

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 tratados como módulos. Esses módulos serão discutidos a seguir.

Módulo 1

O primeiro módulo, 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 último 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 SDCC, 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 posições livre buffer  
HANDLE hocupado;         // Handle semaforo binario de posições 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 início alocamos cada posição da nossa lista circular com auxílio de um **for**. A seguir inicializamos os semáforos de acordo com a imagem a seguir:

```

//Inicialização 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 saída 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 SD CD! Id=%0x\n", dwIdThreads[0]);
else {
    printf("Erro na criação da thread leitura SD CD! 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 SD CD\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 dá 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 dá 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 auxílio 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 SD CD/ Thread leitura do PIMS

Nos 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 até 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 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 **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 sequências, quer dizer que esse lugar do **buffer** já está preenchido. Repetimos esse loop até 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_SDCC(NSEQ)/ gerar_msg_PIMS(NSEQ)**. Logo depois de sair da sua zona crítica, 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 até 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 está inicialmente vazia. Quando as threads de leitura SDCC/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_alarme/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**.

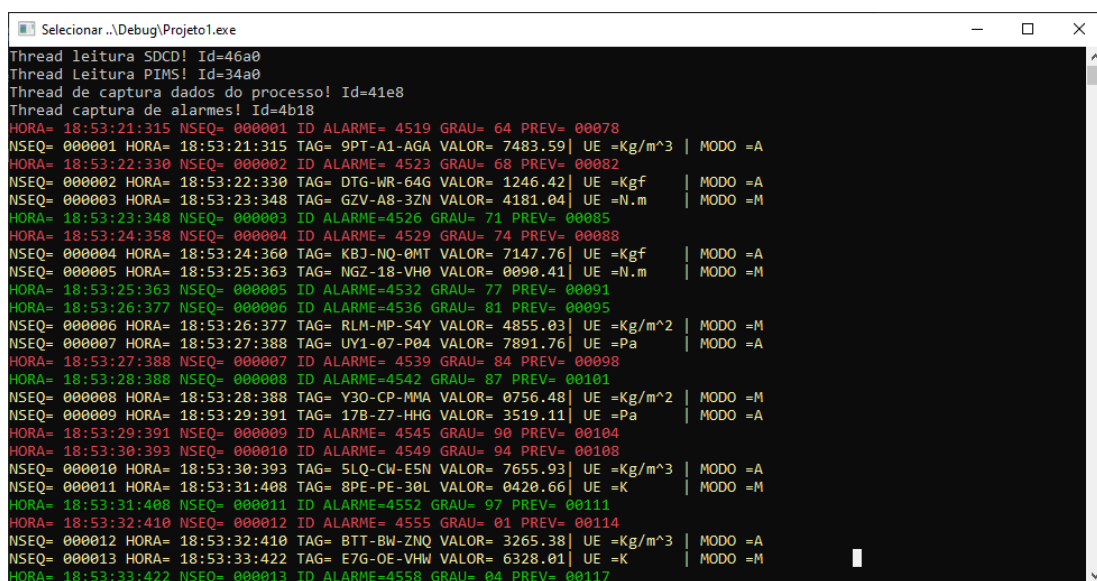
Caso a variável **nTipoEvento == 1** quer dizer que a tecla **ESC** foi digitada, o que faz com que a thread saia do **do while** e termine sua execução.

MODULO 3

Nesse modulo, temos os processos 2 e 3 que são consoles diferentes que esperam, cada uma, a sinalização de um evento específico. 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:



```
Selecionar...\\Debug\\Projeto1.exe
Thread leitura SDCCD! Id=46a0
Thread leitura PIMS! Id=34a0
Thread de captura dados do processo! Id=41e8
Thread captura de alarmes! Id=4b18
HORA= 18:53:21:315 NSEQ= 000001 ID ALARME= 4519 GRAU= 64 PREV= 00078
NSEQ= 000001 HORA= 18:53:21:315 TAG= 9PT-A1-AGA VALOR= 7483.59 | UE =Kg/m^3 | MODO =A
HORA= 18:53:22:330 NSEQ= 000002 ID ALARME= 4523 GRAU= 68 PREV= 00082
NSEQ= 000002 HORA= 18:53:22:330 TAG= DTG-WR-64G VALOR= 1246.42 | UE =Kgf | MODO =A
NSEQ= 000003 HORA= 18:53:23:348 TAG= GZV-A8-3ZN VALOR= 4181.04 | UE =N.m | MODO =M
HORA= 18:53:23:348 NSEQ= 000003 ID ALARME=4526 GRAU= 71 PREV= 00085
HORA= 18:53:24:358 NSEQ= 000004 ID ALARME= 4529 GRAU= 74 PREV= 00088
NSEQ= 000004 HORA= 18:53:24:360 TAG= KBJ-NQ-0MT VALOR= 7147.76 | UE =Kgf | MODO =A
NSEQ= 000005 HORA= 18:53:25:363 TAG= NGZ-18-VH0 VALOR= 0090.41 | UE =N.m | MODO =M
HORA= 18:53:25:363 NSEQ= 000005 ID ALARME=4532 GRAU= 77 PREV= 00091
HORA= 18:53:26:377 NSEQ= 000006 ID ALARME=4536 GRAU= 81 PREV= 00095
NSEQ= 000006 HORA= 18:53:26:377 TAG= RLM-MP-S4Y VALOR= 4855.03 | UE =Kg/m^2 | MODO =M
NSEQ= 000007 HORA= 18:53:27:388 TAG= UY1-07-P04 VALOR= 7891.76 | UE =Pa | MODO =A
HORA= 18:53:27:388 NSEQ= 000007 ID ALARME= 4539 GRAU= 84 PREV= 00098
HORA= 18:53:28:388 NSEQ= 000008 ID ALARME=4542 GRAU= 87 PREV= 00101
NSEQ= 000008 HORA= 18:53:28:388 TAG= Y30-CP-MMA VALOR= 0756.48 | UE =Kg/m^2 | MODO =M
NSEQ= 000009 HORA= 18:53:29:391 TAG= 17B-Z7-HHG VALOR= 3519.11 | UE =Pa | MODO =A
HORA= 18:53:29:391 NSEQ= 000009 ID ALARME= 4545 GRAU= 90 PREV= 00104
HORA= 18:53:30:393 NSEQ= 000010 ID ALARME= 4549 GRAU= 94 PREV= 00108
NSEQ= 000010 HORA= 18:53:30:393 TAG= 5LQ-CW-E5N VALOR= 7655.93 | UE =Kg/m^3 | MODO =A
NSEQ= 000011 HORA= 18:53:31:408 TAG= 8PE-PF-30L VALOR= 0420.66 | UE =K | MODO =M
HORA= 18:53:31:408 NSEQ= 000011 ID ALARME=4552 GRAU= 97 PREV= 00111
HORA= 18:53:32:410 NSEQ= 000012 ID ALARME= 4555 GRAU= 01 PREV= 00114
NSEQ= 000012 HORA= 18:53:32:410 TAG= BTT-BW-ZNQ VALOR= 3265.38 | UE =Kg/m^3 | MODO =A
NSEQ= 000013 HORA= 18:53:33:422 TAG= E7G-OE-VHW VALOR= 6328.01 | UE =K | MODO =M
HORA= 18:53:33:422 NSEQ= 000013 ID ALARME=4558 GRAU= 04 PREV= 00117
```

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 SDCCS 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 SDCCD e PIMS continuam produzindo as mensagens.


```
.\Debug\Projeto1.exe
Thread leitura SDCD! Id=4ad0
Thread leitura PIMS! Id=193c
Thread de captura dados do processo! Id=2fa0
Thread captura de alarmes! Id=40e8
HORA= 19:0:21:808 NSEQ= 000001 ID ALARME= 5891 GRAU= 50 PREV= 01450
NSEQ= 000001 HORA= 19:0:21:808 TAG= MDR-PO-428 VALOR= 9980.39 | UE =KgF | MODO =A
NSEQ= 000002 HORA= 19:0:22:810 TAG= PQE-A6-1P6 VALOR= 2946.01 | UE =N.m | MODO =M
Evento aEvent sinalizado: Bloqueando/desbloqueando a thread tarefa de captura de alarmes
Evento dEvent sinalizado: Bloqueando/desbloqueando a thread de captura de dados do processo
thread de captura de dados do processo bloqueada
thread de captura de alarmes bloqueada
```

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

```
.\Debug\Projeto1.exe
Thread leitura SDCD! Id=48
Thread leitura PIMS! Id=4684
Thread de captura dados do processo! Id=3768
Thread captura de alarmes! Id=4b98
NSEQ= 000001 HORA= 19:3:12:762 TAG= 1PW-NJ-4VA VALOR= 7879.40 | UE =KgF | MODO =A
HORA= 19:3:12:782 NSEQ= 000001 ID ALARME= 6449 GRAU= 14 PREV= 02008
NSEQ= 000002 HORA= 19:3:13:769 TAG= 53K-81-SJG VALOR= 1932.12 | UE =Pa | MODO =A
HORA= 19:3:13:785 NSEQ= 000002 ID ALARME= 6453 GRAU= 18 PREV= 02012
NSEQ= 000003 HORA= 19:3:14:772 TAG= 870-MR-ODN VALOR= 4788.07 | UE =Kg/m^2 | MODO =M
Evento sEvent sinalizado: Bloqueando/desbloqueando a thread leitura do SDCD
Evento pEvent sinalizado: Bloqueando/desbloqueando a thread leitura PIMS
Thread de leitura do SDCD bloqueada
HORA= 19:3:14:789 NSEQ= 000003 ID ALARME=6456 GRAU= 21 PREV= 02015
Thread de leitura do PIMS bloqueada
NSEQ= 000004 HORA= 19:3:15:772 TAG= BBN-08-L1T VALOR= 7543.69 | UE =Pa | MODO =A
HORA= 19:3:15:804 NSEQ= 000004 ID ALARME= 6459 GRAU= 24 PREV= 02018
NSEQ= 000005 HORA= 19:3:16:780 TAG= EPA-LQ-HV0 VALOR= 0606.31 | UE =Kg/m^2 | MODO =M
HORA= 19:3:16:814 NSEQ= 000005 ID ALARME=6462 GRAU= 27 PREV= 02021
```

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:

```
.\Debug\Projeto1.exe
Thread leitura SDCD! Id=3f3c
Thread leitura PIMS! Id=404c
Thread de captura dados do processo! Id=1fd0
Thread captura de alarmes! Id=4390
NSEQ= 000001 HORA= 19:9:4:33 TAG= UMT-YM-026 VALOR= 7902.70| UE =Pa | MODO =A
HORA= 19:9:4:48 NSEQ= 000001 ID ALARME= 7599 GRAU= 75 PREV= 03158
NSEQ= 000002 HORA= 19:9:5:37 TAG= YRG-BC-LWC VALOR= 0867.63| UE =Kg/m^2 | MODO =M
HORA= 19:9:5:52 NSEQ= 000002 ID ALARME=7602 GRAU= 78 PREV= 03161
NSEQ= 000003 HORA= 19:9:6:39 TAG= 14V-WU-HKJ VALOR= 3800.35| UE =Pa | MODO =A
Evento sEvent sinalizado: Bloqueando/desbloqueando a thread leitura do SDCD
Thread de leitura do SDCD bloqueada
HORA= 19:9:6:54 NSEQ= 000003 ID ALARME= 7605 GRAU= 81 PREV= 03164
NSEQ= 000004 HORA= 19:9:7:53 TAG= 58J-AB-DFP VALOR= 7665.98| UE =Kg/m^3 | MODO =A
HORA= 19:9:7:69 NSEQ= 000004 ID ALARME= 7609 GRAU= 85 PREV= 03168
NSEQ= 000005 HORA= 19:9:8:58 TAG= 8DY-NT-A2V VALOR= 0501.60| UE =K | MODO =M
HORA= 19:9:8:73 NSEQ= 000005 ID ALARME=7612 GRAU= 88 PREV= 03171
HORA= 19:9:9:88 NSEQ= 000006 ID ALARME= 7615 GRAU= 91 PREV= 03174
HORA= 19:9:10:90 NSEQ= 000007 ID ALARME=7618 GRAU= 94 PREV= 03177
HORA= 19:9:11:104 NSEQ= 000008 ID ALARME=7622 GRAU= 98 PREV= 03181
HORA= 19:9:12:111 NSEQ= 000009 ID ALARME= 7625 GRAU= 02 PREV= 03184
HORA= 19:9:13:126 NSEQ= 000010 ID ALARME=7628 GRAU= 05 PREV= 03187
HORA= 19:9:14:140 NSEQ= 000011 ID ALARME= 7631 GRAU= 08 PREV= 03190
HORA= 19:9:15:140 NSEQ= 000012 ID ALARME= 7635 GRAU= 12 PREV= 03194
HORA= 19:9:16:145 NSEQ= 000013 ID ALARME=7638 GRAU= 15 PREV= 03197
HORA= 19:9:17:148 NSEQ= 000014 ID ALARME= 7641 GRAU= 18 PREV= 03200
```

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 imprimir na tela dos processos 2 e 3

```
.\Debug\Projeto1.exe
Thread leitura SDCD! Id=30d0
Thread leitura PIMS! Id=19dc
Thread de captura dados do processo! Id=4268
Thread captura de alarmes! Id=16e8
HORA= 19:14:19:380 NSEQ= 000001 ID ALARME= 8627 GRAU= 14 PREV= 04186
NSEQ= 000001 HORA= 19:14:19:380 TAG= 30L-QK-RH9 VALOR= 5387.32| UE =KgF | MODO =A
NSEQ= 000002 HORA= 19:14:20:393 TAG= 728-C2-N4F VALOR= 9443.05| UE =Pa | MODO =A
Evento sEvent sinalizado: Bloqueando/desbloqueando a thread leitura do SDCD
Thread de leitura do SDCD bloqueada
HORA= 19:14:20:395 NSEQ= 000002 ID ALARME= 8631 GRAU= 18 PREV= 04190
HORA= 19:14:21:406 NSEQ= 000003 ID ALARME=8634 GRAU= 21 PREV= 04193
NSEQ= 000003 HORA= 19:14:21:408 TAG= A6N-PJ-K0M VALOR= 2288.67| UE =Kg/m^2 | MODO =M
HORA= 19:14:22:408 NSEQ= 000004 ID ALARME= 8637 GRAU= 24 PREV= 04196
Thread de leitura do SDCD desbloqueada
Evento sEvent sinalizado: Bloqueando/desbloqueando a thread leitura do SDCD
Evento pEvent sinalizado: Bloqueando/desbloqueando a thread leitura PIMS
Evento aEvent sinalizado: Bloqueando/desbloqueando a thread tarefa de captura de alarmes
Thread de leitura do PIMS bloqueada
thread de captura de alarmes bloqueada
Evento dEvent sinalizado: Bloqueando/desbloqueando a thread de captura de dados do processo
thread de captura de dados do processo bloqueada
Evento oEvent sinalizado: Bloqueando/desbloqueando a thread de exibicao de dados do processo
Evento cEvent sinalizado: Indicando a thread de exibicao de dados exibipao de dados do alarme
```

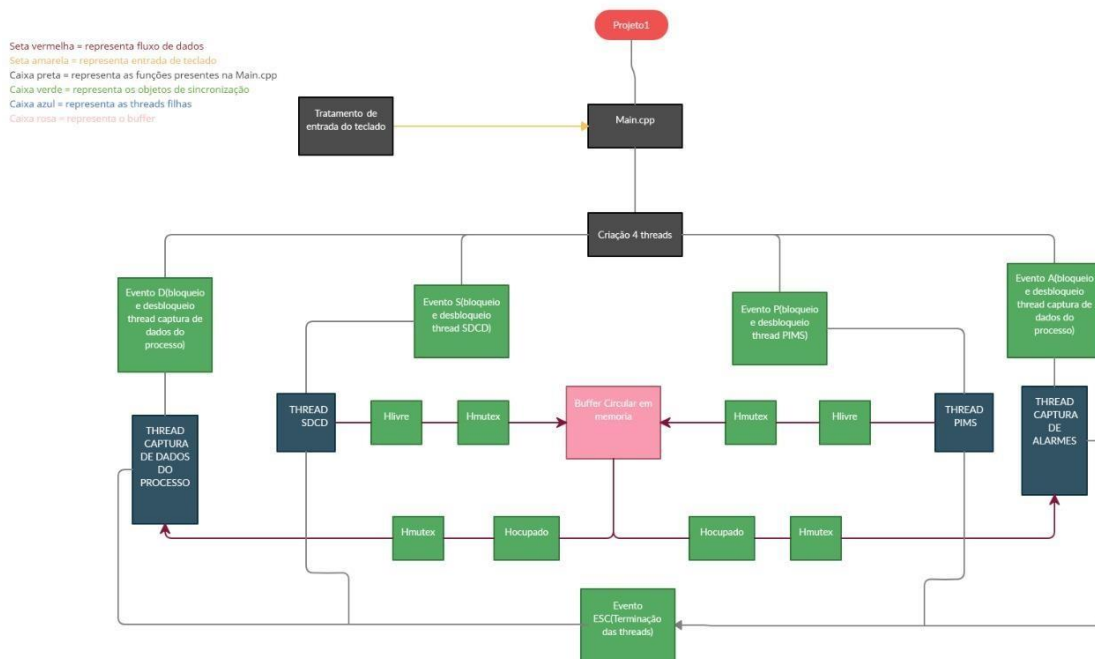
The image shows two separate windows of a Windows command prompt, each running a different program. The top window, titled ".\Debug\Projeto2.exe", shows the following output: "Thread tarefa exibicao de dados do processo! Id=15bc", "Thread de exibicao dos dados do processo (0) esperando o evento 'o'...", "Evento 'o' sinalizado", and "Thread de exibicao dos dados do processo (0) esperando o evento 'o'...". The bottom window, titled ".\Debug\Projeto3.exe", shows the following output: "Thread tarefa exibicao de alarmes! Id=2a58", "Thread de exibicao de alarmes (0) esperando o evento 'c'...", "Evento 'c' sinalizado", and "Thread de exibicao de alarmes (0) esperando o evento 'c'...". Both windows have a black background and white text, with a standard Windows window title bar at the top.

```
.\Debug\Projeto2.exe
Thread tarefa exibicao de dados do processo! Id=15bc
Thread de exibicao dos dados do processo (0) esperando o evento 'o'...
Evento 'o' sinalizado
Thread de exibicao dos dados do processo (0) esperando o evento 'o'...

.\Debug\Projeto3.exe
Thread tarefa exibicao de alarmes! Id=2a58
Thread de exibicao de alarmes (0) esperando o evento 'c'...
Evento 'c' sinalizado
Thread de exibicao de alarmes (0) esperando o evento 'c'...
```

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, teríamos que temporizar a tarefa PIMS e a tarefa SDCC, implementar a comunicação entre processos (IPC) e fazer a implementação de um arquivo circular em memória. Iremos dividir a documentação em três módulos, para facilitar o entendimento da lógica da implementação.

Modulo 1 : Temporização dos processos

Começaremos esse módulo com a temporização da tarefa SDCC. Essa tarefa tem uma temporização de 500 ms. Para implementar essa temporização usamos o **WaitableTimer** dentro de um laço 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 variável **Tempo_de_espera.QuadPart** como negativa, pois o tempo é relativo e atribuímos ela a um valor de 500ms (500 pacotes de 10000 nanosegundos). Logo após usamos a função **SetWaitableTimer** de disparo único, mas como ela está num laço de **do while** ela irá repetir e temporizar de forma periódica, 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 segue sua execução, caso o **htimer** não 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 temporizadores, um para cada tipo de alarme, crítico e não crítico. Iremos discutir a seguir a lógica 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 críticos e críticos, pois se o evento de alarme crítico sinalizar a gente vai gerar uma mensagem de alarme crítico e enviamos ela via **mailslot** para o outro processo. Se a mensagem de alarme não crítico for escolhida vai gerar uma mensagem e depositar no buffer circular. Uma nota, usamos a função **gera_numero_aleatorio**, pois temos uma faixa de valores de tempo que podem ser executados os alarmes. Quando o código continua executando, caso tenha escolhido um alarme crítico, dentro do **do while** tem um outro **SetWaitbletimer** para setar novamente o cronômetro e ter outro valor dentro da faixa decidida na especificação do trabalho, o mesmo acontece com os alarmes não críticos.

MODULO 2

Nesse módulo, 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 utilizamos o evento **mEvento** para indicar a criação do mailslot pelo processo servidor, no caso a tarefa de exibição de alarme. Assim as threads de captura de alarmes e de geração de alarmes funcionam como clientes, que aguardam a sinalização do evento para iniciar a sua execução, 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,
    MAILSLT_WAIT_FOREVER,
    NULL);

SetEvent(mEvent);

WaitForSingleObject(mEvent, INFINITE);

```

Quando o alarme é não crítico, ocorre o depósito da mensagem no buffer circular em memória para que depois ele seja enviado para o outro processo via mailslot.

MODULO 3

Nesse módulo iremos implementar um arquivo circular em memória. Para essa etapa do trabalho

criamos um arquivo circular em memória utilizando um semáforo 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 exibição de dados de processos lê o arquivo, assim exibindo as mensagens em seu console. Para voltar para o início do arquivo utilizamos a função **setfilepointer**.

```
status = WriteFile(hfile, &dado, sizeof(dado), &dwBytesWritten, NULL);
status = ReleaseSemaphore(hocupado2, 1, NULL);
if (status == NULL) {
    printf("Falha na função 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);
```