Automação em Tempo Real

Trabalho prático - Sistema de Automação *multithread* para uma fábrica de cimento

Aluno: Luiz Eduardo Lima Coelho - 2016020991

Introdução

Neste trabalho, foi implementado um sistema de automação *multithread* para uma fábrica de cimentos, na linguagem C++, usando o Microsoft Visual Studio, e a api Win32.

O trabalho considera uma aplicação industrial que possui diversas tarefas diferentes, como leitura do CLP, exibição de dados, retirada de mensagens, que devem ser organizados entre Processos e Threads, contando com mecanismos de sincronização e de comunicação entre processos (IPC), assuntos explorados nas aulas de Automação em Tempo Real e muito presentes em diversas aplicações, incluindo sistemas de automação.

Organização entre as tarefas

Haviam 6 tarefas que precisavam ser organizadas entre processos e *threads*. Foi criada uma única solução no Visual Studio, contendo 3 projetos. Cada projeto representa um processo, e em cada processo as tarefas executam como *threads.* O processo principal é chamado de “TrabalhoPratico” e o código fonte do programa foi denominado “processo_exibicao_dados.cpp”, ele é responsável por iniciar suas *threads,* que incluem a maioria das tarefas do sistema, e também inicializa os outros dois processos. O segundo processo é um projeto chamado “processo_gestao_da_producao” de código fonte homônimo, ele é reponsável por exibir na tela as mensagens de gestão de processo formatadas. O último processo é um projeto chamado “processo_leitura_do_teclado”, que recebe entrada do usuário e manda comando de controle para as demais tarefas.

Como todos os projetos estão em uma mesma solução do Visual Studio, é preciso dar apenas um comando para lançar o programa como um todo. Além disso, cada processo foi iniciado de forma que tenha sua própria janela de console, podendo exibir para o usuário as informações de forma organizada.

Processo principal

O processo principal contém 5 threads. A *thread* primária que é o *main()* que é responsável por criar os objetos de sincronização usados pelas demais *threads* como objetos do tipo semáforo e evento. Ela ainda chama a função “_beginthreadex” para lançar 4 *threads*, e a função “CreateProcess” para iniciar os

outros dois processos. Uma vez iniciados todas as tarefas necessárias, a *thread* primária se bloqueia usando WaitForMultipleObjects() até que todas as *threads* tenham terminado de executar. Por fim ela fecha os Handles utilizados.

A primeira *threads* secundária corresponde a tarefa de leitura do CLP. Sua temporização é feita usando o *timeout* de um evento que nunca é sinalizado, por meio da função WaitForMultipleObjects(). E então os campos da mensagem do CLP são gerados. O número de sequência é incrementado de forma que volte a zero depois de atingir o valor máximo usando a função de módulo (resto da divisão). O tempo corrente é obtido por meio da função time() da biblioteca time.h, que retorna a hora corrente local. Os campos numéricos foram gerados aleatoriamente com a função rand(). A semente aleatória utilizada foi inicializada com a função time(). Após formatar toda a mensagem com o auxílio da biblioteca string.h, fazemos uma chamada de função para depositar a mensagem na primeira lista circular. (Explicada posteriormente).

A segunda *thread* secundária, é a tarefa de leitura do sistema de PCP. A temporização para essa tarefa é feita de maneira análoga à da tarefa anterior, porém o tempo em milissegundos de temporização é obtido aleatoriamente entre 1000 e 5000 milissegundos. A geração de campos numéricos é feita de forma análoga também. Porém para a geração dos campos alfanuméricos, foi chamada uma função (definida no início do arquivo) chamada genRandomString(int size_of_string). Essa função recebe um inteiro com o tamanho desejado e retorna uma *string* com essa quantidade de caracteres. Os caracteres que serão incluídos foram definidos de antemão na variável “alphanum” e incluem números e letras maiúsculas. Após construída a mensagem, ela também na lista circular 1, por meio de uma chamada de função.

A próxima *thread* implementa a tarefa de captura de mensagens. Ela lê a lista circular 1, usando uma chamada de função, preenchida com as mensagens de leitura do CLP e do sistema de PCP. Após ler da lista circular 1, ela precisa definir de qual tipo é a mensagem. Se for uma mensagem de leitura do CLP, ela será depositada na lista circular 2, por meio de uma chamada de função. Caso seja uma mensagem do sistema de PCP, ela será enviada ao processo de gestão da produção por meio de um *Pipe* nomeado em que esta *thread* é a cliente e a tarefa de gestão da produção é a servidora. Para a comunicação entre processos, usando *pipes,* escolhemos definir uma comunicação assíncrona. Com isso garantimos que as tarefas ganham mais autonomia e não fiquem muito acopladas umas às outras. Com a escrita assíncrona, mesmo que a tarefa servidora do *pipe* falhe por algum motivo, a *thread* de retirada de mensagens pode continuar funcionando e distribuindo as demais mensagens de forma normal. Porém para que a escrita seja assíncrona, precisamos criar uma estrutura *overlapped* e passá-la como um argumento em WriteFile. Essa estrutura contém um *handle* para um evento que será sinalizado quando a escrita for concluída. Dessa forma a tarefa servidora do *pipe* sabe o momento que poderá ler a mensagem.

A última *thread* desse processo corresponde a tarefa de exibição dos dados de processo. Ela lê da lista circular 2, por meio de uma função para fazê-lo de forma adequada. Em seguida ela extrai os campos da mensagem usando o método “substr”, da biblioteca string.h, para obter substrings da mensagem inteira. Depois de obtidos os campos, eles são formatados de acordo com o requerimento do projeto e imprimidos na tela do console com o auxílio da biblioteca iostream.h. Uma observação a ser feita aqui é que nas especificações de exibição das mensagens de processo, é pedido para imprimir a letra “T” antes dos dados de pressão do processo. Isso foi considerado como um equívoco, e no lugar, utilizamos a letra “V” para indicar os dados de pressão.

Processo de gestão da produção

Esse processo é mais simples e consiste apenas de uma *thread* primária (main) que realiza a tarefa de gestão da produção. Ela deve imprimir em seu console as mensagens de escalonamento da produção de forma formatada. Porém, como as mensagens são geradas em outro processo, precisamos implementar alguma forma de IPC.

Como já foi dito anteriormente, esse processo é o servidor de um *pipe* nomeado de uma única instância, com comunicação assíncrona. Na função de ReadFile, para receber a mensagem, também passamos o apontador para uma estrutura do tipo *overlapped*. Além disso, precisamos conseguir um *handle* para o evento criado no processo principal, que indica quando a operação de escrita foi realizada. Para tanto, usamos a função de OpenEvent, e passamos o nome do evento designado para isso, “PipeEvent”.

Dessa forma, após realizar a leitura do *pipe* com ReadFile, devemos aguardar a sinalização do evento com WaitForSingleObject, para só então usar o conteúdo lido. Uma vez que isso foi feito, podemos ler a mensagem e extrair os campos adequados da *string* para imprimir uma mensagem formatada na tela, de forma semelhante ao que foi feito na tarefa de exibição de dados do processo.

Além disso, esse processo ainda se comunica com a tarefa de leitura do teclado por meio de *Mailslots.* O que será discutido melhor abaixo.

Processo de leitura do teclado

Essa é a última tarefa restante das especificações do projeto, assim como o anterior, esse projeto possui apenas uma *thread* primária (main). Ela é responsável por receber comandos do usuário e agir nas outras *threads* conforme o necessário. Para realizar a leitura dos comandos do usuário (o que inclui a tecla “Esc”), usamos a função _getchar() da biblioteca conio.h. As teclas válidas são (p, s, r, g, d, c , Esc).

Ao digitar outra tecla qualquer, a janela de console imprime uma mensagem de tecla inválida e indica quais as teclas disponíveis.

As 5 primeiras teclas tem a função de bloquear/desbloquear uma das outras 5 tarefas por meio de um semáforo. A estratégia usada para fazer isso foi guardar o estado de cada tarefa em uma variável booleana, sendo que verdadeiro indica que ela funciona normalmente e falso indica que ela está bloqueada. Dessa forma ao apertar uma tecla de bloqueio, esse processo irá bloquear alguma *thread* fazendo uma chamada a WaitForSingleObject() em um semáforo compartilhado por aquela *thread.* Ao apertar a mesma tecla novamente, a variável booleana indica que ela já está bloqueada e portanto é feita uma chamada a ReleaseSemaphore() para liberar o semáforo e reativar a *thead.* Para facilitar os testes e o trabalho do operador, a cada tecla digitada nesse console, é impresso um pequeno painel com o nome das tarefas e imediatamente abaixo consta um número que pode ser 0 ou 1. 0 indica que a tarefa acima dele está bloqueada enquanto 1 indica que ela executa normalmente. É uma ação simples, porém ajuda bastante em termos de visualização do que acontece no sistema.

Outra função implementada nesta tarefa é a letra “c”, que envia uma mensagem via *mailslots* para a tarefa de gestão da produção para que ela limpe sua tela de console, sendo que ela é a servidora do *maislot*. A tarefa de gestão realiza a limpeza de tela usando o comando system("cls") do windows. Devemos lembrar que a tarefa de gestão também realiza comunicação assíncrona usando *pipes*, e se bloqueia na leitura do mesmo, portanto, após cada leitura do *pipe* e execução do *loop principal* é feita uma checagem no *mailslot* por mensagens de limpeza de tela.

Devemos observar as limitações dessa função, já que se a tarefa de gestão de produção estiver bloqueada pelo comando “g”, a mesma não pode limpar a tela. E ainda, se a mesma estiver bloqueada esperado uma mensagem do *pipe*, (caso a leitura do PCP ou a retirada de mensagens estiverem bloqueadas por exemplo), a tarefa também não poderá apagar a tela imediatamente. Porém o fará após se desbloquear.

A última função realizada por esta tarefa é a leitura da tecla Esc. Ao fazer isso, ela sinaliza um objeto evento, dizendo a todas as outras *threads* que eles devem encerrar suas execuções. Para isso funcionar de maneira adequada, em cada ponto de bloqueio das *threads* foi feito um WaitForMultipleObjects em vez de um WaitForSingleObject. A primeira função permite que se aguarde simultaneamente pela condição original de espera e pelo evento de fim de programa. Dessa forma passamos o parâmetro de WaitForAll do WaitForMultipleObjects como falso, para que independente da condição de bloqueio, a *thread* se encerre quando ocorrer um evento de Esc.

Listas circulares

Realizar escrita e leitura em listas circulares não é algo muito trivial em ambientes *multithread.* Devemos garantir que dados ainda não lidos não sejam sobrescritos e que variáveis compartilhadas não sejam acessadas ao mesmo tempo. Como as listas circulares são acessadas várias vezes no programa, foi decidido que seriam criadas funções para leitura e inserção padrão nas duas listas. Dessa forma, caso seja preciso mudar algum detalhe, poderia ser feito uma única vez nas chamadas de função, e não repetidas vezes ao longo do programa.

Para se assegurar que os dados não sejam sobrescritos, foram usados dois semáforos contadores do tamanho da lista, um indicando a quantidade de posições livres na lista, e outro indicando as posições ocupadas. Assim, as *threads* "produtoras" se bloqueiam caso a lista esteja cheia, e as "consumidoras" se bloqueiam caso não haja dados disponíveis. Além disso, usamos um semáforo binário que funciona como um mutex para evitar o acesso mútuo a variáveis compartilhadas, como a própria lista circular, e seus índices. O algoritmo utilizado foi adaptado do problema de produtores e consumidores visto na disciplina, para funcionar com mais de uma *thread* produtora.

O tipo de dado usado para as listas foram a estrutura de vetor disponibilizada pela biblioteca vector.h, sendo que o tipo base eram strings da biblioteca string.h. O uso dessas bibliotecas ajuda bastante o desenvolvimento e facilita a manipulação de mensagens usando strings.

Objetos de sincronização utilizados

A tabela a seguir lista os objetos de sincronização utilizados com uma breve descrição de seu uso.

TIPO	**NOME**	**FUNÇÃO**
Semáforo	Livres_lista1	Quantidade de posições livres na lista circular 1
Semáforo	Ocupados_lista1	Quantidade de posições ocupadas na lista circular 1
Semáforo	Mutex_lista1	Garantir exclusão mútua para variáveis da lista circular 1

Semáforo	Livres_lista2	Quantidade de posições livres na lista circular 2
Semáforo	Ocupados_lista2	Quantidade de posições ocupadas na lista circular 2
Semáforo	Mutex_lista2	Garantir exclusão mútua para variáveis da lista circular 2
Semáforo	Controla_leitura_clp	Bloqueio da tarefa de leitura do CLP por comandos do usuário
Semáforo	Controla_leitura_pcp	Bloqueio da tarefa de leitura do PCP por comandos do usuário
Semáforo	Controla_retirada_de_mensagens	Bloqueio da tarefa de retirada de mensagens por comandos do usuário
Semáforo	Controla_sistema_de_gestao	Bloqueio da tarefa do sistema de gestão por comandos do usuário
Semáforo	Controla_sistema_de_exibicao_de_dados	Bloqueio da tarefa de exibição de dados do processo por comandos do usuário
Evento	EscEvent	Sinaliza evento de fim do programa por comando do usuário
Evento	hEvent	Evento usado para temporização de WaitForMultipleObjects(), nunca é sinalizado
Evento	hPipeEvent	Evento que sinaliza término da escrita de uma mensagem no pipe. Libera leitura do servidor do pipe.

Comentários sobre o desenvolvimento do programa

Vemos que grande parte do programa está concentrado no processo principal. Por um lado isso significa que ganhamos em eficiência no sentido que não é preciso salvar o contexto para chaveamento entre *threads,* o que precisa ocorrer em caso de processos diferentes. Além disso, podemos criar *handles* globais, sem a necessidade de herdá-los entre processos filhos ou precisar abrir novamente um objeto de sincronização. Porém isso deixa o código menos organizado e o entendimento um pouco mais complexo.

O uso de *pipes* com mensagens assíncronas permite desacoplar a tarefa de retirada de mensagens com a de gestão da produção. Note que bloqueando a última, o sistema de exibição de dados continua a receber as mensagens.

Implementar o envio de mensagens de limpar a tela por meio de *mailslots* foi uma maneira simples de fazê-lo, mas utilizando dois métodos de IPC faz com que a limpeza de tela não ocorra imediatamente, se a tarefa estiver aguardando uma mensagem do *pipe.*

A ordem de chamada das das operações de sincronização foi pensada para evitar a ocorrência de *deadlock.* Por exemplo, os semáforos de bloqueio das tarefas recebem um *Release* logo após o *Wait*, impedindo que a tarefa de leitura do teclado seja bloqueada. Além disso, o *mutex* das listas circulares são conquistados e liberados depois de conquistar o semáforo e antes de liberá-lo.

Conclusão

No desenvolvimento desse programa, foi possível ter um contato prático com diversos conceitos e ideias desenvolvidos ao longo do curso de automação em tempo real. Entre eles então, aplicações *multithread*, exclusão mútua, mecanismos de sincronização, *deadlocks*, temporização e comunicação entre processos.

Foram utilizados vários recursos da api do Windows, o que é uma forma de ter familiaridade com esse forma de programação, porém sem perda de generalização das práticas e conceitos de automação em tempo real. Além disso, foi uma oportunidade de praticar escolhas em desenvolvimento de projetos de *software*, que muitas vezes consistem de *tradeoffs*.

Bibliografia

- Slides da disciplina de Automação em Tempo Real
- Exercícios e exemplos disponibilizados pelo professor da disciplina
- Programação Concorrente Em Ambiente Windows.
  Constantino Seixas Filho e Marcelo Szuster
- Exemplos de programação do livro de Constantino Seixas Filho e Marcelo Szuster (acima)
- Página de informações da linguagem C++ http://www.cplusplus.com/
- Página de referência para códigos de erro do Windows https://docs.microsoft.com/en-us/windows/desktop/debug/system-error-codes
- Página da comunidade que serviu de inspiração para a função de geração de *strings* alfanuméricas aleatórias
  http://www.cplusplus.com/forum/windows/88843/