Trabalho Prático de Automação em Tempo Real

Parte 2

Gustavo Soares Cesário

Introdução

Esse trabalho consiste na elaboração de um sistema automatizado para inspeção de defeitos em tiras metálicas produzidas no setor de laminação a quente em uma indústria siderúrgica.

Durante a primeira parte do trabalho, foram desenvolvidas apenas a funcionalidades básicas da aplicação, que consiste na arquitetura básica da aplicação, bloqueio e desbloqueio de threads secundárias e implementação preliminar da lista circular em memória. Já durante a segunda parte do trabalho foram desenvolvidos temporizadores para o depósito de mensagens na lista circular em memória, foi feita a criação de um arquivo circular em disco para comunicação entre a tarefa de inserção de dados de processo, e por fim, foi feito o desenvolvimento dos mecanismos de IPC entre as tarefas conforme a especificação presente no enunciado do trabalho.

Código

Objetos de sincronização

Nesse trabalho foram usados os seguintes objetos de sincronização:

- **Handle_Thread_****

Esses handles representam as quatro threads executadas pelo processo principal. Cada thread será detalhada no próximo tópico.

- **Evento_Nao_Finalizar_****

Esses handles representam os eventos responsáveis por finalizar cada umas das threads ou processos do programa. Seu estado sinalizado significa que o

programa deve continuar, já o estado não sinalizado sinaliza que a thread deve encerrar seu funcionamento.

- **Evento_Desbloquear_****

Esses handles representam os eventos responsáveis por bloquear ou desbloquear cada uma das threads presentes no programa. Um estado sinalizado significa que a thread deve prosseguir sua execução, já um estado não sinalizado significa que ela deve ser pausada.

- **Semaforo_Acesso_Lista_Circular_Livres**

Semáforo contador responsável por armazenar a informação de quantas posições estão disponíveis na lista circular em memória.

- **Semaforo_Acesso_Lista_Circular_Ocupados**

Semáforo contador responsável por armazenar a informação de quantas posições estão ocupadas na lista circular em memória.

- **Evento_Lista_Circular_Nao_Cheia**

Evento responsável por alertar a thread de inspeção de dados que a lista circular não está mais cheia, e que ela pode prosseguir adicionando novas mensagens.

- **Evento_Lista_Circular_Contem_Dado_Processo**

Evento responsável por avisar a thread de captura de dados de processos que existe uma nova mensagem de dado de processo na lista circular.

- **Evento_Lista_Circular_Contem_Defeito**

Evento responsável por avisar a thread de captura de defeitos que existe uma nova mensagem de defeito na lista circular.

- **Mutex_Acesso_Lista_Circular**

Mutex responsável por realizar a exclusão mútua de acessos na lista circular.

- **Mutex_Acesso_Console**

Mutex responsável por realizar a exclusão mútua do acessos ao console, de modo que o print de uma tarefa não interfira no print de outra, assim como as cores.

- **Semaforo_Arquivo_Dados_Processo_Livre**

Semáforo responsável por limitar a capacidade de mensagens do arquivo circular.

- **Evento_Arquivo_Nao_Cheio**

Evento responsável por alertar a thread de captura de dados de processo que o arquivo circular não está mais cheio, e que ela pode prosseguir adicionando novas mensagens.

- **Evento_Timer_**_Executado**

Evento responsável por alertar a thread de inspeção de defeitos que a rotina de execução do timer específico foi executada.

- **Evento_Arquivo_Nao_Cheio**

Evento responsável por alertar a thread de captura de dados de processo que o arquivo circular não está mais cheio, e que ela pode prosseguir adicionando novas mensagens.

- **Mutex_Acesso_Arquivo**

Mutex responsável por realizar a exclusão mútua do acesso ao arquivo circular em disco.

Objetos de IPC

- **Pipe_Dados_De_Processo**

Esse pipe é responsável por sinalizar a tarefa de exibição de dados de processo que há uma nova mensagem disponível para ser lida no arquivo circular.

- **Pipe_Limpar_Tela**

Esse pipe é responsável por sinalizar a tarefa de exibição de dados de processo que ela deve limpar a sua tela.

- **Pipe_Defeitos_Das_Tiras**

Esse pipe é responsável por enviar mensagens à tarefa de exibição de defeitos para que elas possam ser exibidas em tela.

Objetos de IPC

- **Temporizador_Dados_Processo**

Esse TimerQueueTimer é responsável por executar a função que gera uma mensagem de dados de processo e a coloca na lista circular em memória.

- **Temporizador_Dados_Processo**

Esse TimerQueueTimer é responsável por executar a função que gera uma mensagem de defeito de tira e a coloca na lista circular em memória.

Estruturas de dados

- **DefeitoTira**

Essa estrutura de dados representa um defeito de superfície de tira e foi construída a partir da documentação fornecida com os tipos de variáveis correspondentes.

```
typedef struct DefeitoTira {
      int numero;
      int tipo;
      int cadeira;
      int gravidade;
      int classe;
      std::string id_foto;
      Tempo tempo;
} DefeitoTira;
```

- **DadosProcesso**

Similar à estrutura anterior, essa estrutura de dados representa dados de processo e foi construída a partir da documentação fornecida com os tipos de variáveis correspondentes.

```
typedef struct DadosProcesso {
      int numero;
      int tipo;
      int cadeira;
```

```
      std::string id;
      float temp;
      float vel;
      Tempo tempo;
} DadosProcesso;
```

- **Tempo**

Essa estrutura foi construída apenas para facilitar a manipulação de tempo em diversas parte do código.

```
typedef struct Tempo {
      int hora;
      int minuto;
      int segundo;
      int milissegundo;
} Tempo;
```

Threads

- Thread_Leitura_Teclado

Essa thread é responsável por realizar a leitura das teclas do teclado e enviar mensagens para qualquer outra thread para indicar o seu bloqueio, desbloqueio, ou finalização. Após iniciar os handles e as variáveis que controlam os estados das threads, ela entra em um loop infinito que só é quebrado ao apertar a tecla ESC. Enquanto ela está no loop ela lê constantemente a entrada do teclado pela função _getch(). Após digitar a tecla, o comando correspondente é executado, seja ele alternar uma thread, limpar o console de outro processo, etc.

```
case 'i':
	MostrarMensagem("Alternando tarefa de inspecao de defeitos...", CIANO);
	AlternarEvento(Evento_Desbloquear_Inspecao_Defeitos, &Estado_Inspecao_Defeitos);
	break;
```

No caso do comando para limpar a tela da janela de exibição de dados de processo, ao apertar a tecla “i”, é enviada uma mensagem ao pipe que indica que a tela deverá ser limpa.

```
char Mensagem_Novos_Dados = '1';
DWORD Bytes_Escritos;
case 'c':
	MostrarMensagem("Limpando janela...", CINZA);
	WriteFile(
		Pipe_Limpar_Tela,
		&Mensagem_Novos_Dados,
		sizeof(char),
		&Bytes_Escritos,
		NULL
	);
	break;
```

A função AlternarEvento simplesmente recebe o evento a ser bloqueado ou desbloqueado e a sua variável booleana correspondente. Após executar SetEvent ou ResetEvent (de acordo com o estado atual) a variável booleana é atualizada.

```
void AlternarEvento(HANDLE evento, bool* Estado_Atual) {
	if (*Estado_Atual == DESBLOQUEADA) {
		ResetEvent(evento);
		*Estado_Atual = BLOQUEADA;
	}
	else {
		SetEvent(evento);
		*Estado_Atual = DESBLOQUEADA;
```

```
        }
}
```

Ao pressionar a tecla ESC, o laço é quebrado e logo em seguida a thread sinaliza os eventos de finalização das threads, assim como quaisquer objetos de sincronização que possam estar bloqueando sua execução. Dessa forma, todas as threads saem do seu laço infinito e são encerradas.

- Thread_Sistema_Inspecao_Defeitos

Essa thread é responsável pelo sistema de inspeção de defeitos. Ao iniciar sua execução, ela abre os handles necessários para sua execução e cria os primeiros timers necessários para gerar mensagens que serão colocadas na lista circular em memória.

Em relação a sincronização, no começo da thread o comando WaitForMultipleObjects é executado, passando como parâmetro uma lista contendo os handles para os eventos que sinalizam o término de execução da rotina de um timer. Em seguida, a thread executa a função WaitForSingleObject() com o semáforo *Semaforo_Acesso_Lista_Circular_Livres* com um timeout de valor 0. Caso o retorno dessa função seja WAIT_TIMEOUT, isso significa que o semáforo não estava sinalizado, indicando que a lista circular em memória está cheia. Assim, a thread executa a função WaitForSingleObject() no evento *Evento_Lista_Circular_Nao_Cheia* de modo a esperar uma sinalização de que a lista circular em memória não está mais cheia e que ela pode continuar depositando mensagens nele. Quando o evento é sinalizado, a execução do laço é retornada ao começo por meio do comando “continue”.

```
int Status_Wait_Lista_Livre =
WaitForSingleObject(Semaforo_Acesso_Lista_Circular_Livres, 0);
if (Status_Wait_Lista_Livre == WAIT_TIMEOUT) {
      printf("Lista circular cheia!!!\n");
      WaitForSingleObject(Evento_Lista_Circular_Nao_Cheia, INFINITE);
      continue;
}
```

Após estar apta a adicionar mensagens na lista circular em memória, a thread checa qual evento relacionado ao timer foi executado, deleta o timer correspondente e o recria, para que novas mensagens continuem a serem adicionadas.

```
if (Status_Wait == WAIT_OBJECT_0 + 0) {
       DeleteTimerQueueTimer(
              Fila_Temporizadores,
              Temporizador_Dados_Processo,
              INVALID_HANDLE_VALUE
       );
       Criar_Timer_Dados_Processo(&Temporizador_Dados_Processo, &Fila_Temporizadores);
}
```

As funções que criam os timers possuem um comportamento bem básico. Elas apenas recebem um apontador para o handle do timer a ser criado e um apontador para o handle da fila de temporizadores que irá receber o timer. A diferença da função que cria o timer de dados de processos para a que cria o timer de defeito de tira é apenas o tempo que é passado como parâmetro. Enquanto o timer de dados de processo possui um tempo de 500ms, o timer de dados possui um tempo aleatório entre 100ms e 2s.

```
void Criar_Timer_Defeitos_Tiras(HANDLE* Temporizador_Defeito_Tira,
HANDLE* Fila_Temporizadores) {
	int tempo = 100 + (rand() % 1900);
	CreateTimerQueueTimer(
		Temporizador_Defeito_Tira,
		*Fila_Temporizadores,
		(WAITORTIMERCALLBACK)Rotina_Mensagem_Defeitos_Tiras,
		NULL,
		tempo,
		0,
		WT_EXECUTEDEFAULT
	);
}
```

A rotina que é executada após o término de espera do timer também possui um comportamento bem simples. Primeiramente ela abre os handles necessários para realizar a sincronização. Em seguida ela gera uma nova mensagem e a transforma em string. Por fim, ela adiciona a mensagem na lista circular em memória e sinaliza os handles necessários.

```
VOID CALLBACK Rotina_Mensagem_Defeitos_Tiras(PVOID param, BOOLEAN
TimerOrWaitFired) {

	HANDLE Evento_Timer_Defeitos_Tiras_Executado = OpenEvent(SYNCHRONIZE |
EVENT_MODIFY_STATE, false, "Evento_Timer_Defeitos_Tiras_Executado");
	HANDLE Semaforo_Acesso_Lista_Circular_Ocupados =
OpenSemaphore(SYNCHRONIZE | SEMAPHORE_MODIFY_STATE, false,
"Semaforo_Acesso_Lista_Circular_Ocupados");
	HANDLE Evento_Lista_Circular_Contem_Defeito = OpenEvent(SYNCHRONIZE |
EVENT_MODIFY_STATE, false, "Evento_Lista_Circular_Contem_Defeito");

	std::string mensagem = SerializarDefeitoTira(GerarDefeitoTira());

	Adicionar_Mensagem_Na_Lista(mensagem);
	SetEvent(Evento_Lista_Circular_Contem_Defeito);
	ReleaseSemaphore(Semaforo_Acesso_Lista_Circular_Ocupados, 1, NULL);

	SetEvent(Evento_Timer_Defeitos_Tiras_Executado);
}
```

- Thread_Captura_Defeitos_Tiras

Essa thread é responsável pela captura de mensagens de defeitos de tiras da lista circular em memória. Ela não possui atraso de execução e portanto retira as mensagens da lista de forma quase instantânea.

Após abrir todos os handles necessários, a thread entra em um laço e executa a função WaitForSingleObject() com o semáforo *Semaforo_Acesso_Lista_Circular_Ocupados* com um timeout de valor 0. Caso o retorno dessa função seja WAIT_TIMEOUT, isso significa que o semáforo não estava sinalizado, indicando que a lista circular em memória está vazia. Assim, a thread executa a função WaitForSingleObject() no evento *Evento_Lista_Circular_Contem_Defeito* de modo a esperar uma sinalização de que a lista circular possui uma nova mensagem de defeito de tira e que ela pode continuar retirando mensagens dela. Quando o evento é sinalizado, a execução do laço é retornada ao começo por meio do comando “continue”. Esse comportamento é igual ao comportamento da thread *Thread_Sistema_Inspecao_Defeitos.*

```
int Status_Wait_Lista_Ocupada =
WaitForSingleObject(Semaforo_Acesso_Lista_Circular_Ocupados, 0);
if (Status_Wait_Lista_Ocupada == WAIT_TIMEOUT) {
	WaitForSingleObject(Evento_Lista_Circular_Contem_Defeito, INFINITE);
	continue;
}
```

Após essa validação de execução, a thread conquista o mutex *Mutex_Acesso_Lista_Circular* para poder acessar a lista com segurança, acessa uma mensagem da lista de acordo com o seu próprio ponteiro de posição de acesso da lista e a desserializa usando a função útil DesserializarDefeitoTira() que retorna um struct que representa um defeito de superfície de tira. Em seguida a thread checa o tipo de mensagem que foi lida da lista, caso não seja uma mensagem do tipo 22, a thread incrementa o semáforo contador *Semaforo_Acesso_Lista_Circular_Ocupados* de modo a reverter o acesso que foi feito nele anteriormente, já que não foi feita a retirada da mensagem da lista. Mas caso a mensagem seja do tipo 11, a thread apaga a mensagem da lista circular em memória, envia a mensagem para a tarefa de exibição de defeitos por meio do pipe *Pipe_Defeitos_Das_Tiras*, sinaliza o evento *Evento_Lista_Circular_Nao_Cheia* e incrementa o semáforo contador *Semaforo_Acesso_Lista_Circular_Livres*, indicando que existem posições livres na lista. Por fim, a thread incrementa o seu ponteiro de posição de acesso da lista e libera o mutex *Mutex_Acesso_Lista_Circular*.

```
WaitForSingleObject(Mutex_Acesso_Lista_Circular, INFINITE);
std::string Proxima_Mensagem_Da_Fila = Lista_Circular_Memoria[Ponteiro_Leitura_Defeitos
% TAMANHO_LISTA];
DefeitoTira defeito = DesserializarDefeitoTira(Proxima_Mensagem_Da_Fila);

if (defeito.tipo == 11) {
	Lista_Circular_Memoria[Ponteiro_Leitura_Defeitos % TAMANHO_LISTA] = "";
```

```
        DWORD Bytes_Escritos;
        WriteFile(
             Pipe_Defeitos_Das_Tiras,
             Proxima_Mensagem_Da_Fila.c_str(),
             sizeof(char) * Proxima_Mensagem_Da_Fila.length() + 1,
             &Bytes_Escritos,
             NULL
        );

        ReleaseSemaphore(Semaforo_Acesso_Lista_Circular_Livres, 1, NULL);
        SetEvent(Evento_Lista_Circular_Nao_Cheia);
}
else {
        //A tarefa nao corresponde ao tipo procurado, portanto, nao vamos retira-la.
        ReleaseSemaphore(Semaforo_Acesso_Lista_Circular_Ocupados, 1, NULL);
}

Ponteiro_Leitura_Defeitos++;
ReleaseMutex(Mutex_Acesso_Lista_Circular);
```

- **Thread_Captura_Dados_Processos**

O comportamento dessa thread é muito similar ao comportamento da thread anterior, mas com algumas mudanças em sua estrutura. Após a chamada à função *WaitForSingleObject*, que determina se existem mensagens na lista circular em memória (lógica descrita na Thread_Captura_Defeito_De_Tiras), a thread realiza a mesma operação mas para o semáforo *Semaforo_Arquivo_Dados_Processo_Livre*, dessa forma, caso o arquivo em disco não possua mais espaço livre para armazenar mensagens devido ao seu limite de 100 mensagens proposto pelo enunciado, a thread se bloqueia até que o semáforo seja sinalizado pela tarefa de exibição de dados de processo.

```
int Status_Semaforo_Arquivo =
WaitForSingleObject(Semaforo_Arquivo_Dados_Processo_Livre, 0);
if (Status_Semaforo_Arquivo == WAIT_TIMEOUT) {
      MostrarMensagem("Arquivo cheio!!!", VERMELHO);
      WaitForSingleObject(Evento_Arquivo_Nao_Cheio, INFINITE);
      MostrarMensagem("Arquivo liberado!!!", CIANO);
      continue;
}
```

Em seguida, após retirar a mensagem da lista circular em memória, a thread adiciona a mensagem na lista circular em arquivo, realizando todo o processo de exclusão mútua e posicionamento de ponteiros necessário. Assim, a mensagem pode ser lida no arquivo pela tarefa de exibição de dados de processo.

```
WaitForSingleObject(Mutex_Acesso_Arquivo, INFINITE);
LockFile(
      Arquivo_Dados_De_Processo,
      Ponteiro_Escrita_Arquivos * sizeof(char) * TAMANHO_ARQUIVO,
      NULL,
      sizeof(char) * TAMANHO_ARQUIVO,
      NULL
);
SetFilePointer(
      Arquivo_Dados_De_Processo,
      Ponteiro_Escrita_Arquivos * sizeof(char) * TAMANHO_ARQUIVO,
      NULL,
      FILE_BEGIN
);
WriteFile(
      Arquivo_Dados_De_Processo,
      Proxima_Mensagem_Da_Fila.c_str(),
      sizeof(char) * TAMANHO_ARQUIVO,
      &Bytes_Escritos,
      NULL
);
UnlockFile(
      Arquivo_Dados_De_Processo,
      Ponteiro_Escrita_Arquivos * sizeof(char) * TAMANHO_ARQUIVO,
      NULL,
      sizeof(char) * TAMANHO_ARQUIVO,
      NULL
);
ReleaseMutex(Mutex_Acesso_Arquivo);
```

Em seguida a thread envia uma mensagem básico para a tarefa de exibição de dados por meio do pipe *Pipe_Dados_De_Processo* de modo a sinalizar que existe uma nova mensagem no arquivo pronta para ser lida.

```
char Mensagem_Novos_Dados = '1';
WriteFile(
      Pipe_Dados_De_Processo,
      &Mensagem_Novos_Dados,
      sizeof(char),
      &Bytes_Escritos,
      NULL
);
```

Por fim, a thread libera o semáforo e o evento de modo a sinalizar que existe uma nova posição livre na lista circular em memória e incrementa o ponteiro de escrita no arquivo.

```
ReleaseSemaphore(Semaforo_Acesso_Lista_Circular_Livres, 1, NULL);
SetEvent(Evento_Lista_Circular_Nao_Cheia);
Ponteiro_Escrita_Arquivos = (Ponteiro_Escrita_Arquivos + 1) % 100;
```

Processos

- TrabalhoPratico1

Esse é o processo principal da aplicação, responsável por inicializar os handles de sincronização, as threads e os processos. Após essa inicialização, o processo executa a função WaitForMultipleObjects(), de modo a esperar a finalização das quatro threads descritas acima.

```
status = WaitForMultipleObjects(4, Threads, true, INFINITE);

if (status != WAIT_OBJECT_0) {
      printf("Erro em WaitForMultipleObjects! Codigo = %d\n",
GetLastError());
      return 0;
}

printf("Finalizando...");
```

- ExibicaoDeDefeitos

Esse processo é responsável por mostrar os defeitos lidos pela thread de captura de defeitos na tela. Esse processo inicialmente abre os eventos de sincronização *Evento_Nao_Finalizar_Exibicao_De_Defeitos* e *Evento_Desbloquear_Exibicao_De_Defeitos* e entra em um laço que só é quebrado no momento que o evento *Evento_Nao_Finalizar_Exibicao_De_Defeitos* é levado ao estado não sinalizado. No começo de cada iteração a função WaitForSingleObject() é executada de modo a bloquear a thread caso o evento *Evento_Desbloquear_Exibicao_De_Defeitos* esteja no estado não sinalizado.

Durante a execução de cada loop, a tarefa lê as mensagens recebidas pela tarefa de captura de defeitos das tiras no pipe *Pipe_Defeitos_Das_Tiras*, formata a mensagem de acordo com o padrão proposto no enunciado e em seguida a mostra na tela.

```
do {
	WaitForSingleObject(Evento_Desbloquear_Exibicao_De_Defeitos,
INFINITE);

	char Mensagem_Lida[TAMANHO_MENSAGEM];
	DWORD Bytes_Lidos;

	ReadFile(
		Pipe_Defeitos_Das_Tiras,
		Mensagem_Lida,
		37,
		&Bytes_Lidos,
		NULL
	);

	std::string Mensagem_Formatada =
FormatarDefeitoTira(DesserializarDefeitoTira(Mensagem_Lida));

	std::cout << Mensagem_Formatada << std::endl;

	resultadoEvento =
WaitForSingleObject(Evento_Nao_Finalizar_Exibicao_De_Defeitos, 0);
} while (resultadoEvento == WAIT_OBJECT_0);
```

- **ExibicaoDadosDeProcesso**

O funcionamento desse processo similar ao anterior. Ao iniciar, ele inicia uma thread secundária responsável apenas pela leitura da tela. Essa thread abre o pipe *Pipe_Limpar_Tela* e entra em um loop que só é quebrado quando uma variável auxiliar é sinalizada pela thread principal e quando o pipe é fechado pelo processo servidor.. Ao realizar a leitura de qualquer dado vindo pelo pipe, a thread executa o comando nativo do windows para limpar a tela.

```
DWORD WINAPI Thread_Limpeza_Tela(LPVOID thread_arg) {
	WaitNamedPipe("Pipe_Dados_De_Processo", NMPWAIT_USE_DEFAULT_WAIT);
	Pipe_Limpar_Tela = CreateFile(
		"\\\\.\\pipe\\Pipe_Limpar_Tela",
		GENERIC_READ,
		0,
		NULL,
		CREATE_ALWAYS,
		0,
		NULL
```

```
	);

	while (!Finalizar) {
		char Buffer_Mensagem_Pipe;
		ReadFile(Pipe_Limpar_Tela, &Buffer_Mensagem_Pipe,
sizeof(char), NULL, NULL);
		system("cls");
	}

	_endthreadex((DWORD)6);
	return 6;
}
```

Em relação à thread principal, no começo de sua execução entra ela em um laço similar ao laço das outras threads. Em cada execução do laço, a thread aguarda a leitura de uma mensagem presente no pipe *Pipe_Dados_De_Processo* para que ela possa fazer a leitura do arquivo circular. Em seguida ela lê a mensagem do arquivo, fazendo todo os processos de exclusão mútua e posicionamento de ponteiro necessários. Após feita essa leitura, ela formada a mensagem de acordo com os padrões propostos pelo enunciado, mostra na tela e libera o arquivo.

```
WaitForSingleObject(Mutex_Acesso_Arquivo, INFINITE);
LockFile(
	Arquivo_Dados_De_Processo,
	Ponteiro_Leitura_Arquivo * sizeof(char) * TAMANHO_ARQUIVO, 0,
	(Ponteiro_Leitura_Arquivo + 1) * sizeof(char) * TAMANHO_ARQUIVO,
	0
);
SetFilePointer(
	Arquivo_Dados_De_Processo,
	Ponteiro_Leitura_Arquivo * sizeof(char) * TAMANHO_ARQUIVO,
	NULL,
	FILE_BEGIN
);
DWORD Bytes_Lidos;
char Mensagem_Arquivo[TAMANHO_ARQUIVO];
bool res = ReadFile(
	Arquivo_Dados_De_Processo,
	&Mensagem_Arquivo,
	sizeof(char) * TAMANHO_ARQUIVO,
	&Bytes_Lidos,
	NULL
);

std::string Mensagem_Formatada =
```

```
FormatarDadosProcesso(DesserializarDadosProcesso(Mensagem_Arquivo));
std::cout << Mensagem_Formatada << std::endl;

UnlockFile(
      Arquivo_Dados_De_Processo,
      Ponteiro_Leitura_Arquivo * sizeof(char) * TAMANHO_ARQUIVO,
      0,
      (Ponteiro_Leitura_Arquivo + 1) * sizeof(char) * TAMANHO_ARQUIVO,
      0
);
ReleaseMutex(Mutex_Acesso_Arquivo);
```

Por fim, a tarefa incrementa a posição do ponteiro do arquivo circular e sinaliza o semáforo e o evento responsáveis por sinalizar que a lista circular no arquivo não está mais cheia.

```
Ponteiro_Leitura_Arquivo = (Ponteiro_Leitura_Arquivo + 1) % 100;

ReleaseSemaphore(Semaforo_Arquivo_Dados_Processo_Livre, 1, NULL);
SetEvent(Evento_Arquivo_Nao_Cheio);
```

Utilidades

Durante o desenvolvido do trabalho, foram criados arquivos úteis de modo a facilitar o seu desenvolvimento e proporcionar uma melhor organização.

- DadosDeProcesso.cpp

Esse arquivo contém as principais funcionalidades das mensagens de dados do processo de laminação e contém três principais funções.

A função GerarDadosProcesso() retorna uma estrutura *DadosProcesso* gerada aleatoriamente por meio da função rand() nos campos apropriados. Para o campo **número**, a função usa a variável de controle *Numero_Dado_Processo* que é incrementada a cada geração de mensagem. O campo **tipo** possui um valor fixo de 22. O campo **tempo** usa a função GerarTempoAtual() para definir o tempo de geração de mensagem.

```
DadosProcesso GerarDadosProcesso() {
      DadosProcesso dados;

      srand(time(NULL));
      dados.numero = Numero_Dado_Processo;
      dados.tipo = 22;
      dados.cadeira = rand() % 6 + 1;
```

```
        dados.id = GerarIdAleatorio(8);
        dados.temp = 200.0f + (((float)rand() / (float)(RAND_MAX)) * 800);
        dados.vel = 200.0f + (((float)rand() / (float)(RAND_MAX)) * 800);
        dados.tempo = GerarTempoAtual();

        Numero_Dado_Processo++;
        return dados;
}
```

A função SerializarDadosProcesso() converte a estrutura de dados em uma string que possa ser lida por um usuário ou armazenada na lista circular em memória. Essa função primeiro converte os valores numéricos (reais ou inteiros) e os valores de tempo para strings e os armazena em cadeias de caracteres. Por fim, todas essas cadeias de caracteres convertidas são armazenadas em uma única cadeia de caracteres para que a mensagem possa ser retornada por completa. É importante ressaltar que as cadeias de caracteres possuem uma posição extra disponível além do que foi detalhado na descrição do trabalho para armazenar o caractere '\0', que indica o fim da string. A função FormatarDadosProcesso() possui um comportamento idêntico, sendo diferente apenas no formato de saída dos dados.

```
std::string SerializarDadosProcesso(DadosProcesso dados) {
        char NSEQ_Formatado[6];
        char CADEIRA_Formatado[3];
        char ID_Formatado[9];
        char TEMP_Formatado[6];
        char VEL_Formatado[6];
        char TEMPO_Formatado[13];

        sprintf_s(NSEQ_Formatado, 6, "%05d", dados.numero);
        sprintf_s(CADEIRA_Formatado, 3, "%02d", dados.cadeira);
        strcpy_s(ID_Formatado, 9, dados.id.c_str());
        sprintf_s(TEMP_Formatado, 6, "%05.1f", dados.temp);
        sprintf_s(VEL_Formatado, 6, "%05.1f", dados.vel);
        strcpy_s(TEMPO_Formatado, 13, SerializarTempo(dados.tempo).c_str());

        char Mensagem_Serializada[46];
        sprintf_s(Mensagem_Serializada, 46, "%s/%d/%s/%s/%s/%s/%s",
                NSEQ_Formatado,
                dados.tipo,
                CADEIRA_Formatado,
                ID_Formatado,
                TEMP_Formatado,
                VEL_Formatado,
                TEMPO_Formatado
        );

        return Mensagem_Serializada;
}
```

A função DesserializarDadosProcesso() realiza o procedimento inverso em relação à função anterior. Essa função converte uma string em uma estrutura de dados *DadosProcesso* para facilitar o posterior acesso aos seus campos. Primeiramente ela checa o tamanho da string, e caso ele seja diferente de 45, isso significa que essa não é uma string que representa um dado de processo de laminação e portanto, não pode ser convertida em uma estrutura *DadosProcesso*. Caso isso ocorra, o tipo da estrutura é definido como 0, e a estrutura é logo em seguida retornada pela função. Após a checagem de segurança, a string é dividida pelo delimitador "/" e após cada leitura o contador **posicao** é incrementado, de modo a mostrar qual campo da string a função está lendo no momento. De acordo com a posição, a parte de interesse da mensagem é lida e depositada na estrutura de dados. Após a leitura de todos os campos com sucesso, a estrutura convertida é retornada pela função.

```
DadosProcesso DesserializarDadosProcesso(std::string dadosString) {
	DadosProcesso dados;
	if (dadosString.size() != 45) {
		dados.tipo = 0;
		return dados;
	}

	std::string delim = "/";
	auto start = 0U;
	auto end = dadosString.find(delim);
	int posicao = 0;
	while (end != std::string::npos)
	{
		switch (posicao) {
		case 0:
			dados.numero = std::stoi(dadosString.substr(start, end - start));
			break;
		case 1:
			dados.tipo = std::stoi(dadosString.substr(start, end - start));
			break;
		case 2:
			dados.cadeira = std::stoi(dadosString.substr(start, end - start));
			break;
		case 3:
			dados.id = dadosString.substr(start, end - start);
			break;
		case 4:
			dados.temp = std::stof(dadosString.substr(start, end - start));
			break;
		case 5:
			dados.vel = std::stof(dadosString.substr(start, end - start));
			break;
		}
		start = end + delim.length();
		end = dadosString.find(delim, start);
		posicao++;
	}
	dados.tempo = DesserializarTempo(dadosString.substr(start, end - start));
	return dados;
}
```

- DefeitoSuperficieTira.cpp

Esse arquivo tem a mesma funcionalidade do arquivo anterior, mas adaptado para as mensagens de defeito de superfície de tira.

- Tempo.cpp

Esse arquivo tem a finalidade de gerar estruturas de dados que contém o tempo de geração das mensagens de maneira mais adaptada ao trabalho. As funções de serialização e desserialização ocorrem de forma idêntica à forma que ocorre no arquivo anterior. A função de geração de tempo atual faz uso da função GetSystemTime() do Windows. Após a leitura do relógio do computador, a função faz a devida alteração de hora de acordo com o fuso horário e em seguida retorna a estrutura com o tempo correto para o posterior uso no programa.

```
Tempo GerarTempoAtual() {
	Tempo tempo;

	SYSTEMTIME time_windows;
	GetSystemTime(&time_windows);

	tempo.hora = (time_windows.wHour + FUSO_HORARIO < 0) ? 24 -
((-FUSO_HORARIO) - time_windows.wHour) : time_windows.wHour +
FUSO_HORARIO;
	tempo.minuto = time_windows.wMinute;
	tempo.segundo = time_windows.wSecond;
	tempo.milissegundo = time_windows.wMilliseconds;

	return tempo;
}
```

- ListaCircular.cpp

Esse arquivo contém a definição principal da lista circular em memória. A partir dele, outros arquivos podem incluir seu cabeçalho para fazer uso da lista.

A função GetPosicaoPonteiro() retorna a posição atual do ponteiro de escrita da lista de acordo com o tamanho da lista. O operador de módulo foi usada por segurança para impedir acessos inválidos, já que teoricamente a posição correta do ponteiro já é tratada pela função de incremento.

```
int GetPosicaoPonteiro() {
	return Posicao_Ponteiro % TAMANHO_LISTA;
}
```

A função IncrementarPosicaoPonteiro() realiza o incremento do ponteiro da lista circular em memória. Ela também faz o uso do operador de módulo para impedir posições inválidas.

```
void IncrementarPosicaoPonteiro() {
	Posicao_Ponteiro = (Posicao_Ponteiro + 1) % TAMANHO_LISTA;
}
```

A função Print_Snapshot_Lista() será descrita no tópico “Bônus”.

- RandomUtil.cpp

Esse arquivo contém apenas a função GerarIdAleatorio() que têm a finalidade de gerar um id alfanumérico aleatório de acordo com o tipo de mensagem passado por parâmetro. A partir de uma cadeia de caracteres e números possíveis, a função retira caracteres aleatórios dessa cadeia e deposita na string de retorno. Por fim, a função retorna a string aleatória gerada.

```
std::string GerarIdAleatorio(int Tipo_Mensagem) {
	std::string Random_Id;
	static const char caracteres[] =
		"ABCDEFGHIJKLMNOPQRSTUVWXYZ";
	static const char numeros[] =
		"0123456789";

	int Tamanho_Letras = Tipo_Mensagem == MENSAGEM_DEFEITO ? 2 : 3;
	int Tamanho_Numeros = 4;


	for (int i = 0; i < Tamanho_Letras; i++)
		Random_Id += caracteres[rand() % (sizeof(caracteres) - 1)];

	if (Tipo_Mensagem == MENSAGEM_DADO)
		Random_Id += "-";

	for (int i = 0; i < Tamanho_Numeros; i++)
		Random_Id += numeros[rand() % (sizeof(numeros) - 1)];

	return Random_Id;
}
```

- **Mensagens.cpp**

Esse arquivo contém a função necessária por mostrar mensagens coloridas na tela. Essa função recebe como parâmetro a string a ser mostrada e a cor (definida no header Mensagens.h). Em seguida a função conquista o mutex de acesso ao console, de modo que uma cor não interfira na outra, e mostra a mensagem colorida na tela. Por fim, a função libera o mutex.

```
void MostrarMensagem(std::string mensagem, int cor) {
	HANDLE Mutex_Acesso_Console = OpenMutex(SYNCHRONIZE |
MUTEX_MODIFY_STATE, false, "Mutex_Acesso_Console");
	HANDLE Handle_Console = GetStdHandle(STD_OUTPUT_HANDLE);

	WaitForSingleObject(Mutex_Acesso_Console, INFINITE);
	SetConsoleTextAttribute(Handle_Console, cor);
	std::cout << mensagem << std::endl;
	ReleaseMutex(Mutex_Acesso_Console);
}
```

Estratégias

- Encerramento das threads

Para finalizar as threads a partir de eventos, foi utilizada a seguinte estratégia. Ao finalizar a execução de um laço, a thread executa a função WaitForSingleObject passando o evento *Evento_Nao_Finalizar_*** como parâmetro e um tempo de timeout de 0. Caso o evento esteja sinalizado, o retorno da função será WAIT_OBJECT_0, o que significa que a execução do laço deve continuar. Entretanto, caso o evento não esteja sinalizado, o retorno da função será WAIT_TIMEOUT, o que fará com que a condição de execução do laço seja falsa. Assim, o laço será quebrado e a thread será encerrada.

```
do {

	...

	resultadoEvento = WaitForSingleObject(Evento_Nao_Finalizar_**, 0);
} while (resultadoEvento == WAIT_OBJECT_0);
```

- Tipo 0

O tipo 0 presente nas estruturas tem a finalidade de mostrar que aquela não é uma estrutura válida para que o programa possa fazer o tratamento de acordo. Uma estrutura não válida pode ocorrer ao tentar desserializar uma string que não corresponde à estrutura da função chamada ou ao tentar desserializar uma string vazia. É importante ressaltar que posições livres na lista circular em memória são representadas por strings vazias e dessa forma, caso elas sejam lidas, as threads de captura saberão que essa não é a mensagem esperada e passarão para a próxima posição da lista.

```
DefeitoTira defeito = DesserializarDefeitoTira(Proxima_Mensagem_Da_Fila);

if (defeito.tipo == 11) {
	Lista_Circular_Memoria[Ponteiro_Leitura_Defeitos % TAMANHO_LISTA] = "";
	...
	ReleaseSemaphore(Semaforo_Acesso_Lista_Circular_Livres, 1, NULL);
	SetEvent(Evento_Lista_Circular_Nao_Cheia);
}
else {
	//Estrutura com tipo 0
	ReleaseSemaphore(Semaforo_Acesso_Lista_Circular_Ocupados, 1, NULL);
}
```

Bônus

- Snapshot da lista circular

Na tarefa de leitura de comandos do teclado foi adicionado um novo comando que pode ser executado com a tecla “v”. Ao executar esse comando, é mostrada na tela uma snapshot da lista circular em memória, mostrando as posições vazias e ocupadas da lista. Uma posição vazia é representada por “_”, já uma posição ocupada é representada por “X”. Dessa forma, o usuário pode ver o estado da lista e analisar o comportamento das diversas threads presentes no programa.

Um comportamento interessante de ser observado pode ser recriado com os seguintes comandos:

- Apertar as teclas “d” e “e”, de modo a bloquear as threads de captura de dados das mensagens da lista.
- Esperar a mensagem de que a lista circular está cheia e deve apertar a tecla “i” para bloquear a thread de inspeção de defeitos que deposita mensagens na lista.
- Apertar a tecla “d” **ou** “e” de modo a consumir todas as mensagens de um só tipo da lista.

Dessa forma, ao apertar a tecla “v” para observar a lista, podemos ver que ela está bem fragmentada, o que é um resultado esperado já que os tipos de mensagens são depositados com intervalos diferentes na lista.

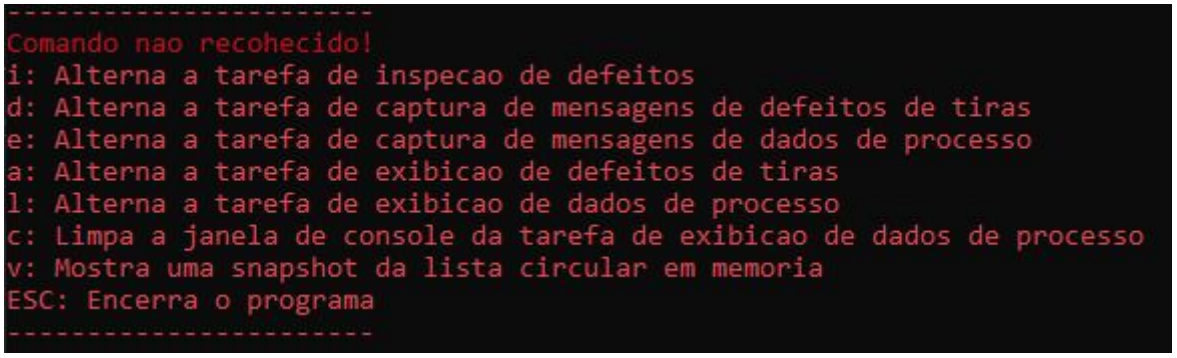
- Cores

Para facilitar a visualização das mensagens na tela, foram usadas diferentes cores para cada tipo de mensagem mostrada na tela. Além disso, foi criado um mutex para implementar a exclusão mútua do console, de modo a impedir que o comando de definir a cor do console de uma thread interfira na outra.

-
- Azul: Mensagens de defeitos de tiras.
- Verde: Mensagens de captura de dados de processo.
- Amarelo: Finalização de threads.
- Vermelho: Sinalização da lista circular cheia.
- Roxo escuro: Snapshot da lista circular em memória.
- Ciano: Bloqueio ou desbloqueio de tarefas por comando do teclado.
- Cinza escuro: Comando de limpeza do console da tarefa de exibição de dados de processo.

- **Comando inválido**

Caso o usuário digite um comando inválido na tela, a tarefa de leitura do teclado mostrará as opções possíveis e uma breve descrição do seu comportamento. Dessa forma, o funcionamento dessa tarefa fica mais claro ao usuário que não precisará voltar à documentação para saber qual comando usar.



Comentários finais

Durante esse trabalho eu busquei organizar o melhor possível a estrutura do código. Sempre que possível um novo arquivo foi criado, de modo que não houvessem muitas responsabilidades do código em um mesmo arquivo. É possível perceber que no caso das threads, cada uma está presente em um arquivo devidamente nomeado. Além disso, todas as variáveis presentes no código possuem nomes bem descritivos da sua função

Um comentário importante que eu gostaria de fazer é que durante a criação e abertura de handles, **não foi realizada a verificação de erros**. Isso se deve ao fato de que por ser uma aplicação simples, em um ambiente controlado e que não trata operações de risco, erros desse tipo são raros de acontecer e mesmo que acontecessem, não ocorreriam falhas catastróficas. Durante toda o desenvolvimento do programa, o único erro em relação aos handles que ocorreu foi por um erro de programação, onde o nome do evento não tinha sido bem definido. A implementação dessa checagem de erros é possível, porém se fosse implementada haveria uma grande poluição do código para uma tarefa que não é absolutamente necessária dadas as circunstâncias. De qualquer forma, compreendo que essa checagem possa fazer parte da avaliação e concordo que sejam descontados pontos da minha implementação nesse caso. Entretanto, eu gostaria de somente um feedback em relação a esse ponto para que eu possa corrigir essa problema na segunda parte caso seja necessário.

Git

Durante todo o desenvolvimento foi feito o uso da ferramenta Git (https://git-scm.com/), para controle de versões. O site GitHub (https://github.com/) foi usado para hospedar o repositório na nuvem por questões de segurança (caso algum defeito ocorresse com o computador) e a ferramenta gráfica GitKraken (https://www.gitkraken.com/) foi utilizada para poder controlar o repositório com maior finalidade.

No repositório é possível encontrar dois *branchs*: **master** e **develop**. O desenvolvimento foi feito diretamente no repositório **develop**, e assim que um fase estável do código fosse atingida, foi feito um *merge* desse branch para o branch **master**.

O repositório pode ser encontrado em:

https://github.com/Cesaario/TrabalhoPraticoIndustrial

Referências

Alguns links de referência usados durante o desenvolvimento do trabalho:

- https://docs.microsoft.com/en-us/windows/win32/api/synchapi/nf-synchapi-waitforsingleobject
- https://docs.microsoft.com/en-us/windows/win32/debug/system-error-codes--0-499-
- https://docs.microsoft.com/en-us/windows/win32/api/synchapi/nf-synchapi-createmutexa
- https://docs.microsoft.com/en-us/windows/win32/api/winbase/nf-winbase-createsemaphorea
- https://docs.microsoft.com/en-us/windows/win32/api/synchapi/nf-synchapi-createeventa
- https://docs.microsoft.com/en-us/windows/win32/api/synchapi/nf-synchapi-waitformultipleobjects
- https://docs.microsoft.com/en-us/windows/win32/api/handleapi/nf-handleapi-closehandle
- https://docs.microsoft.com/en-us/windows/win32/api/synchapi/nf-synchapi-openmutexw
- https://docs.microsoft.com/en-us/windows/win32/api/synchapi/nf-synchapi-opensemaphorew
- https://docs.microsoft.com/en-us/windows/win32/api/synchapi/nf-synchapi-openeventa
- https://docs.microsoft.com/pt-br/cpp/c-runtime-library/reference/sprintf-sprintf-l-swprintf-swprintf-l-swprintf-l?view=vs-2019
- https://docs.microsoft.com/en-us/windows/win32/api/synchapi/nf-synchapi-setevent
- https://docs.microsoft.com/en-us/windows/win32/api/synchapi/nf-synchapi-resetevent
- https://docs.microsoft.com/en-us/windows/win32/api/winbase/nf-winbase-pulseevent
- https://stackoverflow.com/questions/34209854/how-to-detect-the-esc-key-in-c
- https://stackoverflow.com/questions/6486289/how-can-i-clear-console
- https://stackoverflow.com/questions/17008026/windows-how-to-get-the-current-time-in-milliseconds-in-c
- https://docs.microsoft.com/en-us/windows/win32/api/fileapi/nf-fileapi-writefile
- https://docs.microsoft.com/en-us/windows/win32/api/fileapi/nf-fileapi-readfile
- https://docs.microsoft.com/en-us/windows/win32/api/winbase/nf-winbase-createnamedpipea
- https://docs.microsoft.com/en-us/windows/win32/sync/timer-queues