Trabalho Prático – Automação em Tempo Real

Parte 1

Alunos:

João Pedro Bazotte Corgozinho 2016109313

Luís Gustavo Pinto da Silva 2017050401

Metodologia

Nosso trabalho consiste em um compilado de projetos, que serão tradados como módulos. Esses módulos serão discutidos a seguir.

Módulo 1

O primeiro modulo, o ProjetoAtr, é responsável pela criação de três processos diferentes, com auxílio da função **CreateProcess()**. Foram criados três processos, pois serão necessárias três janelas de console independentes.

A função **CreateProcess()** retorna um valor booleano, caso sua execução for bem-sucedida ela retorna o valor **TRUE**. Diante disso, usamos uma estrutura condicional com a função **CreateProcess()** para avaliar o retorno da função, caso o retorno seja **FALSE** quer dizer que a criação do processo falhou, logo será printado na tela uma mensagem de aviso. Para fazer a verificação de erro, foi utilizada a função **GetLastError()**, que irá retornar o código referente ao ultimo erro. Pensamos em utilizar a biblioteca **"CheckForError.h"**, porém não obtivemos sucesso, então optamos para utilizar a função **GetLastError()** para o resto do trabalho prático.

Para a criação dos novos consoles foi utilizado a flag **CREATE_NEW_CONSOLE** e foi passada como parâmetro para a função **CreateProcess()**. Foi necessário a utilização da letra **"L"** como casting antes do diretório do executável do projeto. Para boa prática de programação, fechamos todos os handles utilizados ao final do código.

Módulo 2

Nesse módulo discutiremos sobre o Projeto1, que é o grande corpo do nosso trabalho. Esse processo será responsável pela criação das tarefas leitura do SDCD, leitura do PIMS, captura de dados do processo, captura de alarmes e das funções e as funções auxiliares criadas.

Começamos o código com o casting necessário para a criação das threads e algumas definições globais que irão nos ajudar a simplificar o entendimento do código. A seguir criamos nossa variável compartilhada, denominada **buffer**. Ela é um vetor de strings(ponteiros para caracteres) de tamanho **MAX_MENSAGENS**. Essa variável vai se comportar como uma lista circular de memória, por isso declaramos as variáveis **P_livre** , **P_ocupado_dado, P_ocupado_alarme** que são inicializadas em 0, pois a lista começa vazia. Usamos como base o problema dos produtores e consumidores. Vimos no material didático da disciplina a resolução para um consumidor e um produtor, nesse trabalho temos duas threads produtoras e duas consumidoras para um mesmo recurso compartilhado. Diante disso, usamos um mutex para resolver esse problema.

```
char* buffer[MAX_MENSAGENS];
int P_livre, P_ocupado_dado, P_ocupado_alarme = 0;
```

Logo depois, declaramos os handles que serão utilizados. A imagem a seguir mostra o nome do handle e ao seu lado direito um comentário que especifica sua função no código.

```
HANDLE hmutex;                  // Handle proteção Plivre, Pocupado
HANDLE hlivre;                  // Handle semaforo contador de posiçoes livre buffer
HANDLE hocupado;                // Handle semaforo binario de posiçoes livre buffer
HANDLE sEvent;                  // Handle para evento s
HANDLE cEvent;                  // Handle para evento c
HANDLE dEvent;                  // Handle para evento d
HANDLE pEvent;                  // Handle para evento p
HANDLE aEvent;                  // Handle para evento a
HANDLE oEvent;                  // Handle para evento o
HANDLE escEvent;                // Handle para evento ESC
HANDLE Print;                   // Handle para o mutex que protege a impressao da tela
HANDLE hOut;                    // Handle para a saida do console
```

A seguir são feitas as declarações das funções auxiliares e as threads que serão explicadas com mais detalhes mais a frente da documentação.

Nessa parte do código começa o **main**. No inicio alocamos cada posição da nossa lista circular com auxilio de um **for**. A seguir inicializamos os semáforos de acordo com a imagem a seguir:

```
//Inicializaçao semforos
hmutex = CreateSemaphore(NULL, 1, 1, L"mutex");
if (hmutex == NULL) {
    printf("Erro na criação do mutex: %d\n", GetLastError());
}
hlivre = CreateSemaphore(NULL, 100, 100, L"P_livre");
if (hlivre == NULL) {
    printf("Erro na criação do semaforo hlivre: %d\n", GetLastError());
}
hocupado = CreateSemaphore(NULL, 0, 100, L"P_ocupado");
if (hocupado == NULL) {
    printf("Erro na criação do semaforo hocupado: %d\n", GetLastError());
}
Print = CreateSemaphore(NULL, 1, 1, L"Print_mutex");
if (Print == NULL) {
    printf("Erro na criação do mutex para os prints: %d\n", GetLastError());
}
```

O semáforo **hmutex** foi inicializado com o valor 1 e com um valor máximo de 1, pois ele se comporta como um mutex, ou seja, a primeira thread que solicitar esse mutex deverá consegui-lo, por isso ele é inicializado com 1, caso contrario a aplicação iria travar quando esse semáforo fosse solicitado, pois a thread seria bloqueada eternamente. O semáforo **hlivre** é um semáforo contador que inicia com o valor de 100, pois no início teremos 100 posições livres no nosso buffer e tem um valor máximo de 100 pelo fato de termos 100 posições no buffer. O semáforo **hocupado** é um semáforo contador que inicia com o valor de 0, pois no início não temos nenhuma posição ocupada no nosso buffer, e tem valor máximo de 100 pelo fato de podermos ter 100 posições ocupadas no buffer(o buffer estaria cheio). O semáforo **Print** também se comporta como um mutex, e pelo mesmo motivo do **hmutex** ele começa com 1 e tem valor máximo igual a 1.

A função **CreateSemaphore()** retorna o valor de **NULL** caso ela tenha dado algum erro. Com a utilização da **GetLastError()** ,caso haja um problema na criação do semáforo, ela imprimirá na tela o código de erro referente ao erro acontecido.

Declaramos um handle para a saída do console que se chama **hOut**, ele irá nos auxiliar nas cores da saída do console. Logo após declaramos os eventos. Todos os eventos são de reset automático, ou seja, quando eles são acionados todas as threads que os esperam são desbloqueadas. Quando a função **CreateEvent()** quando falha retorna o valor de **NULL**, usando uma estrutura condicional, quando esse valor de **NULL** é recebido usamos a função **GetLastError()** para imprimir o código de erro.

```
// Obtendo handle saida console
hOut = GetStdHandle(STD_OUTPUT_HANDLE);
if (hOut == INVALID_HANDLE_VALUE)
    printf("Erro ao obter handle para a saída da console\n");

// Inicializando os eventos de sincronização
sEvent = CreateEvent(NULL, FALSE, FALSE, L"sEvento");
if (sEvent == NULL) {
    printf("Falha ao criar o evento sEvent: %d\n", GetLastError());
}
cEvent = CreateEvent(NULL, FALSE, FALSE, L"cEvento");
if (cEvent == NULL) {
    printf("Falha ao criar o evento cEvent: %d\n", GetLastError());
}
dEvent = CreateEvent(NULL, FALSE, FALSE, L"dEvento");
if (dEvent == NULL) {
    printf("Falha ao criar o evento dEvent: %d\n", GetLastError());
}
pEvent = CreateEvent(NULL, FALSE, FALSE, L"pEvento");
if (pEvent == NULL) {
    printf("Falha ao criar o evento pEvent: %d\n", GetLastError());
}
aEvent = CreateEvent(NULL, FALSE, FALSE, L"aEvento");
if (aEvent == NULL) {
    printf("Falha ao criar o evento aEvent: %d\n", GetLastError());
}
oEvent = CreateEvent(NULL, FALSE, FALSE, L"oEvento");
if (oEvent == NULL) {
    printf("Falha ao criar o evento oEvent: %d\n", GetLastError());
}
escEvent = CreateEvent(NULL, TRUE, FALSE, L"escEvento");
if (escEvent == NULL) {
    printf("Falha ao criar o evento escEvent: %d\n", GetLastError());
}
```

A partir dessa parte criamos nossas threads. São 4 threads, cada uma com um escopo diferente. A seguir mostraremos como criamos nossa primeira thread:

```
//Criando Threads
hThreads[0] = (HANDLE)_beginthreadex(
    NULL,
    0,
    (CAST_FUNCTION)tarefa_leitura_do_SDCD,
    (LPVOID)0,
    0,
    (CAST_LPDWORD)&dwIdThreads[0]);
if (hThreads[0])
    printf("Thread leitura SDCD! Id=%0x\n", dwIdThreads[0]);
else {
    printf("Erro na criacao da thread leitura SDCD! N = %d Erro = %d\n", i, errno);
    exit(0);
}
```

Para a criação das outras 3 threads, o único parâmetro mudado foi o identificador da rotina a ser iniciada, pois, como supracitado, cada thread desse trabalho tem funções diferentes.

Para o tratamento de dados oriundos do teclado foi criado um **do while** que termina quando a tecla **ESC** é digitada. Para armazenar a entrada do teclado, utilizamos

a variável **nTecla** e a função _**getch()**. Dentro do **do while** temos um **switch case** como o mostrado a seguir:

```
switch (nTecla) {
case s:
    if (SetEvent(sEvent) == NULL) {
        printf("Erro ao sinalizar o evento s\n");
    }
    dwStatus = WaitForSingleObject(Print, INFINITE);
    if (dwStatus == WAIT_FAILED) {
        printf("Falha na funcao WaitForSingleObject: %d\n", GetLastError());
    }
    SetConsoleTextAttribute(hOut, 11);
    printf("Evento sEvent sinalizado: Bloqueando/desbloqueando a thread leitura do SDCD\n");
    SetConsoleTextAttribute(hOut, WHITE);
    dwStatus = ReleaseSemaphore(Print, 1, NULL);
    if (dwStatus == NULL) {
        printf("Falha na funcao ReleaseSemaphore: %d\n", GetLastError());
    }
    break;
```

O exemplo acima é para o caso de a tecla **s** ser digitada, caso ela for digitada ela entra nesse **case** e da um **SetEvent()** no evento correspondente. Quando o evento for acionado ele imprime na tela em **azul** qual evento foi acionado.

Para o caso da tecla **ESC** não há impressão na tela, o código somente da um **SetEvent()** (para as outras threads/processos que estão esperando poderem ser encerradas) e depois sai do laço do **do while**.

Uma informação importante, a letra digitada deve ser minúscula!!!

Logo depois, o nosso código **main** espera a finalização das threads criadas com a função **WaitForMultipleObjects()**.

```
// Aguarda término das threads homens e mulheres
dwRet = WaitForMultipleObjects(4, hThreads, TRUE, INFINITE);
if (dwRet == WAIT_FAILED) {
    printf("Falha na funcao WaitForMultipleObjects:%d\n", GetLastError());
}
```

Caso a função **WaitForMultipleObjects()** falhar sua execução ela retorna um valor igual a **WAIT_FAILED** o que faz com que ele entre no **if** e com auxilio da função **GetLastError()** será printado na tela o código de erro que aconteceu.

Logo depois, todos os handles são fechados e o buffer é desalocado posição a posição com ajuda de um **for.**

Thread leitura do SDCD/ Thread leitura do PIMS

Nos escolhemos explicar essas duas threads juntas, pois sua logica é bem semelhante e só se diferenciam em poucas coisas que serão explicadas no decorrer da documentação.

Essas threads começam com um **do while** para que elas executem ate que a tecla **ESC** for digitada. Primeiramente esperamos 1 segundo com a função **Sleep()**,logo após ela a thread aguarda a sinalização de um entre três eventos com auxílio da função

WaitForMultipleObjects()(o evento que bloqueia/desbloqueia a thread, o **ESCevent** e o **hlivre**). Como a thread começa desbloqueada, caso o evento de bloqueio seja sinalizado (**nTipoEvento == 0**) uma mensagem aparecerá mostrando que a thread foi bloqueada. A partir dai ela esperará o evento que bloqueia/desbloqueia a thread ou o evento para finalizar a execução com a função **WaitForMultipleObjects()**.

Caso na primeira espera o evento sinalizado seja o **hlivre**, ela entra no **else if** (**nTipoEvento == 2**) e continuará sua execução. Nesse momento a thread espera o semáforo binário **hmutex** ser sinalizado, caso ele seja sinalizado o **hmutex** é conquistado. Já que o **hlivre** foi sinalizado, sabemos que há uma posição livre no buffer, então verificamos dentro de um **while** uma condição se os quatro primeiros dígitos são "NSEQ"(isso indica uma mensagem de dado do processo) ou "HORA"(isso indica um alarme), caso os 4 primeiros dígitos forem uma dessas duas sequencias, quer dizer que esse lugar do **buffer** já está preenchido. Repetimos esse loop ate que achemos a próxima posição livre usando a atualização da posição **P_livre**, como na imagem abaixo:

```
while (strncmp(buffer[P_livre], "HORA", 4) == 0 || strncmp(buffer[P_livre], "NSEQ", 4) == 0) {
    P_livre = (P_livre + 1) % MAX_MENSAGENS;
}
```

Colocamos na posição **P_livre** do buffer uma mensagem aleatória gerada pelas funções **gerar_msg_SDCD(NSEQ)/ gerar_msg_PIMS(NSEQ)**. Logo depois de sair da sua zona critica, fazemos a operação casada do **hmutex** dando um **ReleaseSemaphore()** e sinalizamos o semáforo contador **hocupado** ,indicando que uma posição do buffer foi ocupada. Depois atualizamos o **NSEQ**.

Caso o **ESCEvent** seja sinalizado, esteja a thread bloqueada ou não, ocorre um **break** no **do while** exibindo uma mensagem de finalização na tela, e logo após chamando a função **endthreadex()** para finalizar a thread. Foi passado 0 como parâmetro para a função **endthreadex()** para indicar que finalização ocorreu de maneira bem sucedida.

Essa thread não imprime as mensagens geradas, ela somente gera de forma automática e aleatória e as deposita na lista circular.

Thread captura de dados de processo/ Thread captura de alarmes

Nós escolhemos explicar essas duas threads juntas, pois sua lógica é bem semelhante e só se diferenciam em poucas coisas que serão explicadas no decorrer da documentação.

Essas threads começam com um **do while** para que elas executem ate que a tecla **ESC** for digitada. Primeiramente esperamos 1 segundo com a função **Sleep()**,logo após ela a thread aguarda a sinalização de um entre três eventos com auxílio da função **WaitForMultipleObjects()**(o evento que bloqueia/desbloqueia a thread, o **ESCevent** e o **hocupado**).O **hocupado** começa com o valor do contador igual a 0, pois a lista esta inicialmente vazia. Quando as threads de leitura SDCD/PIMS geram os dados/alarmes

que serão impressos, eles sinalizam o semáforo contador **hocupado**. Quando o **hocupado** está sinalizado quer dizer que podemos retirar alguma informação do **buffer** e imprimir na tela, pois há algum alarme/dado. Caso o evento de bloqueio seja sinalizado (**nTipoEvento == 0**) uma mensagem aparecerá mostrando que a thread foi bloqueada. A partir daí ela esperará o evento que bloqueia/desbloqueia a thread ou o evento para finalizar a execução com a função **WaitForMultipleObjects()**.

Caso na primeira espera o evento sinalizado seja o **hocupado**, ela entra no **else** (**nTipoEvento == 2**) e continuará sua execução. Primeiro aguardamos a sinalização do **hmutex**(conquista do mutex) e em segundo lugar nós atribuímos a uma variável auxiliar de nome **t** o valor do tamanho do **buffer** na posição **P_ocupado_alrme/P_ocupado_dado**. Logo depois, fazemos um **if** para ver se a variável **t** é maior ou igual a 46(no caso de retirar um dado, pois isso sinaliza que é uma mensagem do tipo dado) ou se é menor ou igual a 35(no caso de retirar um alarme, pois isso sinaliza uma mensagem do tipo alarme). Somado a essa comparação, olhamos também se há algum caractere igual a “ç” que é um caractere não utilizado na criação de alarmes/dados e será utilizado como peça chave na logica dessas threads. Caso ele entre no **if,** quer dizer que a o **buffer** na posição verificada é um alarme(no caso da thread de captura de alarmes) ou é um dado(no caso da thread de captura de dados) então ele pode imprimir com o auxilio das funções **imprime_dado_processo() e imprime_alarme()**. Sobre essas funções, a função **imprime_alarme()** verifica se o alarme é critico ou não, caso ele seja critico ele imprime o alarme na cor vermelha, caso não ele imprime na cor verde e a função **imprime_dado_processo()** imprime os dados na cor amarelo. Após imprimir o dado/alarme, atribuímos ao **buffer** nessa posição uma sequencia do caracteres “ç” ,o que faz com que limpemos o **buffer** nessa posição ,ou seja, é, na nossa lógica, um espaço vazio para gravação de mensagens. A seguir atualizamos a variável **P_ocupado_alarme/P_ocupado_dado** com o método do resto da divisão pelo numero máximo de mensagens, como o mostrado na figura abaixo:

```
int t = strlen(buffer[P_ocupado_alarme]);
    if (t <= 35 && (strstr(buffer[P_ocupado_alarme], "ç") == NULL)) {
        imprime_alarme(buffer[P_ocupado_alarme]);
        memset(buffer[P_ocupado_alarme], 'ç', 34);
        P_ocupado_alarme = (P_ocupado_alarme + 1) % MAX_MENSAGENS;
        status = ReleaseSemaphore(hlivre, 1, NULL);
    if (status == NULL) {
        printf("Falha na funcao ReleaseSemaphore: %d\n", GetLastError());
    }
```

e logo depois damos **ReleaseSemaphore()** no **hmutex**(liberação do mutex).

Caso ele não entre no **if** quer dizer que a mensagem na posição do **buffer** é um alarme(no caso da captura dado processo) ou é um dado (no caso da captura de alarmes). Para as threads não ficarem travadas nesse ponto, fizemos um esse **else** que atualiza a posição do **buffer** para a próxima, sinaliza a variável **hocupado**(porque ela não retira nenhuma mensagem nessa situação, então temos que devolver mais um na variável **hocupado**) e libera a variável **hmutex**.

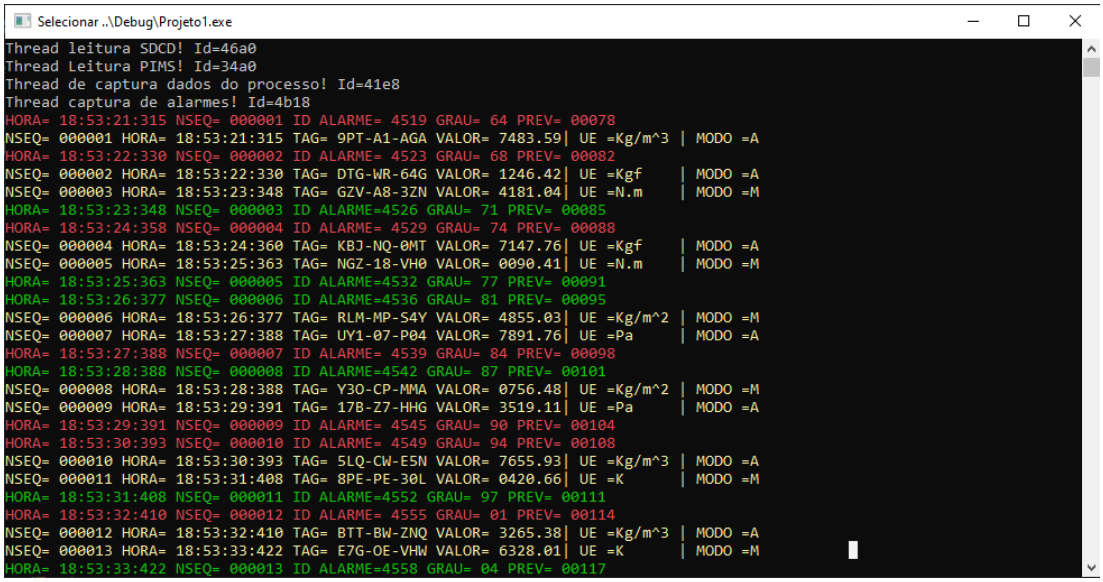
Caso a variável **nTipoEvento == 1** quer dizer que a tecla **ESC** foi digitada, o que faz com que a thread saia do **do while** e termine sua execução.

MODULO 3

Nesse modulo, temos os processos 2 e 3 que são consoles diferentes que esperam, cada uma, a sinalização de um evento especifico. Cada uma delas tem uma thread que aguarda a sinalização do **ESCevent ou cEvent/oEvent.** Antes das threads serem executadas nos usamos a função **OpenEvent()** para utilizar os eventos **ESCevent e cEvent/oEvent** que foram inicializados no processo 1. Quando digitamos “o” ou “c” no console do processo 1, uma notificação é imprimida no console do processo 2 ou 3(dependendo de qual evento você digitou). Caso seja digitado a tecla **ESC**, a thread termina sua execução.

TESTES

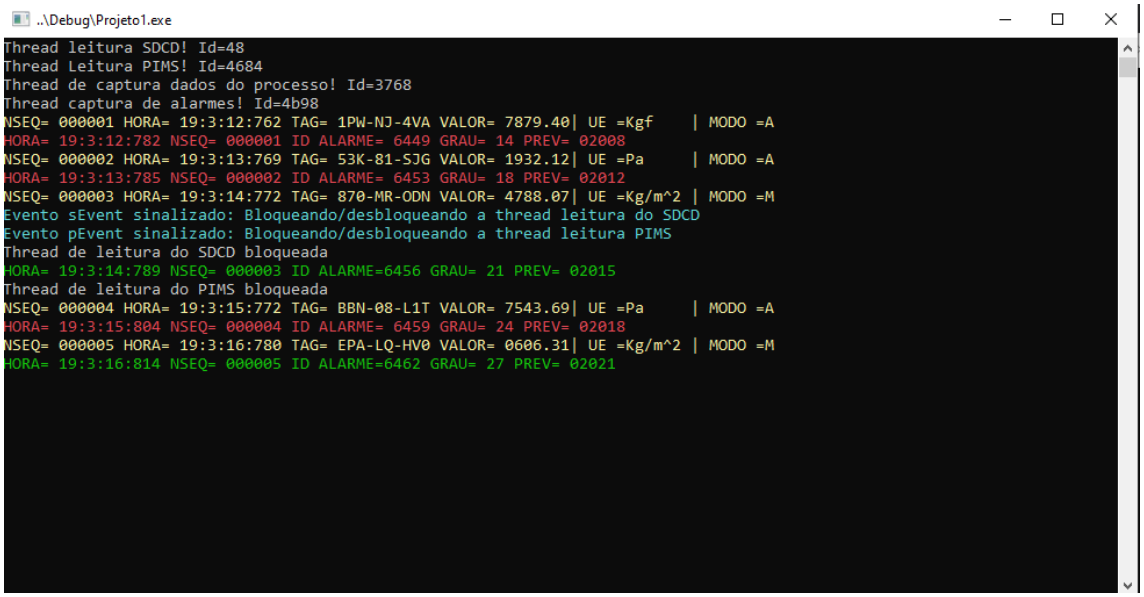
Agora vamos mostrar um pouco da execução do nosso programa. Primeiro vamos executar e deixar um tempo para mostrar o comportamento:



O console do processo 1 é onde estão sendo impressos os dados(mensagens em amarelo), alarmes críticos(mensagens em vermelho) e alarmes não críticos. Nessa situação as threads leitura SDCS e PIMS estão produzindo dados/alarmes a cada 1 segundo e as threads de retirada de alarmes e dados estão retirando essas mensagens na lista.

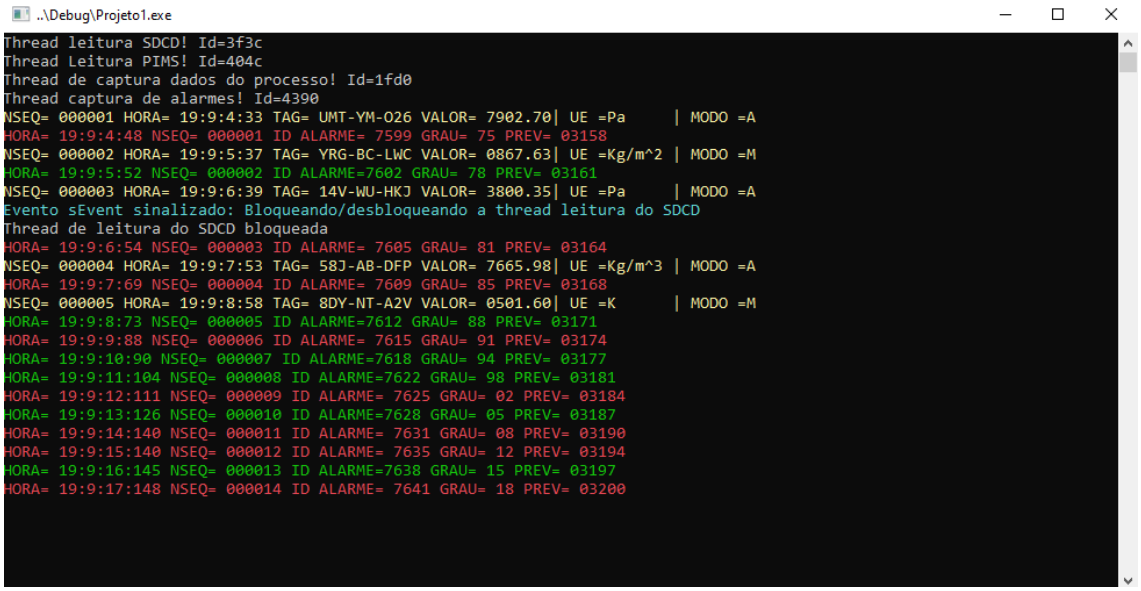
Nessa nova imagem temos o bloqueio das threads que retiram alarmes e dados (apertando as letras “a” e “d”). As threads de leitura SDCD e PIMS continuam produzindo as mensagens.

Nessa próxima imagem as threads de leitura SDCD e PIMS são bloqueadas logo no inicio da execução:

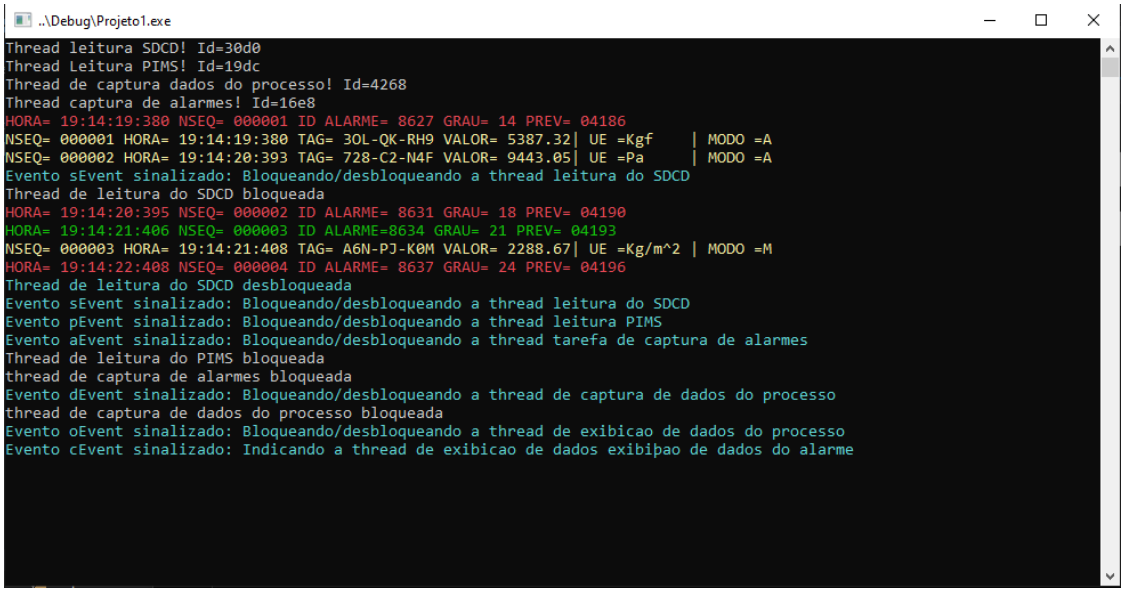


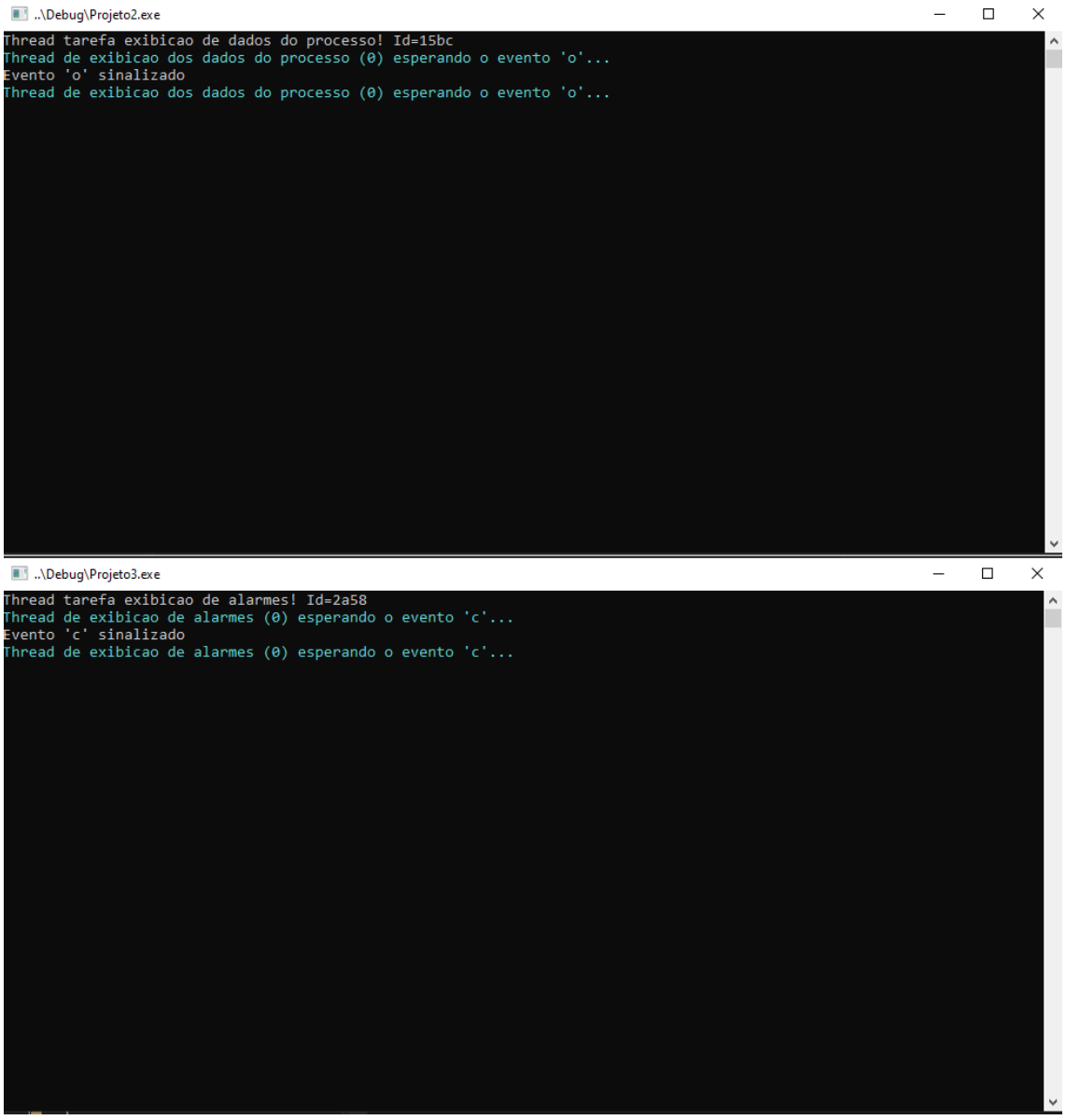
Notamos que após o bloqueio das duas threads supracitadas, não haverá mais mensagem a ser retirada após um tempo, e como as threads de retirada de mensagens precisam da sinalização do **hocupado,** quando o seu contador chegar ao valor de 0, ambas as threads de retirada ficam bloqueadas até a próxima sinalização do **hocupado**.

Nessa próxima imagem bloqueamos a somente a thread de criação de dados(leitura SDCD) depois de um tempo da execução. As outras threads continuam disputando CPU. Depois de imprimir as mensagens de dados feitas antes do bloqueio da thread de leitura SDCD, o console somente imprime mensagens de alarme, como na imagem a seguir:



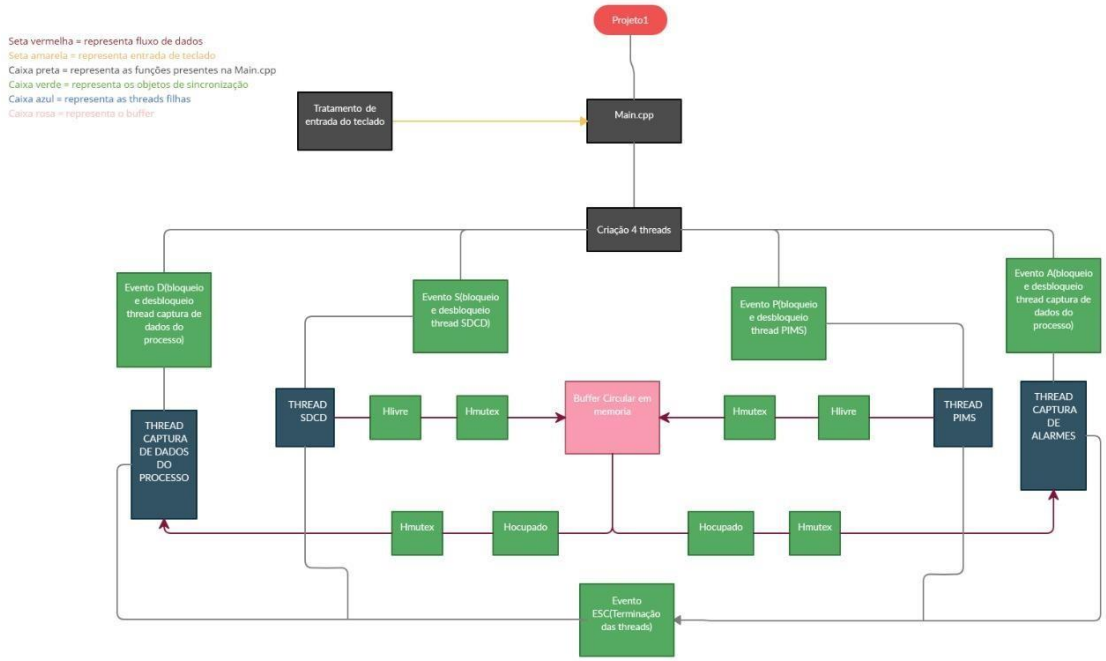
Sobre os processos 2 e 3 mostraremos a sua execução nas próximas imagens. Primeiramente bloqueamos todas as threads do processo 1 para que o print da tela não fique muito poluído. Depois disso apertamos as teclas “c” e “o” para sinalizar os eventos e printar na tela dos processos 2 e 3





FLUXOGRAMA DO FUNCIONAMENTO DO NOSSO TRABALHO

Nota: Caso fique de difícil visualização por causa do tamanho, enviamos na pasta do trabalho o .png do fluxograma



Parte 2 documentação

Na segunda parte do projeto , teriamos que temporizar a tarefa PIMS e a tarefa SDCD , implementar a comunicação entre processos(IPC) e fazer a implementação de um arquivo circular em memoria.Iremos dividir a documentação em três modulos,para facilitar o entendimento da logica da implementação.

Modulo 1 : Temporização dos processos

Começaremos esse modulo com a temporização da tarefa SDCD. Esssa tarefa tem uma temporização de 500 ms. Para implementar essa temporização usamos o **WaitbleTimer** dentro de um laçõ de **do while** como o mostrado na figura a seguir:

```
do {

    if (NSEQ == 1000000) {
        NSEQ = 1;
    }

    Tempo_de_espera.QuadPart = -(500 * 10000);
    //thread se bloqueia esperando o timer
    SetWaitableTimer(htimer, &Tempo_de_espera, 0, NULL, NULL, FALSE); // Inicia o temporizador da thread
    ret = WaitForMultipleObjects(2, Events3, FALSE, INFINITE);
    if (ret == WAIT_FAILED) {
        printf("Falha na funcao WaitForMultipleObjects %d:\n", GetLastError());
    }
```

Declaramos a variavel **Tempo_de_espera.QuadPart** como negativa,pois o tempo é relativo e atribuimos ela a um valor de 500ms(500 pacotes de 10000 nanosegundos).Logo apos usamos a função **SetWaitbleTimer** de disparo unico,mas como ela esta num laço do **do while** ela irá repetir e temporizar de forma periodica, de 500 em 500 ms. Para que haja a temporização precisamos de um **WaitForMultipleObjects**, com os eventos **ESCEvent e htimer**,quando o htimer sinaliza(de 500 em 500 ms) a thread fica desbloqueada e seque sua execução ,caso o htimer nao sinalizar a thread fica bloqueada e quando o ESCevent sinaliza significa que a thread quer terminar sua execução,logo se encerra.

A segunda thread que precisa de temporização é a leitura PIMS, ela tem dois temporizador, uma pra cada tipo de alarme, critico e nao critico.Iremos discutir a seguir a logica de implementação:

```
LARGE_INTEGER Tempo_de_espera, Tempo_de_espera2;
Tempo_de_espera.QuadPart = -(gera_numero_aleatorio(1000, 5000) * 10000);
Tempo_de_espera2.QuadPart = -(gera_numero_aleatorio(3000, 8000) * 10000);
SetWaitableTimer(htimer_nao_critico, &Tempo_de_espera, 0, NULL, NULL, FALSE);
SetWaitableTimer(htimercritico, &Tempo_de_espera2, 0, NULL, NULL, FALSE);


do {

    if (NSEQ == 1000000) {
        NSEQ = 1;
    }

    ret = WaitForMultipleObjects(2, Events3, FALSE, INFINITE);
    if (ret == WAIT_FAILED) {
        printf("Falha na funcao WaitForMultipleObjects %d:\n", GetLastError());
    }
```

Primeiro fazemos uma chamada fora do **do while** dos **WaitbleTimers** dos alarmes não criticos e criticos,pois se o evento de alarme critico sinalizar a gente vai gerar um mensagem de alarme critico e enviamos ela via **mailslot** para o outro processo.Se a mensagem de alarme não critico for escolhida vai gerar uma mensagem e depositar no buffer circular.Uma nota, usamos a funçã **gera_numero_aleatorio**,pois temos uma faixa de valores de tempo que podem ser executados os alrmes.Quando o codigo continua executando,caso tenha escolhido um alarme critico,dentro do **do while** tem um outro **SetWaitbletimer** para setar novamente o cronometro e ter outro valor dentro da faixa decidida na especificação do trabalho, o mesmo acontece com os alarmes não criticos.

MODULO 2

Nesse modulo, iremos falar sobre a IPC mailslot para comunicação entre processo. Usamos o mecanismo de mailslot e que o processo servidor e a tarefa de execução de alarmes cria o mailslot onde utlizamos o evento **mEvento** para indicar a criaçao do mailslolt pelo processo servidor, no caso a tarefa de exibiçao de alarme. Assim as threads de captura de alarmes e de geraçao de alarmes funcionam como clientes, que aguardam a sinalizaçao do evento para inciar a sua execuçao, assim garantindo que o servidor seja estabelecido antes dos processos clientes, enviando mensagem para o servidor, como demonstrado no print abaixo.

```
    hMailslot = CreateMailslot(
        L"\\\\.\\mailslot\\Debug\\CaixaPostal",
        0,
        MAILSLOT_WAIT_FOREVER,
        NULL);

    SetEvent(mEvent);


    WaitForSingleObject(mEvent, INFINITE);
```

Quando o alarme e nao critico, ocorre o deposito da mensagem no buffer circular em memoria para que depois ele seja enviado para o outro processo via mailslot.

MODULO 3

Nesse modulo iremos implementar um arquivo circular em memoria. Para essa etapa do trabalho

criamos um arquivo circular em memoria utilizando um semaforo contador para limitar o espaço no arquivo, com 100 espaços para ocupar. A thread de captura de dados escreve no arquivo, enquanto a thread de exibiçao de dados de processos le o arquivo, assim exibindo as mensagens em seu console. Para voltar para o inicio do arquivo utilizamos a funçao **setfilepointer**.

```
    status = WriteFile(hfile, &dado, sizeof(dado), &dwBytesWritten, NULL);
    status = ReleaseSemaphore(hocupado2, 1, NULL);
    if (status == NULL) {
        printf("Falha na funcao ReleaseSemaphore1: %d\n", GetLastError());
    }
    dwPos = iInput * sizeof(dado);
    SetFilePointer(hfile, dwPos, NULL, FILE_BEGIN);
    iInput++;
    if (iInput > 100) {
        SetFilePointer(hfile, 0, NULL, FILE_BEGIN);
        iInput = 1;
    }
status = ReadFile(hfile, &dado, sizeof(dado), &dwBytesRead, NULL);
imprime_dado_processo(dado);
```