware. Có
    return 1;
}

EnterCriticalSection(&consoleSection);
SetConsoleTextAttribute(hConsole, FOREGROUND_GREEN);
printf("Thread_Retirada: pipe 2 para envio de alarmes de Falha de Hardware criado com SU
SetConsoleTextAttribute(hConsole, FOREGROUND_RED | FOREGROUND_GREEN | FOREGROUND_BLUE);
LeaveCriticalSection(&consoleSection);


DWORD waitResult;
HANDLE eventos[] = { dadoProduzido, eventoDeSaida };

while (true) {

    if (WaitForSingleObject(hFinalizaTudo, 0) == WAIT_OBJECT_0) {
        break;  // Encerra a thread
    }

    WaitForSingleObject(hControlaThreadRetirada, INFINITE);

    waitResult = WaitForMultipleObjects(2, eventos, FALSE, INFINITE);

    if (waitResult - WAIT_OBJECT_0 == 1) {
        printf("Thread_Retirada: Evento de saída recebido, encerrando thread.\n");
        break;
    }

    std::string dado;
    if (buffer1->removerDado(dado)) {
        printf("Thread_Retirada: Dado lido do buffer1 -> %s.\n", dado.c_str());
        diag = dado.substr(8, 2);

        if (diag == "55") {

            DWORD bytesWritten;
            WriteFile(hPipe2, dado.c_str(), (DWORD)dado.size(), &bytesWritten, NULL);

        }
        else {
            if (!buffer2->estaCheio()) {
```

```
                    buffer2->adicionarDado(dado);
                }
                else {
                    printf("Thread_Retirada: Buffer2 cheio, aguardando espaço disponível.\n"
                    WaitForSingleObject(espacoDisponivelBuffer2, INFINITE);
                    buffer2->adicionarDado(dado);
                    printf("Thread_Retirada: Dado adicionado ao buffer2 após espera.\n");
                }
            }
            if (!buffer1->estaCheio()) {
                SetEvent(espacoDisponivel);
            }
        }
    }

    CloseHandle(hPipe2);
    CloseHandle(hConsole);
    _endthreadex(0);
    return 0;
}
```

8.1 Processamento e Distribuição de Mensagens

Ao receber uma mensagem, a thread verifica o campo DIAG e decide o destino da mensagem. Mensagens com DIAG "55" são enviadas diretamente para a exibição de alarmes, enquanto outras são armazenadas no segundo buffer.

Para mensagens de alarme com DIAG "55", que significam falha no cartão de I/O do PLC, a thread usa um pipe nomeado para enviar esses dados diretamente para a tarefa de exibição de alarmes.

Demais mensagens são de processo e são depositadas no segundo buffer e posteriormente serão coletadas pela tarefa de exibição de dados de processo.

9 Threads de Exibição e a Interface de Usuário Industrial

Em um ambiente industrial, a interface de usuário (UI) desempenha um papel crucial na monitoração e resposta rápida a condições de processo críticas. As threads `Thread_Exibicao_Alarmes` e `Thread_Exibicao_Dados_Proceso` são componentes chave desta UI, permitindo que os operadores recebam atualizações de alarmes e dados de processo em tempo real.

9.1 Thread de Exibição de Alarmes

```
unsigned WINAPI Thread_Exibicao_Alarmes(void* parametro) {
```

```
ConsoleAlarmes();
HANDLE hConsole = GetStdHandle(STD_OUTPUT_HANDLE);

HANDLE hPipe = CreateFile(
    TEXT("\\\\.\\pipe\\AlarmPipe"),
    GENERIC_READ,
    0,
    NULL,
    OPEN_EXISTING,
    0,
    NULL);

if (hPipe == INVALID_HANDLE_VALUE) {
    PrintAlarme("Erro: Falha ao abrir o pipe. Código: %lu\n");
    return 1;
}

HANDLE hPipe2 = CreateFile(
    TEXT("\\\\.\\pipe\\AlarmPipe2"),
    GENERIC_READ,
    0,
    NULL,
    OPEN_EXISTING,
    0,
    NULL);

if (hPipe2 == INVALID_HANDLE_VALUE) {
    PrintAlarme("Erro: Falha ao abrir o pipe 2 para conexão com a Thread de Retirada. Có
    return 1;
}

char buffer[128];
DWORD bytesRead;
DWORD bytesAvailable;

while (true) {
    if (WaitForSingleObject(hFinalizaTudo, 0) == WAIT_OBJECT_0) {
        break;  // Encerra a thread
    }

    WaitForSingleObject(hControlaDisplayAlarme, INFINITE);

    if (PeekNamedPipe(hPipe, NULL, 0, NULL, &bytesAvailable, NULL) && bytesAvailable > (
        if (ReadFile(hPipe, buffer, sizeof(buffer), &bytesRead, NULL)) {
            std::string message(buffer, bytesRead);
            std::string alarmMessage = processMessage(message) + "\n";
```

```
                PrintAlarme(alarmMessage);
            }
        }

        if (PeekNamedPipe(hPipe2, NULL, 0, NULL, &bytesAvailable, NULL) && bytesAvailable >
            if (ReadFile(hPipe2, buffer, sizeof(buffer), &bytesRead, NULL)) {
                std::string message(buffer, bytesRead);
                std::string alarmMessage = processHardwareFailureMessage(message) + "\n";
                PrintAlarme(alarmMessage);

            }
        }
    }

    PrintAlarme("Thread_Exibicao_Alarmes: Encerrando thread e fechando os pipes.\n");
    CloseHandle(hPipe);
    CloseHandle(hPipe2);

    CloseHandle(hNamedPipe);
    _endthreadex(0);
    return 0;
}
```

A `Thread_Exibicao_Alarmes` lida com a apresentação de alertas críticos ao operador. Ela lê de pipes nomeados, que são canais de comunicação estabelecidos para transmitir dados entre processos distintos, e exibe mensagens de alarme formatadas na UI.

9.2 Thread de Exibição de Dados de Processo

```
unsigned WINAPI Thread_Exibicao_Dados_Processo(void* parametro){
    ConsoleProcesso();
    BufferCircular* buffer2 = static_cast<BufferCircular*>(parametro);


    while (true) {
        if (WaitForSingleObject(hFinalizaTudo, 0) == WAIT_OBJECT_0) {
            break;  // Encerra a thread
        }

        WaitForSingleObject(hControlaDisplayProcesso, INFINITE);

        WaitForSingleObject(espacoDisponivelBuffer2, INFINITE); // Aguarda até que um dado s

        std::string dado;
        if (buffer2->removerDado(dado)) { // Tenta remover um dado do buffer2
```

```
            std::string processMessage = processaMensagemProcesso(dado) + "\n";
            PrintProcesso(processMessage);
        }

        // Verifica se é necessário enviar um sinal para a Thread_Retirada
        if (!buffer2->estaCheio()) {
            SetEvent(espacoDisponivelBuffer2);
        }


    }


    _endthreadex(0);
    return 0;
}
```

A `Thread_Exibicao_Dados_Proceso` é responsável por exibir dados de processo, como temperaturas, pressões e outras leituras de sensores vitais. Ela consome dados da segunda lista circular em memória, processa-os e os apresenta em uma janela de console dedicada para dados de processo.

9.3 Múltiplas Janelas para Diferentes Tipos de Dados

A utilização de múltiplas janelas para exibir diferentes tipos de dados é uma prática importante na UI industrial. Ela permite que os operadores se concentrem em tipos específicos de informação sem distração, melhorando a capacidade de resposta a eventos críticos.

10 Thread de Leitura do Teclado

A `Thread_Leitura_Teclado` permite que o usuário do sistema de automação controle dinamicamente o estado de outras threads por meio de interações no teclado.

```
unsigned WINAPI Thread_Leitura_Teclado(void* parametro) {

    HANDLE hEvents[] = { hFinalizaTudo }; // Array de eventos para esperar, inclui apenas o
    DWORD dwEvent;

    while (true) {
        int key = _getch();  // Bloqueia e espera uma tecla ser pressionada

        switch (key) {
        case '1':
            ToggleEvent("Leitura do CLP 1",hControlaLeituraPLC1);
```

```
            break;
        case '2':
            ToggleEvent("Leitura do CLP 2", hControlaLeituraPLC2);
            break;
        case 'm':
            ToggleEvent("Leitura de Alarmes dos PLCs", hControlaLeituraAlarmesPLC);
            break;
        case 'r':
            ToggleEvent("Retirada de Mensagens", hControlaThreadRetirada);
            break;
        case 'p':
            ToggleEvent("Exibição de Dados do Processo", hControlaDisplayProcesso);
            break;
        case 'a':
            ToggleEvent("Exibição de Alarmes", hControlaDisplayAlarme);
            break;
        case 27: // ESC key
            SetEvent(hFinalizaTudo); // Sinaliza para encerrar todas as tarefas
            return 0; // Encerra a thread
        default:
            // Código para outras teclas, se necessário
            break;
        }
    }
        return 0; // Retorna para encerrar a thread
    }
```

10.1 Função ToggleEvent

A função `ToggleEvent` é chamada pela thread de leitura do teclado para alternar o estado de execução das threads. Ela utiliza eventos de sincronização para ativar ou desativar a execução de threads específicas.

10.2 Mecanismo de Bloqueio e Desbloqueio de Threads

Ao pressionar uma tecla específica, a função `ToggleEvent` é chamada e verifica o estado do evento associado à thread correspondente. Se o evento estiver sinalizado, a função o reseta, pausando a execução da thread. Se não estiver sinalizado, a função sinaliza o evento, permitindo que a thread continue ou retome a execução.

10.3 Encerramento da Aplicação

Quando a tecla ESC é pressionada, a `Thread_Leitura_Teclado` sinaliza o evento `hFinalizaTudo`, que é verificado por todas as threads do sistema. Este evento instrui todas as threads a encerrarem suas execuções de maneira ordenada.

11 Função Main

A função `main` é o coração do sistema de automação industrial, coordenando a inicialização, execução e terminação de todas as threads e processos.

11.1 Inicialização do Ambiente

Inicialmente, a `main` configura o ambiente, criando eventos de sincronização que serão usados para controlar o fluxo de execução das threads. Também são criadas seções críticas para proteger a escrita concorrente no console.

11.2 Criação de Threads e Eventos

A função `main` prossegue para criar threads responsáveis pela leitura de dados dos CLPs, leitura de alarmes, retirada de mensagens, exibição de dados de processo e alarmes, e leitura do teclado. Cada thread é projetada para uma função específica, garantindo o paralelismo e a eficiência do sistema.

11.3 Gerenciamento do Ciclo de Vida da Aplicação

A aplicação aguarda eventos específicos do teclado para controlar a execução das threads. Por exemplo, a tecla ESC é configurada para sinalizar a finalização de todas as threads e a consequente terminação do programa.

11.4 Encerramento Ordenado

Quando instruída a encerrar, a `main` garante que todas as threads finalizem suas execuções de forma ordenada antes de liberar todos os recursos do sistema, como eventos e seções críticas, e finalmente terminar a execução do programa.

Conclusão

A jornada através do vasto universo da automação em tempo real e das aplicações soft-real time, particularmente no contexto da indústria siderúrgica e do processo de dessulfuração do ferro-gusa, é uma que ecoa o espírito pioneiro de Nikola Tesla e a sagacidade encontrada nas páginas do "Guia do Mochileiro das Galáxias". "Se alguma vez descobrirmos como a energia cósmica funciona," disse Tesla, "então o tempo fará parte do passado." É com essa energia cósmica que as indústrias siderúrgicas buscam sincronizar suas operações em tempo real, assegurando a qualidade do aço e a segurança dos processos.

Como Douglas Adams habilmente nos lembra, "Nada viaja mais rápido do que a luz com algo a dizer." Em nossas fábricas modernas, os dados viajam à velocidade da luz de sensores para controladores, e a capacidade de processá-los de forma eficiente, sem o luxo do tempo, é o que define as aplicações soft-real time. Não é apenas sobre fazer as coisas rapidamente, mas fazer com que "a resposta esteja lá, antes que a pergunta seja completamente formulada" - uma

filosofia que guia as operações de dessulfuração que garantem a integridade do metal que forma a espinha dorsal de nossa civilização.

Encerramos esta documentação não com um adeus, mas com a expectativa de um futuro onde a integração da automação em tempo real é tão universal quanto as leis da física, um futuro onde, como Adams disse, "Não entre em pânico" é um conselho não apenas para viajantes intergalácticos, mas também para engenheiros e operadores que confiam em sistemas de automação sofisticados para manter as engrenagens da indústria siderúrgica girando de maneira limpa, eficiente e ininterrupta.