TRABALHO FINAL – PARTE 1
CONTEXTO EM AUTOMAÇÃO INDUSTRIAL

Bernardo Fonseca Maia - 2021013779
Marcela Fontes Abreu – 2018013798

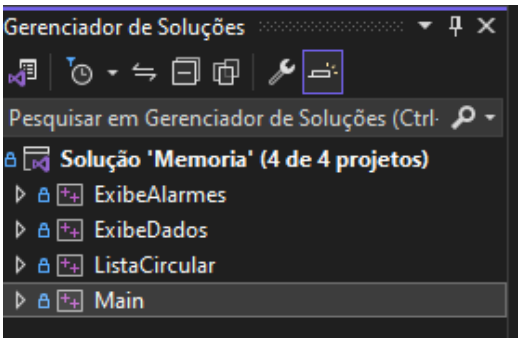
Organização

O programa possui 4 processos, que executam uma ou mais tarefas:

- Main:
  1. Tarefa de leitura do teclado: Tratamento aos comandos do operador obtidos pelo terminal principal.
- Lista Circular:
  2. Tarefa de leitura do sistema de pesagem;
  3. Tarefa de leitura do CLP;
  4. Tarefa de captura de alarmes;
  5. Tarefa de captura de dados do CLP;
- Exibe Alarme
  6. Tarefa de exibição de alarmes;
- Exibe Dados
  7. Tarefa de exibição de dados do processo.

Solução Visual Studio

O arquivo possui uma única solução “Memoria” com 4 projetos, 1 para cada processo.



Main.cpp

O processo Main tem duas principais funções:

- Criar os demais processos;
- Ativar/Desativar as tarefas com base nas interações entre usuário e teclado.

O processo “ListaCircular” foi criado com a flag de criação 0 pois não possui console próprio e utiliza o do processo pai.

```
//ListaCircular
status = CreateProcess(
    "..\\x64\\Debug\\ListaCircular.exe",        // Caminho do arquivo executável
    NULL,                   // Apontador p/ parâmetros de linha de comando
    NULL,                   // Apontador p/ descritor de segurança
    NULL,                   // Idem, threads do processo
    FALSE,                  // Herança de handles
    0,                      // Flags de criação
    NULL,                   // Herança do ambiente de execução
    "..\\x64\\Debug",       // Diretório do arquivo executável
    &si,                    // lpStartUpInfo
    &NewProcess[0]);        // lpProcessInformation

if (!status)
    std::cerr << "Erro na abertura cmd teclado = " << GetLastError() << "\n";
```

Os processos ExibeAlarme e ExibeDados possuem terminais próprios, portanto foram criados com flag de criação “CREATE_NEW_CONSOLE”:

```
//Criação de Console de exibição de Alarmes
status = CreateProcess(
    "..\\x64\\Debug\\ExibeAlarmes.exe",         // Caminho do arquivo executável
    NULL,                   // Apontador p/ parâmetros de linha de comando
    NULL,                   // Apontador p/ descritor de segurança
    NULL,                   // Idem, threads do processo
    FALSE,                  // Herança de handles
    CREATE_NEW_CONSOLE,     // Flags de criação
    NULL,                   // Herança do ambiente de execução
    "..\\x64\\Debug",       // Diretório do arquivo executável
    &si,                    // lpStartUpInfo
    &NewProcess[1]);        // lpProcessInformation

if (!status)
    std::cerr << "Erro na abertura Exibe Alarmes = " << GetLastError() << "\n";

//Criação de Console de exibição de Dados
status = CreateProcess(
    "..\\x64\\Debug\\ExibeDados.exe",         // Caminho do arquivo executável
    NULL,                   // Apontador p/ parâmetros de linha de comando
    NULL,                   // Apontador p/ descritor de segurança
    NULL,                   // Idem, threads do processo
    FALSE,                  // Herança de handles
    CREATE_NEW_CONSOLE,     // Flags de criação
    NULL,                   // Herança do ambiente de execução
    "..\\x64\\Debug",               // Diretório do arquivo executável
    &si,                    // lpStartUpInfo
    &NewProcess[2]);        // lpProcessInformation

if (!status)
    std::cerr << "Erro na abertura Exibe Dados = " << GetLastError() << "\n";
```

Para aguardar que todos os processos sejam devidamente criados antes de prosseguir com o programa, é utilizado o seguinte comando:

WaitForMultipleObjects (3, &NewProcess->hProcess, TRUE, INFINITE);

Em seguida é o trecho responsável pela interação com o teclado. Primeiramente é criado 1 evento para cada comando existente. Esses eventos são todos de reset automático e iniciam não sinalizados, com exceção do evento hEventESC que possui reset manual:

```
//Cria os eventos que acordam as threads
hEventA = CreateEvent(NULL, FALSE, FALSE, "CapturaAlarmes");  //Reset automático e inicializa não-sinalizado
hEventB = CreateEvent(NULL, FALSE, FALSE, "Pesagem");
hEventC = CreateEvent(NULL, FALSE, FALSE, "LeituraCLP");
hEventD = CreateEvent(NULL, FALSE, FALSE, "CapturaDados");
hEvent1 = CreateEvent(NULL, FALSE, FALSE, "Alarme");
hEvent2 = CreateEvent(NULL, FALSE, FALSE, "Dados");
hEventESC = CreateEvent(NULL, TRUE, FALSE, "ESC");          //Reset manual e inicializa não-sinalizado
```

Em seguida, as opções são impressas na tela, o valor digitado é lido e associado à variável *action*, que por sua vez é avaliada em uma função *Switch.* Para cada possível entrada, o evento correspondente é acionado por meio de SetEvent():

```
do {
    action = _getch();
    std::cout << "\nFoi selecionado: " << action << std::endl;

    switch (action) {
    case '1':
        SetEvent(hEvent1);
        break;
    case '2':
        SetEvent(hEvent2);
        break;
    case 'a':
        SetEvent(hEventA);
        break;
    case 'b':
        SetEvent(hEventB);
        break;
    case 'c':
        SetEvent(hEventC);
        break;
    case 'd':
        SetEvent(hEventD);
        break;
    default:
            break;
    }
} while (action != ESC);

SetEvent(hEventESC);
```

Essa captura está dentro de um loop infinito que é quebrado quando ESC é selecionado.

```
CloseHandle(hEventA);
CloseHandle(hEventB);
CloseHandle(hEventC);
CloseHandle(hEventD);
CloseHandle(hEvent1);
CloseHandle(hEvent2);
CloseHandle(hEventESC);

// Fechar handles dos processos
for (int i = 0; i < 3; i++) {

    CloseHandle(NewProcess[i].hProcess);
}

return EXIT_SUCCESS;
```

A lógica do processo finaliza com o fechamento dos handles utilizados.

ExibeAlarme e ExibeDados

Esses processos possuem uma lógica idêntica. São compostos por uma thread principal e uma thread secundária. A Thread principal apenas inicializa a secundária:

```
int main()
{
    SetConsoleTitle("Console Alarmes");
    HANDLE hThread;
    DWORD dwThreadId;
    int i = 0;
    hThread = (HANDLE)_beginthreadex(
        NULL,
        0,
        (CAST_FUNCTION)ThreadFunc,
        (LPVOID)i,
        0,
        (CAST_LPDWORD)&dwThreadId
    );

    DWORD ret;
    HANDLE hEvents[2] = { hEvent, hEventESC };
```

E controla o estado de bloqueio da thread secundária:

```
DWORD ret;
HANDLE hEvents[2] = { hEvent, hEventESC };

while (1) {
    ret = WaitForMultipleObjects(2, hEvents, FALSE, INFINITE);
    int i = ret - WAIT_OBJECT_0;
    if (ret == 0) {
        if (estado == 0) {
            estado = 1;
            SetEvent(hInterruptor);
            cout << "\nTarefa desbloqueada\n";
        }
        else {
            estado = 0;
            ResetEvent(hInterruptor);
            cout << "\nTarefa bloqueada\n";
        }
    }
    else { ESC = 1; break; }
}
```

Para isso, três eventos são utilizados: hEvent, hEventESC e hInterruptor.

O evento hEvent é o mesmo que hEvent1 (No caso de alarme) ou hEvent2 (No caso de Dados). Ele é acionado quando as teclas 1 ou 2 são acionadas no teclado. (*Switch* do processo *Main*). hEventESC é sinalizado quando a tecla ESC é acionada.

É utilizado também uma variável booleana que funciona como um interruptor. Quando hEvent for acionado, essa variável inverte de estado. Se for TRUE, se torna FALSE, e se for FALSE, se torna TRUE.

Além disso, sempre que a variável booleana se torna verdadeira, o evento hInterruptor (reset manual) é sinalizado, e quando a variável se torna falsa, o evento passa a ser não sinalizado.

Por fim, quando a tecla ESC é acionada no teclado, a thread primária finaliza fechando todos os handles utilizados.

A thread secundária executa, paralela à thread principal, a seguinte função:

```
DWORD WINAPI ThreadFunc(LPVOID index)
{
    while (!ESC)
    {
        WaitForSingleObject(hInterruptor, INFINITE);
        cout << "\nEXIBE ALARMES\n";
        Sleep(500);
    }
    return(0);
}
```

Enquanto hInterruptor está sinalizado, ela fica executando seu código de exibição de alarmes, para a primeira etapa do trabalho simplificamos esse código para apenas a impressão da mensagem "EXIBE ALARMES" no console do processo.

Quando hInterruptor deixar de ser sinalizado, a thread é bloqueada.

ListaCircular

O processo "lista circular" possui 6 threads secundárias, 5 são para executar as tarefas 1 a 4, onde a tarefa de simulação de dados e alarmes foi dividida em duas threads por causa da diferente periodização das mensagens produzidas, onde nessa primeira etapa foi utilizada a função *Sleep()*. As threads ThreadCLPdado e ThreadAlarmes são as tarefas 3 e 4 respectivamente.

Nessa primeira etapa, as tarefas deveriam manipular a lista de forma a inserir dados e retirá-los para exibir os dados futuramente nas telas de exibição.

Logo, foram feitas duas threads "produtoras", que alimentavam a lista circular principal, e duas threads "consumidoras", onde foi criada mais uma thread auxiliar para separar as mensagens de alarme e as de dados do CLP:

```
//Tarefa de leitura do sistema de pesagem
hThreadPesagem = (HANDLE)_beginthreadex(
    NULL,
    0,
    (CAST_FUNCTION)FuncPesagem,
    (LPVOID)Pesagem,
    0,
    (CAST_LPDWORD)&dwThreadId
);
if (hThreadPesagem) printf("Thread criada Id= %0x \n", dwThreadId);

//Tarefa de leitura do CLP
hThreadCLP = (HANDLE)_beginthreadex(
    NULL,
    0,
    (CAST_FUNCTION)FuncCLPalarme,
    (LPVOID)CLP,
    0,
    (CAST_LPDWORD)&dwThreadId
);
if (hThreadCLP) printf("Thread criada Id= %0x \n", dwThreadId);
```

Threads produtoras.

```
//Tarefa de leitura do CLP
hThreadCLPdado = (HANDLE)_beginthreadex(
    NULL,
    0,
    (CAST_FUNCTION)FuncCLPdado,
    (LPVOID)CLPdado,
    0,
    (CAST_LPDWORD)&dwThreadId
);
if (hThreadCLP) printf("Thread criada Id= %0x \n", dwThreadId);

//Tarefa de captura de alarmes
hThreadAlarme = (HANDLE)_beginthreadex(
    NULL,
    0,
    (CAST_FUNCTION)FuncAlarme,  // casting necessário
    (LPVOID)Alarme,
    0,
    (CAST_LPDWORD)&dwThreadId   // cating necessário
);
if (hThreadAlarme) printf("Thread criada Id= %0x \n", dwThreadId);
```

Threads consumidoras.

Além das threads criadas para o funcionamento do processo, foi necessária a criação de várias funções para manipulação dos dados da lista, funções essas visando facilitar o entendimento do funcionamento das threads. Uma melhoria será refatorar o código, fazendo funções mais gerais para facilitar o entendimento do programa nesse arquivo específico.

A interação entre as tarefas da lista circular e o teclado segue uma lógica similar às dos processos de exibição de alarme e de dados. Porém, por serem 5 threads executando simultaneamente no mesmo processo, foi necessário o uso de vetores. O vetor booleano Interruptores é o responsável por controlar o estado atual de cada tarefa, sendo que o último elemento do vetor foi (Interruptores[4]) foi utilizado como sendo o indicador de que ESC foi selecionado.

```
HANDLE hEvents[5] = { hEventA, hEventB, hEventC, hEventD, hEventESC };
DWORD Ret;
while (1) {
    Ret = WaitForMultipleObjects(5, hEvents, FALSE, INFINITE); //Espera qualquer evento
    int i = Ret - WAIT_OBJECT_0;
    if (i != 4) {
        if (Interruptores[i] == 0) {
            Interruptores[i] = 1;
            SetEvent(hInts[i]);
            cout << "\nTAREFA " << i << " DESBLOQUEADA\n";
        }
        else {
            Interruptores[i] = 0;
            ResetEvent(hInts[i]);
            cout << "\nTAREFA " << i << " BLOQUEADA\n";
        }
    }
    else {
        Interruptores[4] = 1;
        cout << "\n ESC reconhecido \n";
        for (int j = 0; j < 4; j++) { //Reseta todos os interruptores
            ResetEvent(hInts[j]);
        }
        WaitForMultipleObjects(4, hInts, TRUE, INFINITE);
        break;
    }
}
```

Da mesma forma que os outros processos, a thread principal fica em um looping infinito aguardando os eventos acionados pelo teclado. Portando, é utilizado um *WaitForMultipleObjects()* que bloqueia a thread e, assim que um dos eventos do vetor de eventos do teclado (*hEvents)* for acionado, ela prossegue seu funcionamento identificando qual foi o evento. Ao subtrair o retorno da função *WaitForMultipleObjects()* com WAIT_OBJECT_0, é possível definir o índice do evento que foi acionado. Então, é feito a função que dá nome ao Interruptor, caso o booleano referente àquele evento seja verdadeiro, ele se torna falso, se falso, se torna verdadeiro. Além disso, foi criado um novo vetor de eventos (*hInts)*, só que de reset manual, que são responsáveis por bloquear e desbloquear as threads. Esses eventos são sinalizados quando o respectivo interruptor se torna verdadeiro, e deixam de ser sinalizados quando o interruptor se torna falso.

As funções das threads, como exemplificado a seguir, começam um *WaitForSingleObject(),* o que as bloqueia até que o evento dela seja sinalizado, e como elas estão em looping infinito, sempre que vão repetir o processo elas primeiro testam se o evento continua sinalizado, caso ele tenha sido resetado, a thread é bloqueada até que o evento seja sinalizado novamente.

```
DWORD WINAPI FuncAlarme(LPVOID id)
{
    string teste;
    do {
        WaitForSingleObject(hInts[0], INFINITE); //Bloqueia se interruptor não sinalizado
        WaitForSingleObject(hMutexA, INFINITE);
        if (!isemptyA() && !Interruptores[4]) {
            showTopA();
            cout << "\nEXIBE ALARME: " << topoA << "\n\n";
            popA();
        }
        ReleaseMutex(hMutexA);
        Sleep(1000);
    } while (!Interruptores[4]);
    _endthreadex(0);
    return(0);
}
```

Divisão de tarefas da dupla

A parte inicial do trabalho de esboçar como seriam as divisões de threads e processos, e a criação dos processos foi feita em conjunto. A dupla se encontrou presencialmente para resolver essas questões.

Uma vez como isso resolvido, fizemos a escolha de duas partes que poderiam ser divididas inicialmente. Como as tarefas que usam a lista precisam estar em um mesmo processo, para facilidade de manipulação, ListaCircular e suas 4 tarefas foi responsabilidade da Marcela. As demais tarefas e a sincronização de todos os processos e threads ficou a cargo do Bernardo. O trabalho foi feito utilizando o GitHub, o que nos permitiu trabalhar nos diferentes processos simultaneamente à distância. Apesar da divisão de responsabilidade, os testes foram feitos em conjunto e cada membro da dupla interferia no trabalho do outro membro quando percebia algum problema ou para ajudar quando o outro estava com dificuldade.