UFMG

CURSO DE ESPECIALIZAÇÃO EM AUTOMAÇÃO INDUSTRIAL

Fundamentos e Aplicações de Sistemas de Automação

Módulo 6:
IPC – Inter Process Communication

Quem não se comunica se trumbica
Abelardo Barbosa Rodriguez – o Chacrinha

Professor: Constantino Seixas Filho

sábado, 7 de maio de 2005

1

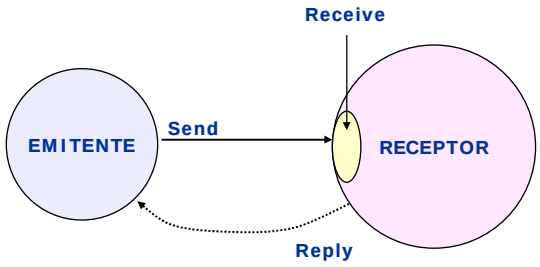
Comunicação entre threads e processos

UFMG

- Constituem paradigma para a construção de aplicações modulares e bem distribuídas
- No S.O. QNX é o modelo básico para construção de aplicações locais e distribuídas. Para o S.O. é indiferente se as unidades que se comunicam estão no mesmo computador ou em máquinas diferentes
- No QNX mensagens asseguram o compartilhamento de informação com exclusão mútua já que a informação é transferida da memória local de uma thread para a memória local de outra thread
- QNX = A Message Passing Operating System

2

Comunicação síncrona

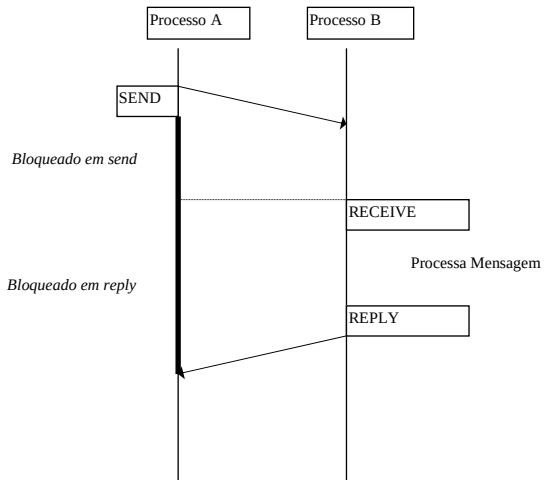


- As operações de Send e Receive são bloqueantes
- A resposta é dada pela mensagem de Reply que é assíncrona
- Send e Reply podem transferir de 0 a 64 kbytes em cada operação
- O que acontece se ao invés de emitir um Receive, o receptor também coloca uma mensagem para o emitente através de outro Send ?

DEADLOCK !

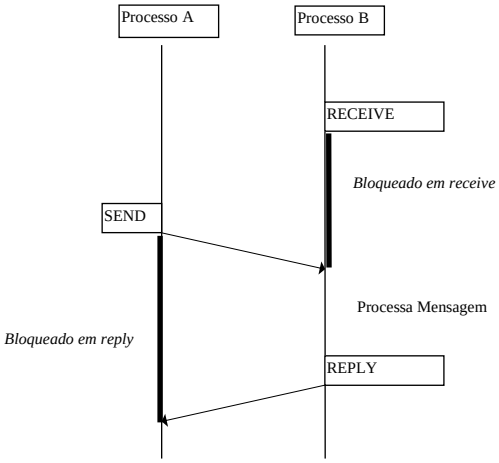
3

A envia mensagem antes de B emitir *receive*



4

B emite *receive* antes de A enviar mensagem

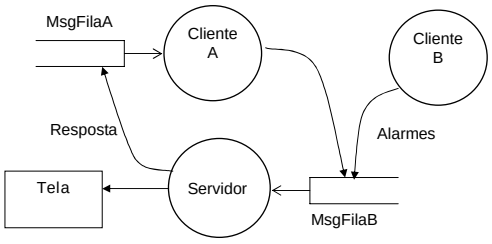


5

Comunicação Assíncrona

Vantagens da comunicação Assíncrona:

- **Ausência de bloqueio dos processos comunicantes, o que implica em menor em menor overhead na comunicação**
- **Inexistência de deadlock**
- **Maior modularidade e escalabilidade de aplicativos. Exemplo: Vários aplicativos podem colocar mensagens de alarme em uma fila definida por outra aplicação.**



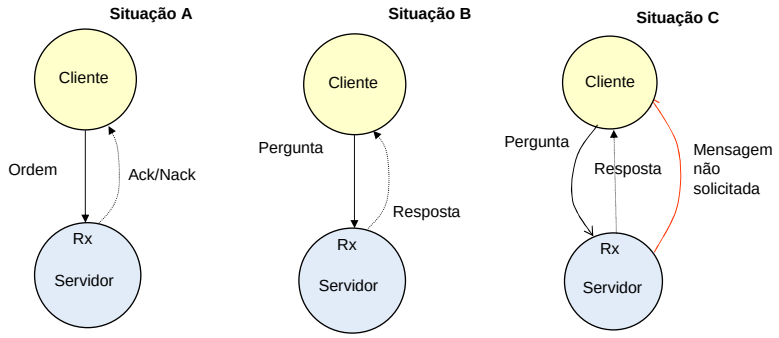
6

Relacionamentos básicos

a) O processo cliente envia uma ordem ao processo servidor e recebe uma confirmação de que a mensagem foi recebida (*acknowledge = ack/nack*)

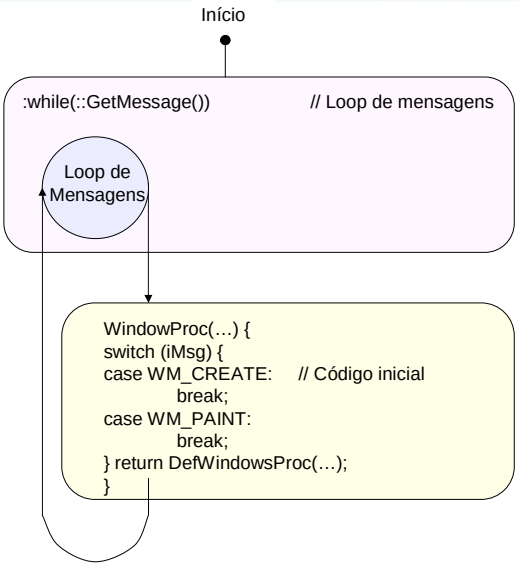
b) O processo cliente faz uma pergunta ao servidor e recebe uma mensagem como resposta

c) O processo servidor envia uma *unsolicited message* ao processo cliente quando uma situação anormal é detectada. Este tipo de comunicação é utilizada para implementar arquiteturas variantes do conceito mestre-escravo tradicional representado pelas situações a) e b). Na situação c) o servidor pode receber também solicitações dos tipos a) e b)



7

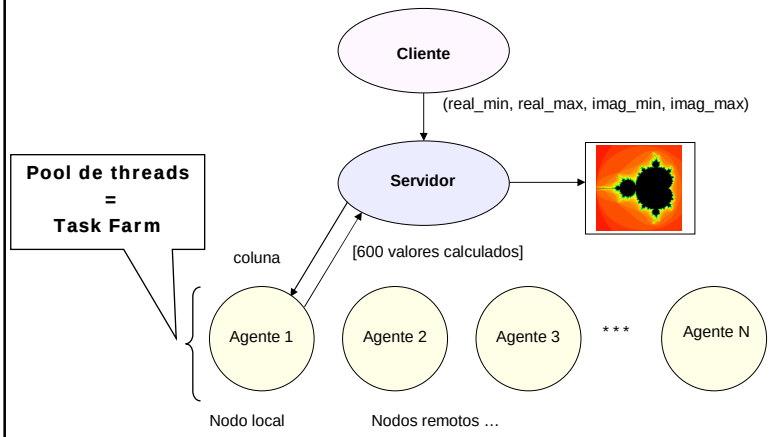
Mecanismo básico de funcionamento do Windows



8

Quando é vantajoso se distribuir uma aplicação ?

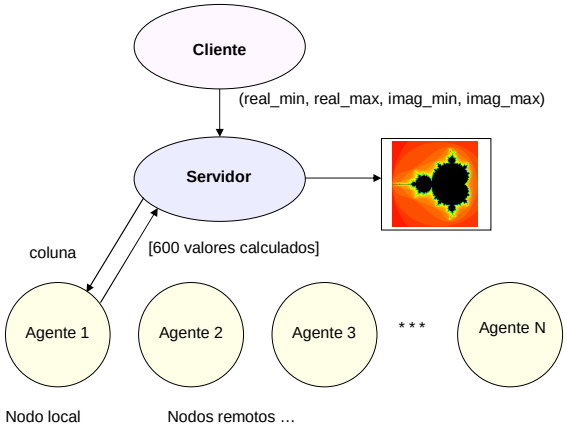
- O tempo de comunicação entre os processos nos diferentes nodos, usado para que o processo mestre especifique as tarefas a serem executadas e para que os processos escravos enviem o resultado do processamento, deve ser muito menor que o tempo de execução das tarefas



9

Quando é vantajoso se distribuir uma aplicação ?

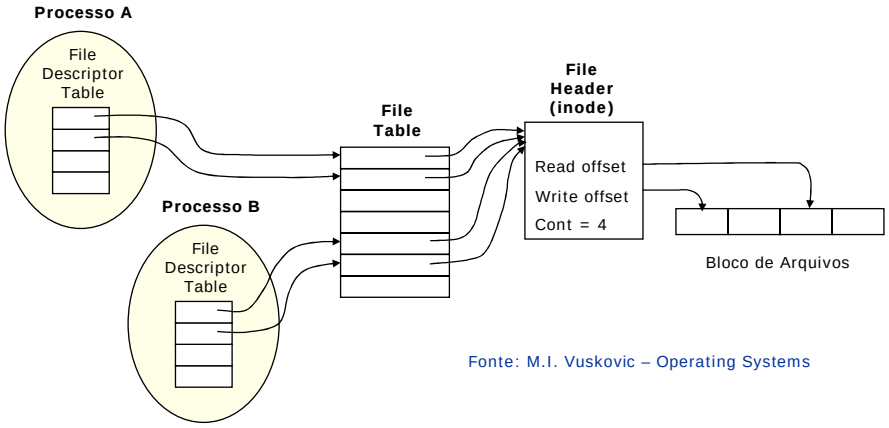
1. Discuta como implementar esta comunicação com o conjunto de mensagens: ***send***, ***receive*** e ***reply*** do QNX
2. Discuta como resolver o mesmo problema num ambiente monoprocessador dividindo as tarefas entre diversas threads com o uso de ***completion ports***



10

Pipes

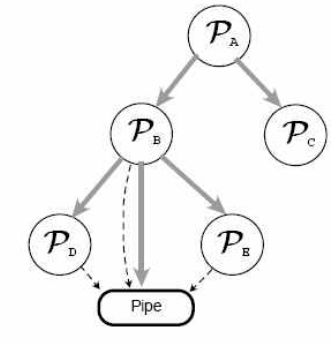
- Pipes surgiram no S.O. UNIX e foram muito aperfeiçoados no WNT
- Pipes são implementados no sub sistema de arquivos
- Pipes são como arquivos com dois offsets: um para escrita e outro para leitura
- A leitura e a escrita seguem disciplina FIFO: *First In First Out*
- Pipes são residentes em memória e não em disco

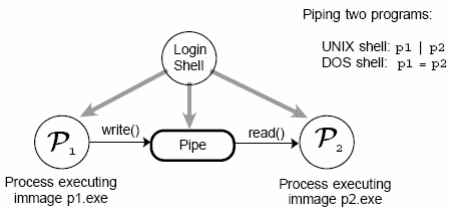


Fonte: M.I. Vuskovic – Operating Systems

Pipes anônimos

- Como Pipes anônimos não têm nome, apenas processos correlacionados podem utilizar este mecanismo para comunicação
- A herança de handles para pipes funciona apenas entre pai e filhos
- São usados geralmente pelo shell para conectar dois programas: o stdout de um programa com o stdin do outro





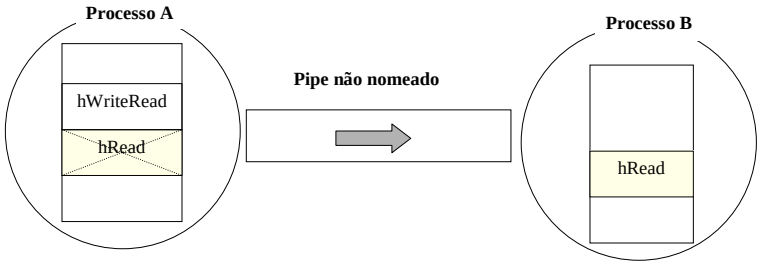
Fonte: Marcos Vuskovic 2002

Comunicação entre Processos
Pipes anônimos

Os pipes anônimos são usados para transferir dados entre dois processos interrelacionados, tais como pai e filho ou dois processos irmãos

Suas principais limitações são:

- **São unidirecionais. Um dos processos deve operar como leitor e o outro como escritor**
- **São locais. Não podem ser usados para comunicação via rede**
- **Possibilitam apenas comunicação síncrona entre processos**
- **Não definem fronteiras entre mensagens. Cada mensagem é apenas um string de bytes**



13

CreatePipe

```
BOOL CreatePipe (
PHANDLE phReadPipe,                    // Endereço para receber handle de leitura
PHANDLE phWritePipe,                   // Endereço para receber handle de escrita
LPSECURITY_ATTRIBUTES lpPipeAttributes, // Apontador para atributos de segurança.
                                       // Se NULL os handles não serão herdáveis.
DWORD nSize                            // Tamanho do buffer reservado ao pipe.
                                       // 0: WNT usa tamanho default.
);
```

Retorno da função:

Status	Interpretação
TRUE	Sucesso
FALSE	Falha

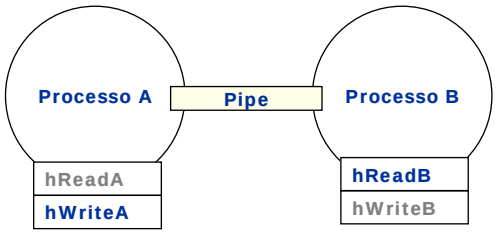
14

Como obter um handle para o pipe no outro processo ?

- O processo A cria um pipe
- O processo B precisa obter um handle para aquele pipe, mas o pipe não tem nome. Não se esqueça que handles são locais a um processo

Problemas:

- Necessitamos de uma outra forma de comunicação para passar a mensagem para o outro processo
- O handle, mesmo que passado para o outro lado, seria inválido



Vamos pedir ajuda ao Sistema Operacional

15

DuplicateHandle

```
HANDLE DuplicateHandle(
HANDLE hSourceProcessHandle,    // Handle do processo que fornece o handle
HANDLE hSourceHandle,           // Handle a ser duplicado
HANDLE hTargetProcessHandle,    // Handle do processo que receberá o handle.
HANDLE lpTargetHandle,          // Apontador para handle duplicado.
HANDLE dwDesiredAccess,         // Tipo de acesso do novo handle .
BOOL bInheritHandle,            // TRUE: o novo handle pode ser herdado por novos processos criados pelo processo alvo.
DWORD dwOptions                 // Opções:
                                • DUPLICATE_CLOSE_SOURCE: Fecha o handle fonte. Isto irá ocorrer independente de qualquer status de erro retornado.
                                • DUPLICATE_SAME_ACCESS: O handle duplicado terá o mesmo acesso do handle fonte
);
```

Retorno da função:

Status	Interpretação
TRUE	Sucesso
FALSE	Falha

16

Criação do canal de comunicação
Alternativa 1:
O processo fonte usa DuplicateHandle()

- O processo A (cliente) cria o processo B e obtém um handle para ele, que pode ser usado em DuplicateHandle()
- O processo A cria um pipe com atributo não herdável e obtém um handle de leitura e outro de escrita. Vamos supor que A é o escritor e que B seja o leitor
- O processo A usa *DuplicateHandle()* para criar uma cópia do handle de leitura dentro da tabela de B. Esta operação é simples: A obtém um handle para si próprio usando *GetCurrentProcess()* a ser usado como handle do processo fonte. A obteve o handle do processo alvo quando criou o processo B. O segundo parâmetro: handle a ser duplicado também é conhecido, é o handle de leitura do pipe, logo o quadro está completo. Uma entrada contendo um descritor para o pipe é criada na tabela de objetos do processo B e um handle correspondendo ao índice deste objeto na tabela de B é retornado para A, mas o processo B desconhece onde esta entrada está localizada em sua própria tabela
- O processo A deve enviar o valor do handle do pipe na tabela do processo B para o Processo B. Como isto é feito ? Utilizando outra forma de comunicação entre processos, por exemplo pipes nomeados ou memória compartilhada. Mas eu preciso usar outra forma de IPC para inicializar uma forma de IPC ??? Isto parece ridículo a qualquer programador !! Nós não conhecemos uma melhor solução para o problema indo por este caminho
- Vamos escolher outra alternativa

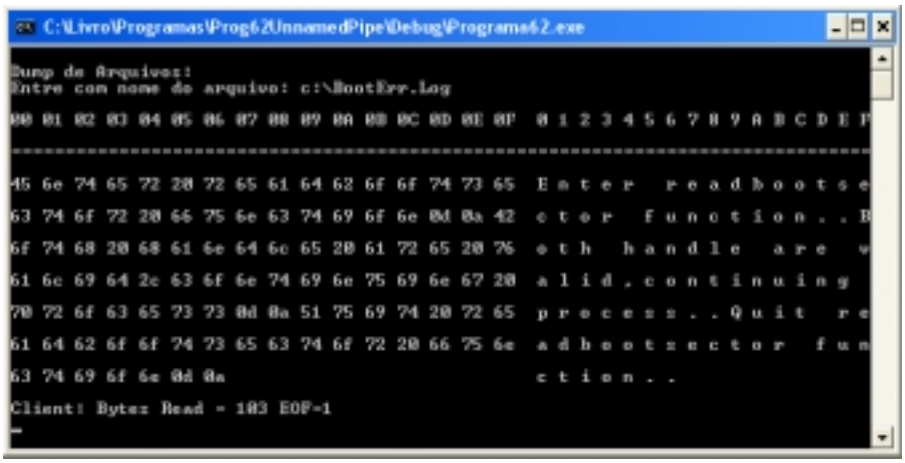
17

Alternativa 2:
O processo alvo usa *DuplicateHandle()*

- O processo A cria um pipe com atributo não herdável e obtém um handle de leitura e outro de escrita. Vamos supor que A é o escritor e que B seja o leitor
- Processo A (cliente) cria o processo B. O processo A passa para seu filho via linha de comando o seu *ProcessId*, e o valor do handle de leitura do pipe. Este valor só é válido no escopo de A
- O processo B usa *DuplicateHandle()* para criar uma cópia do handle de leitura dentro da tabela de B. Como isto é feito ? O handle para o processo que possui o handle do pipe (processo A) é obtido utilizando-se a função *OpenProcess()* a partir do ProcessId de A, com o parâmetro PROCESS_DUP_HANDLE ativado. A obtém um handle para si próprio usando *GetCurrentProcess()* a ser usado como handle do processo destino. O segundo parâmetro: handle a ser duplicado foi recebido por B na linha de comando. Uma entrada contendo um descritor para o pipe é criada na tabela de objetos do processo B e um handle correspondendo ao índice deste objeto na tabela de B é retornado para B.
- Pronto! Podemos usar *FileRead()* agora. Só mais uma observação: *DuplicateHandle()* foi chamada com o primeiro parâmetro, DUPLICATE_CLOSE_SOURCE ativado para apagar o handle copiado do escopo de A. Tudo *comme il faut* !

18

Exemplo: Dump de Arquivos



Dump - Cliente

```
#define     ESC                     0x1B

int main()
{
    char FileName[50];
    int iBloco;                     // Bloco de memória de 512 caracteres
    DWORD dwBytesRead;
    DWORD dwBytesWritten;
    DWORD dwRegLength;              // Tamanho do registro a ser lido do arq em bytes
    LONG lFilePosLow=0L;            // Offset dentro do arquivo
    BOOL bStatus;
    PROCESS_INFORMATION NewProcess; // Info sobre novo processo criado
    int iTecla;
    STARTUPINFO si;                 // StartUpInformation para novo processo
    char LinhaDeComando[30];
    FileReadDesc DumpRequest;
    DumpFileAck DumpAck;
    // Handles do pipe no sentido Cliente Servidor
    HANDLE hCSReadPipe;                 // Handle para operações de leitura
    HANDLE hCSWritePipe;                // Handle para operações de escrita
    // Handles do pipe no sentido Sevidor Cliente
    HANDLE hSCReadPipe;                 // Handle para operações de leitura
    HANDLE hSCWritePipe;                // Handle para operações de escrita
```

Dump - Cliente

```
bStatus=CreatePipe(&hCSReadPipe, &hCSWritePipe,
        NULL,       // lpPipeAttributes
        0);         // tamanho do bufferCreatePipe();
CheckForError(bStatus);
// Cria pipe não nomeado no sentido Servidor Cliente
bStatus=CreatePipe(&hSCReadPipe, &hSCWritePipe,
        NULL,       // lpPipeAttributes
        0);         // tamanho do bufferCreatePipe();
CheckForError(bStatus);
// Cria processo servidor
ZeroMemory(&si, sizeof(si));
si.cb = sizeof(si);             // Tamanho da estrutura em bytes
sprintf(LinhaDeComando, "%u %u %u", GetCurrentProcessId(), hCSReadPipe, hSCWritePipe);
bStatus = CreateProcess(
        "c:\\Livro\\Programas\\Prog62Server\\Debug\\Programa62Server.exe", // Nome
        LinhaDeComando,                 // linha de comando: Passa id do processo
        NULL,                           // atributos de segurança: Processo
        NULL,                           // atributos de segurança: Thread
        FALSE,                          // herança de handles
        NORMAL_PRIORITY_CLASS,          // CreationFlags
        NULL,                           // lpEnvironment
        "c:\\Livro\\Programas\\Prog62Server\\Debug",         // dir corrente do filho
        &si,                            // lpStartUpInfo
        &NewProcess);           /       / lpProcessInformation
CheckForError(bStatus);
```

Dump - Cliente

```
printf("\nDump de Arquivos: ");          // Interage com o usuário
printf("\nEntre com nome do arquivo: ");
scanf("%s", FileName);
iBloco = 0;
dwRegLength = 128L;
strcpy(DumpRequest.FileName, FileName);
DumpRequest.Length = dwRegLength;
do {
        DumpRequest.InitialPosition = iBloco * dwRegLength;
        // Envia dados dos pedidos
        WriteFile(hCSWritePipe, &DumpRequest, sizeof(DumpRequest), &dwBytesWritten, NULL);
        // Recebe acknowledge da transação
        ReadFile(hSCReadPipe, &DumpAck, sizeof(DumpAck), &dwBytesRead, NULL);
        if (DumpAck.Erro > 0) {
           if (DumpAck.Erro == ERRO_ARQUIVO_INEXISTENTE)  printf("\nArquivo Inexistente\n");
             else printf("\nErro na operacao com o arquivo\n");
           Sleep(3000);
           break;
        }
        printf("Client: Bytes Read = %d EOF=%d \n", DumpAck.BytesRead, DumpAck.EndOfFile);
```

22

Dump - Cliente

```
    do
            iTecla = _getch();
            while ((iTecla != ESC) && (iTecla != '+') && (iTecla != '-'));
            if ((iTecla == '+') && (!DumpAck.EndOfFile)) { lFilePosLow += dwRegLength; iBloco++; }
            if ((iTecla == '-') && (iBloco >0)) { lFilePosLow -= dwRegLength; iBloco--; }
            if (iTecla == ESC) {
                DumpRequest.Length = 0;       // Flag para abortar servidor
                WriteFile(hCSWritePipe, &DumpRequest, sizeof(DumpRequest), &dwBytesWritten, NULL);
                break;
            } // if
    } while (1);
    CloseHandle(hCSWritePipe);
    CloseHandle(hSCReadPipe);
    // Aposto que você se esqueceu destes handles!!!
    CloseHandle(NewProcess.hProcess);
    CloseHandle(NewProcess.hThread);

    printf("\nAperte uma tecla...\n");
    _getch();

    ExitProcess(0);
    return EXIT_SUCCESS;
}   // main
```

Aqui aborta o Servidor

23

Dump - Servidor

```
void ShowDump(char *, int);

int main(int argc, char *argv[])     // Receberá id do processo pai como parâmetro e handles para os dois pipes
{
    DWORD dwExitCode = 0;
    char FileName[50]= "";           // Nome do arquivo a ser lido
    HANDLE hFile= 0;                 // Handle para arquivo a ser aberto
    DWORD dwBytesRead;
    DWORD dwBytesWritten;
    LONG lFilePosLow=0L;             // Offset para posicao corrente do arquivo
    BOOL bStatus;
    char MyBuffer[512];              // Buffer para leitura do arquivo
    FileReadDesc DumpRequest;        // Descritor do pedido de leitura
    DumpFileAck DumpAck;             // Resposta do servidor para cliente
    DWORD dwLength;
    DWORD dwFatherId;                // Id do processo pai
    // Handle do pipe no sentido Cliente Servidor
    HANDLE hCSReadPipe;              // Handle para operações de leitura
    // Handle do pipe no sentido Sevidor Cliente
    HANDLE hSCWritePipe;             // Handle para operações de escrita
    HANDLE hFather;                  // Handle do processo pai
    HANDLE hPipe1;                   // Handle para o pipe no dominio do cliente
    HANDLE hPipe2;                   // Handle para o pipe no domínio do cliente
    HANDLE hMyProcess;               // Handle para o próprio processo
```

24

Dump - Servidor

```
// Obtém handle para o processo pai
dwFatherId = atoi(argv[0]);
hPipe1   = (HANDLE) atoi(argv[1]);
hPipe2   = (HANDLE) atoi(argv[2]);
hFather  = OpenProcess(PROCESS_DUP_HANDLE, FALSE, dwFatherId);
CheckForError(bStatus);
hMyProcess = GetCurrentProcess();
// Cria handle Cliente-Servidor de leitura no espaço do cliente
bStatus = DuplicateHandle(
        hFather,                        // Processo que fornece o handle
        hPipe1,                         // Handle a ser copiado
        hMyProcess,             // Processo que receberá o handle
        &hCSReadPipe,                   // Valor do novo handle
        GENERIC_READ,                   // FLAGS
        FALSE,                          // Handle não herdável
        DUPLICATE_CLOSE_SOURCE          // Opções: apaga o handle de origem
        );
CheckForError(bStatus);

// Cria handle Servidor-Cliente de escrita no espaço do servidor
bStatus = DuplicateHandle(
        hFather,                        // Processo que fornece o handle
        hPipe2,                         // Handle a ser copiado
        hMyProcess,             // Processo que receberá o handle
        &hSCWritePipe,                  // Valor do novo handle
        GENERIC_WRITE,                  // FLAGS
        FALSE,                          // Handle não herdável
        DUPLICATE_CLOSE_SOURCE          // Opções: apaga o handle de origem
        );
CheckForError(bStatus);
```

Dump - Servidor

```
do {    // Recebe pedido de dump via pipe
        bStatus=ReadFile(hCSReadPipe, &DumpRequest, sizeof(DumpRequest), &dwBytesRead, NULL);
        CheckForError(bStatus);
        lFilePosLow = DumpRequest.InitialPosition;
        dwLength = DumpRequest.Length;
        if (dwLength==0) break;   // Flag para abortar processo servidor
        if (strcmp(FileName, DumpRequest.FileName) != 0) {
                strcpy(FileName, DumpRequest.FileName);
                if (hFile != 0) CloseHandle(hFile);
                hFile= CreateFile(      // Abre novo arquivo
                        FileName,
                        GENERIC_READ|GENERIC_WRITE,
                        FILE_SHARE_READ|FILE_SHARE_WRITE,
                        NULL,                   // atributos de segurança
                        OPEN_EXISTING,          // abre se existir
                        FILE_ATTRIBUTE_NORMAL,
                        NULL);                  // Template para atributos e flags
                CheckForError(hFile != INVALID_HANDLE_VALUE);
                if (hFile == INVALID_HANDLE_VALUE) {
                    DumpAck.Erro = ERRO_ARQUIVO_INEXISTENTE;
                    // Envia Acknowledge para Cliente
                    WriteFile(hSCWritePipe, &DumpAck, sizeof(DumpAck), &dwBytesWritten, NULL);
                    break;
                } // if
        }
```

Neste ponto eu aborto o servidor

Dump - Servidor

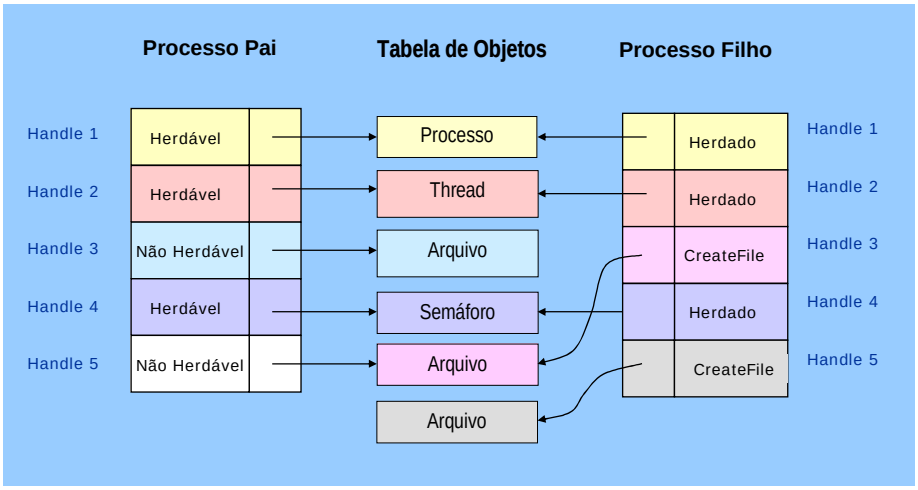
```
            // Le registro desejado
            lFilePosLow = SetFilePointer(hFile,lFilePosLow,NULL,FILE_BEGIN);
            bStatus =ReadFile(hFile, &MyBuffer, dwLength, &dwBytesRead, NULL);
            CheckForError(bStatus);
            DumpAck.EndOfFile = (dwBytesRead != dwLength)? TRUE: FALSE;

            ShowDump(MyBuffer, (int)dwBytesRead);

            // Envia status da leitura para processo cliente
            DumpAck.BytesRead = dwBytesRead;
            DumpAck.Erro = 0;

            // Envia Acknowledge para Cliente
            WriteFile(hSCWritePipe, &DumpAck, sizeof(DumpAck), &dwBytesWritten, NULL);

    } while (1);
    CloseHandle(hFile);
    CloseHandle(hFather);
    CloseHandle(hSCWritePipe);
    CloseHandle(hCSReadPipe);
    return EXIT_SUCCESS;
}   // main
```

27

Dump - Servidor

```
void ShowDump(char *lpBuffer, int iLength)
{   int i=0;
    int iColuna;
    int iLinha;
    int iLinhas;
    int iResto;
    iLinhas = iLength / 16;
    iResto = iLength %  16;

    // Imprime cabeçalho
    printf("\n00 01 02 03 04 05 06 07 08 09 0A 0B 0C 0D 0E 0F  0 1 2 3 4 5 6 7 8 9 A B C D E F\n");
    printf("=================================================================================\n");
    for (iLinha=0; iLinha<iLinhas; iLinha++)  {
       for (iColuna=0; iColuna<16; ++iColuna)  printf("%02x ", lpBuffer[i++] & 0xFF);
            i-= 16;
            printf(" ");
            for (iColuna=0; iColuna<16; ++iColuna)
                if (isprint(lpBuffer[i]))  printf("%c ", lpBuffer[i++]);  else { printf(". "); i++; }
            printf("\n");
    }  // for
    if (iResto > 0) {  // imprime última linha
        for (iColuna=0; iColuna<16; ++iColuna)
            if (iColuna < iResto) printf("%02x ", lpBuffer[i++] & 0xFF);  else printf("   ");
        i-= iResto;
        printf(" ");
        for (iColuna=0; iColuna<iResto; ++iColuna)
            if (isprint(lpBuffer[i]))  printf("%c ", lpBuffer[i++]);  else { printf(". "); i++; }
            printf("\n\n");
    }
} // ShowDump
```

Pipes Anônimos – Uso de Herança



29

Pipes Anônimos – Uso de Herança Cliente

```
//   Capítulo 6 - Exemplo 2b - Pipes Não Nomeados - Com Herança  - Programa Cliente

#define     ESC                     0x1B

int main()
{    char FileName[50];
     int iTecla, iBloco;                        // Bloco de memória de 512 caracteres
     DWORD dwBytesRead;
     DWORD dwBytesWritten;
     DWORD dwRegLength;                         // Tamanho do registro a ser lido do arquivo em bytes
     LONG lFilePosLow=0L;                       // Offset dentro do arquivo
     BOOL bStatus;
     PROCESS_INFORMATION NewProcess;            // Informações sobre novo processo criado
     STARTUPINFO si;                            // StartUpInformation para novo processo
     SECURITY_ATTRIBUTES sa;                    // Atributos de segurança do pipe
     char LinhaDeComando[30];
     FileReadDesc DumpRequest;
     DumpFileAck DumpAck;

     // Handles do pipe no sentido Cliente Servidor
     HANDLE hCSReadPipe;          // Handle para operações de leitura
     HANDLE hCSWritePipe;         // Handle para operações de escrita
     // Handles do pipe no sentido Sevidor Cliente
     HANDLE hSCReadPipe;          // Handle para operações de leitura
     HANDLE hSCWritePipe;         // Handle para operações de escrita
```

30

Pipes Anônimos – Uso de Herança Cliente

```
// Define o título da janela
SetConsoleTitle("Programa 6.2b - Pipes nao nomeados com heranca");

sa.nLength = sizeof(SECURITY_ATTRIBUTES);
sa.lpSecurityDescriptor = 0;
sa.bInheritHandle = TRUE;      // Handles dos pipes serão herdáveis

// Cria pipe não nomeado no sentido Cliente Servidor
bStatus=CreatePipe(&hCSReadPipe, &hCSWritePipe,
        &sa,        // lpPipeAttributes
        0);         // tamanho do bufferCreatePipe();
CheckForError(bStatus);

// Cria pipe não nomeado no sentido Servidor Cliente
bStatus=CreatePipe(&hSCReadPipe, &hSCWritePipe,
        &sa,        // lpPipeAttributes
        0);         // tamanho do bufferCreatePipe();
CheckForError(bStatus);

// Cria processo servidor
ZeroMemory(&si, sizeof(si));
si.cb = sizeof(si);              // Tamanho da estrutura em bytes

sprintf(LinhaDeComando, "%u %u", hCSReadPipe, hSCWritePipe);
```

31

Pipes Anônimos – Uso de Herança Cliente

```
bStatus = CreateProcess(
        "c:\\Livro\\Programas\\Prog62hUPipeServer\\Debug\\Prog62hUPipeServer.exe",// Nome
        LinhaDeComando,                  // linha de comando: Passa id do processo como parâmetro
        NULL,       // atributos de segurança: Processo
        NULL,       // atributos de segurança: Thread
        TRUE,       // Todos os handles herdável do pai serão recebidos pelo filho
        NORMAL_PRIORITY_CLASS,          // CreationFlags
        NULL,       // lpEnvironment
        "c:\\Livro\\Programas\\Prog62hUPipeServer\\Debug",       // diretório corrente do filho
        &si,                     // lpStartUpInfo
        &NewProcess);            // lpProcessInformation
CheckForError(bStatus);

// Fecha handles que irá mais usar
CloseHandle(hCSReadPipe);
CloseHandle(hSCWritePipe);

// Interage com o usuário
cout << "Dump de Arquivos:" << endl;
cout << "Entre com Nome do arquivo:" << endl;
scanf("%s", FileName);
iBloco = 0;
dwRegLength = 128L;
strcpy(DumpRequest.FileName, FileName);
DumpRequest.Length = dwRegLength;
```

32

Pipes Anônimos – Uso de Herança Cliente

```
do {    DumpRequest.InitialPosition = iBloco * dwRegLength;
        // Envia dados dos pedidos
        WriteFile(hCSWritePipe, &DumpRequest, sizeof(DumpRequest), &dwBytesWritten, NULL);
        // Recebe acknowledge da transação
        ReadFile(hSCReadPipe, &DumpAck, sizeof(DumpAck), &dwBytesRead, NULL);
        if (DumpAck.Erro > 0) {
                if (DumpAck.Erro == ERRO_ARQUIVO_INEXISTENTE)
                        printf("\nArquivo Inexistente\n");
                else printf("\nErro na operacao com o arquivo\n");
                Sleep(3000);
                break;
        }
        printf("Client: Bytes Read = %d EOF=%d \n", DumpAck.BytesRead, DumpAck.EndOfFile);

        do
           iTecla = _getch();
        while ((iTecla != ESC) && (iTecla != '+') && (iTecla != '-'));
        if ((iTecla == '+') && (!DumpAck.EndOfFile)) { lFilePosLow += dwRegLength; iBloco++; }
        if ((iTecla == '-') && (iBloco >0)) { lFilePosLow -= dwRegLength; iBloco--; }
        if (iTecla == ESC) {
          DumpRequest.Length = 0;        // Flag para abortar servidor
          WriteFile(hCSWritePipe, &DumpRequest, sizeof(DumpRequest), &dwBytesWritten, NULL);
          break;
        } // if
} while (1);
```

33

Pipes Anônimos – Uso de Herança Cliente

```
    CloseHandle(hCSWritePipe);
    CloseHandle(hSCReadPipe);

    // Aposto que você se esqueceu destes handles!!!
    CloseHandle(NewProcess.hProcess);
    CloseHandle(NewProcess.hThread);

    cout << endl << "Aperte uma tecla...:";
    cout << endl << endl;
    _getch();

    ExitProcess(0);
    return EXIT_SUCCESS;
}   // main
```

34

Pipes Anônimos – Uso de Herança Servidor

```
//   Capítulo 6 - Exemplo 2b        - Pipes Não Nomeados - Com Herança
//    Programa Servidor
void ShowDump(char *, int);

int main(int argc, char *argv[])     // Receberá id do processo pai como parâmetro
{                                    // e handles para os dois pipes
     DWORD dwExitCode = 0;
     char FileName[50]= "";          // Nome do arquivo a ser lido
     HANDLE hFile= 0;                // Handle para arquivo a ser aberto
     DWORD dwBytesRead;
     DWORD dwBytesWritten;
     LONG lFilePosLow=0L;            // Offset para posicao corrente do arquivo
     BOOL bStatus;
     char MyBuffer[512];             // Buffer para leitura do arquivo
     FileReadDesc DumpRequest;       // Descritor do pedido de leitura
     DumpFileAck DumpAck;            // Resposta do servidor para cliente
     DWORD dwLength;

     // Handle do pipe no sentido Cliente Servidor
     HANDLE hCSReadPipe;                    // Handle para operações de leitura
     // Handle do pipe no sentido Sevidor Cliente
     HANDLE hSCWritePipe;                   // Handle para operações de escrita
     // Obtém handles a partir da linha de comando
     hCSReadPipe  = (HANDLE) atoi(argv[0]);
     hSCWritePipe = (HANDLE) atoi(argv[1]);
```

35

Pipes Anônimos – Uso de Herança Servidor

```
do {  // Recebe pedido de dump via pipe
     bStatus=ReadFile(hCSReadPipe, &DumpRequest, sizeof(DumpRequest), &dwBytesRead, NULL);
     CheckForError(bStatus);

     lFilePosLow = DumpRequest.InitialPosition;
     dwLength = DumpRequest.Length;
     if (dwLength==0) break;        // Flag para abortar processo servidor.
     if (strcmp(FileName, DumpRequest.FileName) != 0) {
            strcpy(FileName, DumpRequest.FileName);
            if (hFile != 0) CloseHandle(hFile);
            hFile= CreateFile( FileName,        // Abre novo arquivo
                        GENERIC_READ|GENERIC_WRITE,
                        FILE_SHARE_READ|FILE_SHARE_WRITE, // abre para leitura e escrita
                        NULL,                   // atributos de segurança
                        OPEN_EXISTING,          // abre se existir
                        FILE_ATTRIBUTE_NORMAL,
                        NULL);                  // Template para atributos e flags
            CheckForError(hFile != INVALID_HANDLE_VALUE);
            if (hFile == INVALID_HANDLE_VALUE) {
                        DumpAck.Erro = ERRO_ARQUIVO_INEXISTENTE;
                        // Envia Acknowledge para Cliente
                        WriteFile(hSCWritePipe, &DumpAck, sizeof(DumpAck), &dwBytesWritten, NULL);
                        break;
            } // if
     }
```

36

Pipes Anônimos – Uso de Herança Servidor

```
            // Le registro desejado
            lFilePosLow = SetFilePointer(hFile,lFilePosLow,NULL,FILE_BEGIN);
            bStatus = ReadFile(hFile, &MyBuffer, dwLength, &dwBytesRead, NULL);
            CheckForError(bStatus);
            DumpAck.EndOfFile = (dwBytesRead != dwLength)? TRUE: FALSE;

            ShowDump(MyBuffer, (int)dwBytesRead);

            // Envia status da leitura para processo cliente
            DumpAck.BytesRead = dwBytesRead;
            DumpAck.Erro = 0;

            // Envia Acknowledge para Cliente
            WriteFile(hSCWritePipe, &DumpAck, sizeof(DumpAck), &dwBytesWritten, NULL);

    } while (1);

    CloseHandle(hFile);
    CloseHandle(hSCWritePipe);
    CloseHandle(hCSReadPipe);
    return EXIT_SUCCESS;
}   // main2
```

37

CreateNamedPipe

```
HANDLE CreateNamedPipe(
LPCTSTR lpName,                          // Apontador para nome
DWORD dwOpenMode,                        // Modo de abertura do pipe
DWORD dwPipeMode,                        // Modos específicos.
DWORD nMaxInstances,                     // Número máximo de instâncias.
DWORD nOutBufferSize,                    // Tamanho do buffer de saída em bytes
DWORD nInBufferSize,                     // Tamanho do buffer de entrada em bytes
DWORD nDefaultTimeOut,                   // Timeout em milisegundos.
LPSECURITY_ATTRIBUTES                    // Atributos de segurança.
lpSecurityAttributes
);
```

Retorno da função:

Status	Interpretação
Handle válido	Sucesso
INVALID_HANDLE_VALUE	Falha

38

Asynchronous Procedure Call

lpName	Nome do pipe segundo o seguinte formato: \\.**pipe****pipename** O ponto indica que o pipe será criado no computador local. **Exemplo:** char lpName[]= **"\\\\.\\pipe\\mypipe";**
dwOpenMode	**Direção do fluxo de dados (obrigatórios):** Este parâmetro deverá ser o mesmo para qualquer instância do pipe. PIPE_ACCESS_DUPLEX: pipe bidirecional PIPE_ACCESS_INBOUND: cliente para servidor PIPE_ACCESS_OUTBOUND: servidor para cliente **Flags (opcionais):** Podem ser diferentes em cada instância: FILE_FLAG_WRITE_THROUGH – escrita no pipe só retorna após dado ser copiado no buffer do computador remoto. FILE_FLAG_OVERLAPPED – habilita comunicação assíncrona.
dwPipeMode	**Modos de escrita:** (deve ser igual para todas as instâncias) PIPE_TYPE_BYTE – dados serão escritos no pipe como uma seqüência de bytes. PIPE_TYPE_MESSAGE – dados são escritos no buffer como uma seqüência de mensagens. **Modos de Leitura:** (podem ser diferentes para diferentes instâncias) PIPE_READMODE_BYTE – dados são lidos do pipe como uma seqüência de bytes. PIPE_READMODE_MESSAGE – dados são lidos do buffer como uma seqüência de mensagens. Se o modo de escrita for MENSAGEM, o método de leitura pode ser BYTE ou MENSAGEM. Se o modo de escrita for BYTE, o método de leitura só poderá ser BYTE. **Modos de espera:** (podem ser diferentes para diferentes instâncias) PIPE_WAIT – Operações de *WriteFile*, *ReadFile* e *ConnectNamedPipe* só retornam quando todos os dados foram escritos, há dados a serem lidos, ou um cliente está conectado (operação síncrona). PIPE_NOWAIT – Modo não bloqueante. *ReadFile*, *WriteFile* ou *ConnectNamedPipe* sempre retornam imediatamente. Mantém compatibilidade com LAN Manager 2.0. **NÃO deve ser usado.** Quando em modo de mensagem, um cabeçalho é introduzido transparentemente a cada mensagem enviada através do comando WriteFile, prefaceando o conteúdo da mensagem com o seu com o seu tamanho. Isto facilita a recepção, que será interrompida arutomaticamente quando o final da mensagem for atingido.

Pipes nomeados

nMaxInstances	Valor de 1 a PIPE_UNLIMITED_INSTANCES
nOutBufferSize	Tamanho de referência apenas. O sistema operacional aloca o tamanho necessário.
nInBufferSize	Tamanho de referência apenas. O sistema operacional aloca o tamanho necessário.
nDefaultTimeout	Timeout em milisegundos para o cliente se conectar a uma instância de um pipe nomeado.
lpSecutityAttributes	Apontador para descritor de segurança para o handle do novo pipe (contém a ACL do novo handle). Se NULL será utilizada estrutura default e o handle para o pipe não será herdável

Servidor: ConnectNamedPipe

- **O servidor chamará a função *ConnectNamedPipe()* para se conectar a um cliente**
- **Se a função é chamada antes que um cliente tenha tentado se conectar ao pipe, o servidor ficará bloqueado e a função retornará TRUE, assim que o cliente se conectar. A partir deste ponto a comunicação pode se iniciar**

```
BOOL ConnectNamedPipe(
HANDLE hNamedPipe,                  // Handle para pipe alvo
LPOVERLAPPED lpOverlapped           // Apontador para estrutura OVERLAPPED
                                       NULL: operação síncrona.
);
```

Retorno da função:

Status	Interpretação
TRUE	Cliente ainda não havia se conectado e se conectou com Sucesso.
FALSE	Analise GetLastError(): ERROR_NO_DATA: processo cliente fechou o seu handle. ERROR_PIPE_CONNECTED: Cliente já havia se conectado ao pipe. Conexão completada.

41

Servidor: DisconnectNamedPipe / FlushFileBuffers

```
BOOL DisconnectNamedPipe(
HANDLE hNamedPipe,                  // Handle para pipe alvo
);
```

- **Esta função só poder usada pelo servidor. Ela tornará o handle do cliente inválido e fará com que todos os dados não lidos sejam perdidos**
- **Para garantir que os dados sejam lidos, o servidor deverá chamar *FlushFileBuffers()* antes de *DisconnectNamedPipe()***

```
BOOL FlushFileBuffers(
HANDLE hFile,                       // Handle da instância
);
```

42

Cliente: WaitNamedPipe

- **A função *WaitNamedPipe()* é usada por processos clientes para esperar que uma instância de um pipe nomeado esteja disponível para conexão. Assim que o pipe esteja disponível, o cliente deve utilizar a função *CreateFile()* para se conectar.**

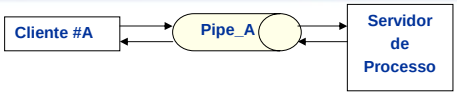
```
BOOL WaitNamedPipe(
LPCTSTR lpPipeName,        // Nome do pipe nomeado
DWORD dwTimeout            // Timeout em ms ou:
                           NMPWAIT_USE_DEFAULT_WAIT: o timeout usado pelo servidor na função
                           CreateNamedPipe será usado.
                           NMPWAIT_WAIT_FOREVER: a função só irá retornar quando uma instância
                           estiver disponível.
);
```

Retorno da função:

Status	Interpretação
TRUE	Sucesso
FALSE	Falha.

43

Uso de Pipes nomeados



Cliente	
Espera que uma instância do pipe esteja disponível [2]	*WaitNamedPipe()*
Se conecta [2]	*CreateFile()*
Escreve no Pipe [1 2]	*WriteFile()*
Recebe Reply [1 2]	*ReadFile()*
Fecha handle para pipe [2]	*CloseHandle()*
[1] Operações podem ser substituídas por *TransactNamedFile*	
[2] Conjunto de Operações pode ser substituído por *CallNamedPipe*	

Servidor	
Cria Pipe	*CreateNamed Pipe()*
Espera cliente ler todos os dados do Pipe [1]	*FlushFileBuffers()*
Quebra quaisquer conexões ativas e torna handle do cliente inválido [1]	*DisconnectNamedPipe()*
Espera pela conexão de um cliente	*ConnectNamedPipe()*
[1] Quando a instância do pipe já é existente.	

44

TransactNamedPipe

Realiza uma operação de escrita e espera um reply

```
BOOL TransactNamedPipe(
HANDLE hNamedPipe,                  // Handle do pipe
LPVOID lpInBuffer,                  // Buffer que contém os dados que serão escritos
DWORD nInBufferSize                 // Tamanho do buffer de escrita
LPVOID lpOutBuffer,                 // Buffer que receberá os bytes lidos
DWORD nOutBufferSize,               // Tamanho do buffer de leitura em bytes
LPDWORD lpBytesRead,                // Número de bytes lidos
LPOVERLAPPED lpOverlapped           // Define operação assíncrona
);
```

Retorno da função:

Status	Interpretação
TRUE	Sucesso
FALSE	Falha. GetLastError(): ERROR_MORE_DATE: tamanho do buffer de reply (*lpOutBuffer*) é insuficiente. Complemente a leitura com *ReadFile()*.

45

Call Named Pipe

Permite se conectar a um pipe, enviar uma mensagem, receber um reply e se desconectar em uma única operação

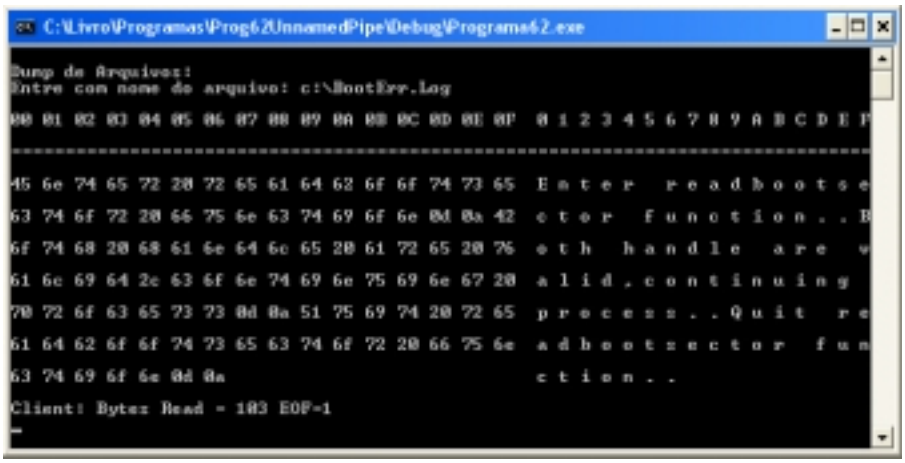
```
BOOL CallNamedPipe(
LPCTSTR lpNamedPipeName,    // Apontador para nome do pipe
LPVOID lpInBuffer,          // Apontador para buffer de escrita
DWORD nInBufferSize         // Tamanho do buffer de escrita
LPVOID lpOutBuffer,         // Apontador para buffer de leitura
DWORD nOutBufferSize,       // Tamanho do buffer de leitura em bytes
LPDWORD lpBytesRead,        // Número de bytes lidos
DWORD nTimeOut);            // Timeout em ms ou:
                            NMPWAIT_NOWAIT: retorna imediatamente
                            NMPWAIT_USE_DEFAULT_WAIT: usa timeout definido pelo servidor na função CreateNamedPipe
                            NMPWAIT_WAIT_FOREVER: a função só irá retornar, quando uma instância estiver disponível
```

Retorno da função:

Status	Interpretação
TRUE	Sucesso
FALSE	Falha. *GetLastError():* ERROR_MORE_DATE: tamanho do buffer de reply (*lpOutBuffer*) é insuficiente. Complemente a leitura com *ReadFile()*.

46

Exemplo: Dump de Arquivos



Pipes nomeados em operação síncrona Cliente

```
#define     ESC         0x1B

int main()
{
    char FileName[50];
    int iBloco;                             // Bloco de memória de 512 caracteres
    DWORD dwBytesRead;
    DWORD dwBytesWritten;
    DWORD dwRegLength;                      // Tamanho do registro a ser lido em bytes
    LONG lFilePosLow=0L;                    // Offset dentro do arquivo
    BOOL Status;
    PROCESS_INFORMATION NewProcess;         // Info sobre novo proc. criado
    int iTecla;
    STARTUPINFO si;                         // StartUpInformation para novo processo
    FileReadDesc DumpRequest;
    DumpFileAck DumpAck;
    BOOL bResult;
    HANDLE hPipe;
    LPTSTR lpszPipename = "\\\\.\\pipe\\Pipe63";
    int iProc;                              // Número de processadores
    char strVer[30];                        // Versão do sistema operacional
    BOOL bRet;
```

Cliente

```
        bRet = GetOSVersion(strVer, &iProc);
        printf("Versao do SO = %s  Processadores = %d\n", strVer, iProc);
        if (!bRet) {    // Não é NT. Não é NT ...
        printf("Falha: Sistema operacional NAO E O NT\n");
        printf("\nAcione uma tecla para terminar\n");
        _getch();     // Pare aqui, caso não esteja executando no ambiente MDS
        return 0;
} // if
// Cria processo servidor. Poderia ter sido criado a partir da linha de comando
ZeroMemory(&si, sizeof(si));
si.cb = sizeof(si);   // Tamanho da estrutura em bytes
Status = CreateProcess(
        "c:\\Livro\\Programas\\Programa63Server\\Debug\\Programa63Server.exe
        NULL,                   // linha de comando: Passa id do processo como parâmetro
        NULL,                   // atributos de segurança: Processo
        NULL,                   // atributos de segurança: Thread
        FALSE,                  // herança de handles
        NORMAL_PRIORITY_CLASS,          // CreationFlags
        NULL,                   // lpEnvironment
        "c:\\Livro\\Programas\\Programa63Server\\Debug",         // diretório corrente do filho
        &si,                    // lpStartUpInfo
        &NewProcess);           // lpProcessInformation
CheckForError(Status);
```

49

Cliente

```
while (1) {          // Espera conexão
        hPipe = CreateFile(
                lpszPipename,           // nome do pipe
                GENERIC_READ | GENERIC_WRITE,           // acesso para leitura e escrita
                0,                      // sem compartilhamento
                NULL,                   // lpSecurityAttributes
                OPEN_EXISTING,          // dwCreationDistribution
                0,                      // dwFlagsAndAttributes
                NULL);                  // hTemplate
        if (hPipe != INVALID_HANDLE_VALUE) break; // Se o handle é válido pode usar o pipe
        // Todas as instâncias estão ocupadas, então espere pelo tempo default
        if (WaitNamedPipe(lpszPipename, NMPWAIT_USE_DEFAULT_WAIT) == 0)
                printf("\nEsperando por uma instancia do pipe...");
        // Temporização abortada: o pipe ainda não foi criado
} // while
do {    // Interage com o usuário
        printf("\nDump de Arquivos: ");
        printf("\nEntre com nome do arquivo: ");
        bResult = bGetString(FileName, 30);
        If (!bResult) {  // Acionou Escape
                DumpRequest.Length = 0;          // Flag para abortar servidor
                WriteFile(hPipe, &DumpRequest, sizeof(DumpRequest), &dwBytesWritten, NULL);
                printf("\n");
                break; // Termina o programa
        }
```

Aqui aborta o Servidor

50

Cliente

```
            iBloco = 0;
            dwRegLength = 128L;
            strcpy(DumpRequest.FileName, FileName);
            DumpRequest.Length = dwRegLength;
            do {  DumpRequest.InitialPosition = iBloco * dwRegLength;
                  WriteFile(hPipe, &DumpRequest, sizeof(DumpRequest),   // Envia dados dos pedidos
                  ReadFile(hPipe, &DumpAck, sizeof(DumpAck), &dwBytesRead, NULL); // Recebe Ack
                  if (DumpAck.Erro > 0) {
                        if (DumpAck.Erro == ERRO_ARQUIVO_INEXISTENTE) printf("\nArquivo Inexistente\n");
                        else printf("\nErro na operacao com o arquivo\n");
                        break;  } // if
                  printf("Client: Bytes Read = %d EOF=%d \n", DumpAck.BytesRead, DumpAck.EndOfFile);
                  do  iTecla = _getch();
                  while ((iTecla != ESC) && (iTecla != '+') && (iTecla != '-'));
                  if ((iTecla == '+') && (!DumpAck.EndOfFile)) {lFilePosLow += dwRegLength; iBloco++; }
                  if ((iTecla == '-') && (iBloco >0)) { lFilePosLow -= dwRegLength; iBloco--; }
                  if (iTecla == ESC) break;  // Volta à linha de comando
               } while (1);
            } while (1);
            CloseHandle(hPipe);
            CloseHandle(NewProcess.hProcess);  // Aposto que você se esqueceu destes handles!!!
            CloseHandle(NewProcess.hThread);
            printf("\nAperte uma tecla...\n");   _getch();
            ExitProcess(0); return EXIT_SUCCESS;
}    // main
```

51

Servidor

```
void ShowDump(char *, int);

int main()   // Receberá id do processo pai e handles para os dois pipes como parâmetro
{
    DWORD ExitCode = 0;
    char FileName[50]= "";                  // Nome do arquivo a ser lido
    HANDLE hFile= 0;                        // Handle para arquivo a ser aberto
    DWORD BytesRead;
    DWORD BytesWritten;
    LONG FilePosLow=0L;                     // Offset para posicao corrente do arquivo
    BOOL Status;
    char MyBuffer[512];                     // Buffer para leitura do arquivo
    FileReadDesc DumpRequest;               // Descritor do pedido de leitura
    DumpFileAck DumpAck;                    // Resposta do servidor para cliente
    DWORD Length;
    DWORD nBytes=-1;                        // Número de bytes retornado na leitura do environment
    HANDLE hPipe;                           // Handle para o pipe nomeado
    DWORD ErrorCode;                        // Código de erro retornado por GetLastError()
```

52

Servidor

```
// Cria instância de um pipe: neste caso só uma: Só funciona no Windows NT
hPipe = CreateNamedPipe(
        "\\\\.\\pipe\\Pipe63",
        PIPE_ACCESS_DUPLEX,// Comunicação Full Duplex
        PIPE_TYPE_MESSAGE | PIPE_READMODE_MESSAGE | PIPE_WAIT,
        1,                      // Número de instâncias
        0,                      // nOutBufferSize
        0,                      // nInBufferSize
        INFINITE,               // Timeout para esperar por cliente
        NULL);                  // Atributos de segurança
CheckForError(hPipe != INVALID_HANDLE_VALUE);
Status = ConnectNamedPipe(hPipe, NULL);
if (Status) printf("Cliente se conectou com sucesso\n");
else {
        ErrorCode = GetLastError();
        if (ErrorCode == ERROR_PIPE_CONNECTED) printf("Cliente já havia se conectado\n");
        else if (ErrorCode == ERROR_NO_DATA) {
                printf("Cliente fechou seu handle\n");
                return 0;
            } // if
        else CheckForError(FALSE);
        }  // else
```

53

Servidor

```
do {    // Recebe pedido de dump via pipe
        Status=ReadFile(hPipe, &DumpRequest, sizeof(DumpRequest), &BytesRead, NULL);
        CheckForError(Status);
        FilePosLow = DumpRequest.InitialPosition;
        Length = DumpRequest.Length;
        if (Length==0) break;  // Flag para abortar processo servidor.
        if (strcmp(FileName, DumpRequest.FileName) != 0) {
            strcpy(FileName, DumpRequest.FileName);
            // Vai abrir novo arquivo
            if (hFile != 0) CloseHandle(hFile); // Fecha arquivo anterior
            hFile= CreateFile(
                    FileName,
                    GENERIC_READ|GENERIC_WRITE,
                    FILE_SHARE_READ|FILE_SHARE_WRITE,   // abre para leitura e escrita
                    NULL,                   // atributos de segurança
                    OPEN_EXISTING,          // abre se existir
                    FILE_ATTRIBUTE_NORMAL,
                    NULL);                  // Template para atributos e flags
          CheckForError(hFile != INVALID_HANDLE_VALUE);
          if (hFile == INVALID_HANDLE_VALUE) {
             DumpAck.Erro = ERRO_ARQUIVO_INEXISTENTE;
             WriteFile(hPipe, &DumpAck, sizeof(DumpAck), &BytesWritten, NULL); // Envia Ack para Cliente
             break;
           } // if
        } // if
```

Neste ponto eu aborto o servidor

54

Servidor

```
            // Le registro desejado
            FilePosLow = SetFilePointer(hFile,FilePosLow,NULL,FILE_BEGIN);
            Status = ReadFile(hFile, &MyBuffer, Length, &BytesRead, NULL);
            CheckForError(Status);
            DumpAck.EndOfFile = (BytesRead != Length)? TRUE: FALSE;
            ShowDump(MyBuffer, (int)BytesRead);
            // Envia status da leitura para processo cliente
            DumpAck.BytesRead = BytesRead;
            DumpAck.Erro = 0;
            // Envia Acknowledge para Cliente
            WriteFile(hPipe, &DumpAck, sizeof(DumpAck), &BytesWritten, NULL);
    } while (1);
    CloseHandle(hFile);
    CloseHandle(hPipe);
    return EXIT_SUCCESS;
}   // main
```

55

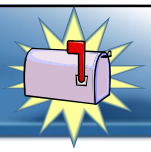
Operações Assíncronas com Pipes

- **A operação assíncrona com pipes é semelhante a operação assíncrona com I/O. O handle para o pipe deve ser obtido usando *CreateNamedPipe()* e *CreateFile()* com a flag FILE_FLAG_OVERLAPPED ativada.**
- **As funções *ReadFile()*, *WriteFile()*, *TransactNamedPipe()* e *ConnectNamedPipe()* deverão ter o parâmetro lpOverlapped definido com um apontador para a estrutura OVERLAPPED.**
- **O problema com operações assíncronas é verificar se a operação foi ou não completada quando a função retornou. A análise é a seguinte:**
  - **Se a função retornar TRUE isto significa que a operação foi concluída com sucesso antes do retorno da função**
  - **Se a função retornar FALSE devemos examinar o retorno de *GetLastError()*:**
  - **Se retorno for ERROR_IO_PENDING então a leitura foi enfileirada normalmente**
  - **Caso contrário a operação fracassou**
- **A estrutura OVERLAPPED deve ser carregada com a identificação do evento a ser causado.**

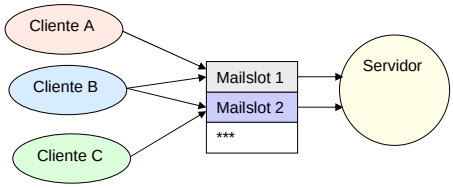
```
typedef struct _OVERLAPPED {
        DWORD  Internal;
        DWORD  InternalHigh;
        DWORD  Offset;
        DWORD  OffsetHigh;
        HANDLE hEvent;
  } OVERLAPPED;
```

56

MAISLOTS = CAIXA POSTAL

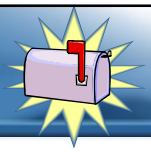


- Mecanismo usado para implementar uma comunicação unidirecional de um processo servidor com os processos clientes
- O servidor cria uma caixa postal e fica a espera de mensagens
- Os clientes inserem mensagens na caixa postal do servidor
- As mensagens são armazenadas na sua ordem de chegada. Os clientes localizam o servidor através do nome da caixa postal
- Mailsots são indicados para aplicações onde uma confirmação de recebimento de mensagem não é necessária, como por exemplo para se criar um gerenciador de alarmes. Todas as tarefas que quiserem inserir uma mensagem de alarme escrevem na caixa postal. Também como uma caixa postal, o mailslot pode deixar de existir quando todos os handles para ele foram apagados. Todos os dados existentes no mailslot são perdidos nesta situação



57

Mailslots



Características gerais:

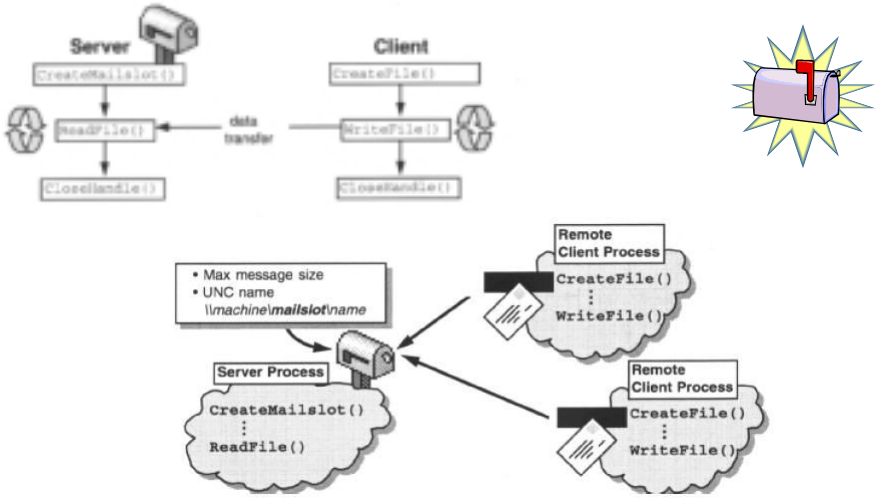
- **A comunicação é unidirecional.**
- **Possui menos segurança do que pipes já que nenhuma conexão é necessária Quando o mailslot é fechado, todos os dados são perdidos**
- **Mailslots operam através da rede**
- **Mailslots funcionam tanto no Windows NT como no Windows 95/98**

Resumo da operação:

- **Servidor cria o mailslot através de *CreateMailslot()* e obtém um handle. O mailslot possui um nome**
- **Servidor fica a espera de mensagens, usando a diretiva *ReadFile()***
- **Cliente abre mailslot usando *CreateFile()***
- **Cliente escreve mensagem no mailslot através de *WriteFile()***

58

Realizando operações de I/O sem notificação



Fonte: Johanes Plachy – Win32 Programming

59

CreateMailslot

```
HANDLE CreateMailslot (
LPCTSTR lpName,                  // Apontador para nome do mailslot no formato[1]:
                                 "\\\\.mailslot\\path\\nome"
DWORD nMaxMessageSize,           // Tamanho máximo da mensagem.
                                 0: Mensagem pode ser de qualquer tamanho
DWORD dwReadTimeout,             // Tempo máximo de espera por mensagem
                                 MAILSLOT_WAIT_FOREVER
LPSECURITY_ATTRIBUTES Ipsa       // Apontador para atributos de segurança
);
```

"\\\\.\\mailslot\\path\\nome"	Mailslot local com o nome especificado
"\\\\nome-do_computador\\mailslot\\path\\nome"	Mailslot remoto com o nome especificado
"\\\\nome_do_dominio\\mailsot\\path\\nome"	Todos os mailslots no domínio especificado que tenha o nome especificado
"*\\mailslot\\path\\nome"	Todos os mailslots com o nome especificado no domínio primário

Retorno:

Status	Interpretação
Handle válido	Sucesso
INVALID_HANDLE_VALUE	Falha

60

GetMailslotInfo

```
BOOL GetMailslotInfo(
HANDLE hMailslot,                 //  Handle para o mailslot
LPDWORD IpMaxMessageSize,         //  Tamanho máximo da mensagem. Pode ser NULL.
LPDWORD IpNextMessage,            //  Tamanho da próxima mensagem
                                  //  MAILSLOT_NO_MESSAGE: não há mensagens no buffer
LPDWORD IpMessageCount,           //  Número de mensagens existentes
LPDWORD IpReadTimeout             //  Valor do timeout. Este parâmetro pode ser NULL.
);
```

Retorno da função:

Status	Interpretação
Handle válido	Sucesso
INVALID_HANDLE_VALUE	Falha

61

Leitura de Mailslot

- A função de leitura de um Mailslot será sempre síncrona:

```
BOOL ReadFile(hMailslot, lpBuffer, BytesToRead, &BytesLidos, NULL);
```

62

O lado do Cliente

O cliente obtém um handle para o Mailslot:

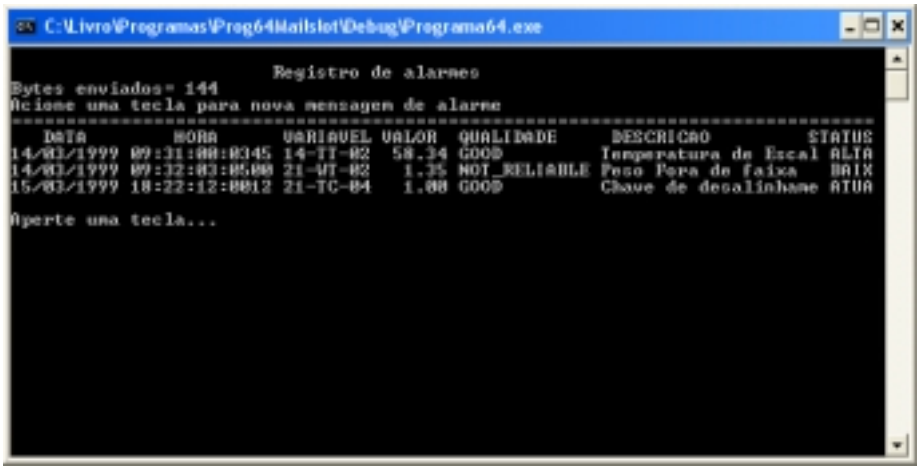
```
hMailslot = CreateFile(
                    "\\\\.\\mailslot\\path\\filename",
                    GENERIC_WRITE,
                    FILE_SHARE_READ,
                    lpSecurityAttribute,
                    OPEN_EXISTING,
                    FILE_ATTRIBUTE_NORMAL,
                    NULL);
```

- Para escrever no mailslot o cliente usa a função *WriterFile():*
  BOOL WriteFile(hMailslot, &Buffer, sizeof(Buffer), &BytesEscritos, NULL);

- O parâmetro lpOverlapped deverá ser sempre NULL.

63

Mailslot – Programa de teste



64

Exemplo – Programa cliente

```
#define WNT_LEAN_AND_MEAN
#include <windows.h>
#include <stdio.h>
#include <stdlib.h>
#include <conio.h>                      // _getch()
#include "Exemplo64.h"
#define  _CHECKERROR        1           // Ativa função CheckForError
#include "CheckForError.h"
#define  ESC                0x1B

AlarmMsg FormatMsg(DWORD, DWORD, DWORD, DWORD, DWORD, DWORD, DWORD, Quality,
double, char *, char *, char *);

int main()
{       int iTecla;
        HANDLE hMailslot;
        HANDLE hEvent;
        AlarmMsg Msg;
        BOOL bStatus;
        DWORD dwBytesEnviados;
        STARTUPINFO si;                         // StartUpInformation para novo processo
        PROCESS_INFORMATION NewProcess;         // Info sobre novo proc. Criado

        // Cria um objeto do tipo evento para sincronizar dois processos
        // O evento será sinalizado quando o servidor tiver criado o mailslot
        hEvent = CreateEvent(NULL, TRUE, FALSE, "Prog64");
```

Exemplo – Programa cliente

```
        // Cria processo servidor
        ZeroMemory(&si, sizeof(si));
        si.cb = sizeof(si);                     // Tamanho da estrutura em bytes
        bStatus = CreateProcess(
                "c:\\Livro\\Programas\\Prog64Server\\Debug\\Programa64Server.exe
                NULL,       // linha de comando
                NULL,       // atributos de segurança: Processo
                NULL,       // atributos de segurança: Thread
                FALSE,      // herança de handles
                NORMAL_PRIORITY_CLASS,          // CreationFlags
                NULL,       // lpEnvironment
                "c:\\Livro\\Programas\\Prog64Server\\Debug",     // dir corrente do filho
                &si,                    // lpStartUpInfo
                &NewProcess);           // lpProcessInformation
        CheckForError(bStatus);
        // Espera sincronismo para servidor e mailslot serem criados
        WaitForSingleObject(hEvent, INFINITE);
        hMailslot = CreateFile("\\\\.\\mailslot\\Livro\\Programas\\Programa64\\MyMailslot",
                        GENERIC_WRITE,
                        FILE_SHARE_READ,
                        NULL,
                        OPEN_EXISTING,
                        FILE_ATTRIBUTE_NORMAL,
                        NULL);
        CheckForError(hMailslot != INVALID_HANDLE_VALUE);
```

66

Exemplo – Programa cliente

```
        CheckForError(hMailslot != INVALID_HANDLE_VALUE);
        Msg = FormatMsg(14, 03, 1999, 9, 31, 0, 345, GOOD, 58.345, "14-TT-02", "Temperatura de
        Escaldagem", "ALTA");
        bStatus = WriteFile(hMailslot, &Msg, sizeof(AlarmMsg), &dwBytesEnviados, NULL);
        printf("Bytes enviados= %d\n", dwBytesEnviados);
        printf("Acione uma tecla para nova mensagem de alarme\n");
        iTecla = _getch();
        Msg = FormatMsg(14, 03, 1999, 9, 32, 3, 500, NOT_RELIABLE, 1.345, "21-WT-02", "Peso Fora de
        faixa", "BAIXO");
        bStatus = WriteFile(hMailslot, &Msg, sizeof(AlarmMsg), &dwBytesEnviados, NULL);
        iTecla = _getch();
        Msg = FormatMsg(15, 03, 1999, 18, 22, 12, 12, GOOD, 1, "21-TC-04DEF", "Chave de
        desalinhamento", "ATUADO");
        bStatus = WriteFile(hMailslot, &Msg, sizeof(AlarmMsg), &dwBytesEnviados, NULL);
        iTecla = _getch();
        // Envia Mensagem para abortar o servidor
        Msg = FormatMsg(0, 0, 0, 0, 0, 0, 0, ABORT, 0., "", "", "");
        bStatus = WriteFile(hMailslot, &Msg, sizeof(AlarmMsg), &dwBytesEnviados, NULL);
        CloseHandle(hMailslot);
        CloseHandle(hEvent);
        CloseHandle(NewProcess.hProcess);
        CloseHandle(NewProcess.hThread);
        printf("\nAperte uma tecla...\n");
        _getch();
```

Aborta o servidor aqui

67

Exemplo – Programa cliente

```
        ExitProcess(0);
        Return EXIT_SUCCESS;
}       // main

AlarmMsg FormatMsg(DWORD dia, DWORD mes, DWORD ano, DWORD hora, DWORD minuto, DWORD
segundo, DWORD ms, Quality Q, double value, char *tag, char *descr, char *status)
{
        AlarmMsg Msg;

        Msg.TimeStp.Date.Dia= dia;
        Msg.TimeStp.Date.Mes= mes;
        Msg.TimeStp.Date.Ano= ano;
        Msg.TimeStp.Time.Hora= hora;
        Msg.TimeStp.Time.Min= minuto;
        Msg.TimeStp.Time.Seg= segundo;
        Msg.TimeStp.Time.Ms= ms;
        Msg.Q= Q;
        Msg.Value= value;
        strcpy((char *) &Msg.Tag, tag);
        strcpy((char *) &Msg.Description, descr);
        strcpy((char *) &Msg.Status, status);
        return Msg;
} // FormatMsg
```

68

Exemplo – Programa Servidor

```
#define WIN32_LEAN_AND_MEAN
#include <windows.h>
#include <stdio.h>
#include <stdlib.h>
#include "Exemplo64.h"
#define _CHECKERROR1              // Ativa função CheckForError
#include "CheckForError.h"
// As estrutura definidas são ineficientes mas são mais fáceis de se decodificar
void ShowAlarm(AlarmMsg, DWORD);

int main()
{
    DWORD dwAlarmCont = 0;      // contador de mensagem de alarmes
    AlarmMsg MsgBuffer;
    HANDLE hMailslot;
    BOOL bStatus;
    DWORD dwBytesLidos;
    HANDLE hEvent;
    hEvent = OpenEvent(EVENT_MODIFY_STATE, FALSE, "Prog64");
    hMailslot= CreateMailslot(
            "\\\\.\\mailslot\\Livro\\Programas\\Programa64\\MyMailslot",
            0,
            MAILSLOT_WAIT_FOREVER,
            NULL);
    CheckForError(hMailslot != INVALID_HANDLE_VALUE);
```

9

Exemplo – Programa Servidor

```
    // Avisa que está apto a receber mensagens
    SetEvent(hEvent);
    if (hMailslot != INVALID_HANDLE_VALUE)
    do {
            #ifdef TEST_GET_MAILSLOT_INFO
            DWORD MaxMsgSize;
            DWORD NextMsgSize;
            DWORD MsgCont;
            DWORD Timeout;
            bStatus=GetMailslotInfo(hMailslot, &MaxMsgSize, &NextMsgSize, &MsgCont,&Timeout);
            CheckForError(bStatus);
            printf("NextMsgSize= %d\n", MaxMsgSize);
            #endif
            bStatus = ReadFile(hMailslot, &MsgBuffer, sizeof(AlarmMsg), &dwBytesLidos, NULL);
            CheckForError(bStatus);
            if (MsgBuffer.Q == ABORT) break; // aborta servidor
            ShowAlarm(MsgBuffer, dwAlarmCont++);
    } while (1);

    CloseHandle(hEvent);
    CloseHandle(hMailslot);
    return EXIT_SUCCESS;
}   // main
```

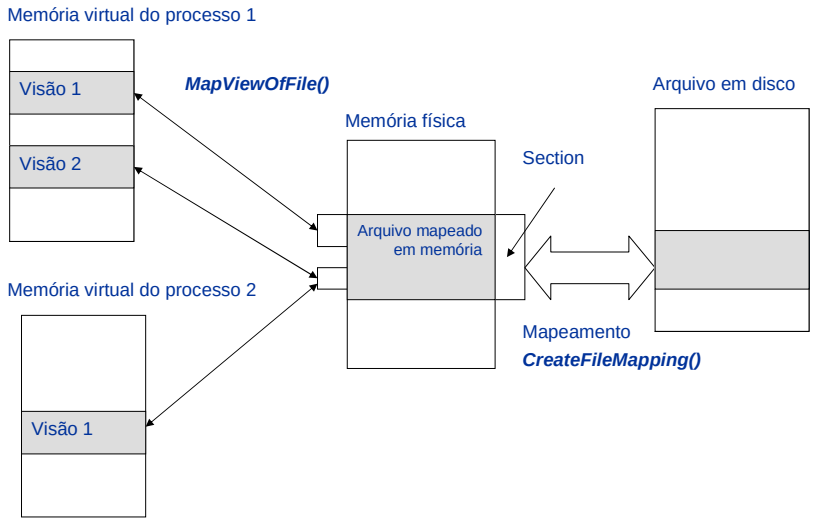
70

Exemplo – Programa Servidor

```
void ShowAlarm(AlarmMsg Msg, DWORD dwAlarmCont)
{
    char *QualityMsg[]= {"INVALID","GOOD","NOT_RELIABLE","FORA DE FAIXA"};
    char Quality[15];
    if ((dwAlarmCont % 16) ==0) {  // Imprime cabeçalho
            printf("\n                              Registro de alarmes\n");
            printf("=============================================================
            ======================\n");
            printf("  DATA       HORA      VARIAVEL VALOR  QUALIDADE     DESCRICAO        STATUS\n");
    //     13/03/1999 14:02:54:0673 21-WT-45 1245.93 GOOD  Temperatura do Forno      ALTA
    }  // if

    if (Msg.Q < 3)
            strcpy(Quality, QualityMsg[Msg.Q]);
    else strcpy(Quality, QualityMsg[3]);
    printf("%02d/%02d/%4d %02d:%02d:%02d:%04d %8.8s %6.2f %-12.12s %-20.20s %-4.4s\n",
    Msg.TimeStp.Date.Dia, Msg.TimeStp.Date.Mes, Msg.TimeStp.Date.Ano,
    Msg.TimeStp.Time.Hora, Msg.TimeStp.Time.Min, Msg.TimeStp.Time.Seg,
    Msg.TimeStp.Time.Ms,
    Msg.Tag,
    Msg.Value,
    Quality,
    Msg.Description,
    Msg.Status);
} // ShowAlarm
```

1

Memória Compartilhada



72

CreateFileMapping

```
HANDLE CreateFileMapping(
HANDLE hFile,                  // Handle do arquivo a ser mapeado
                               // 0xFFFFFFFF – paging file
LPSECURITY_ATTRIBUTES lpsa,    // Apontador para atributos de segurança
DWORD flProtect,               // Tipo de acesso
DWORD dwMaximumSizeHigh,       // Tamanho do objeto – palavra mais significativa.
DWORD dwMaximumSizeLow,        // Tamanho do objeto – palavra menos significativa.
                               // Se ambos forem 0 a section terá o mesmo tamanho do arquivo.
LPCTSTR lpName                 // Nome do objeto de arquivo mapeado.
);
```

Retorno da função:

Status	Interpretação
Handle válido	Sucesso. *GetLastError()* retorna ERROR_ALREADY_EXISTS se o objeto já existir e retorna um handle válido para o objeto já existente. O tamanho será o especificado na criação original.
NULL	Falha

73

CreateFileMapping

flProtect — Proteção desejada para a visão do arquivo quando o arquivo é mapeado:

PAGE_READONLY: Dá direito de leitura apenas. O arquivo deve ter sido criado com atributo GENERIC_READ.

PAGE_READWRITE: Dá direito de leitura e escrita. O arquivo deve ter sido criado com atributos GENERIC_READ e GENERIC_WRITE.

PAGE_WRITECOPY: Dá direito de cópia na escrita da section. O arquivo deve ter atributos de leitura e escrita.

SEC_NOCACHE: Todas as páginas da section serão não cacheáveis.

SEC_IMAGE: a seção do disco é um arquivo executável.

74

OpenFileMapping

```
HANDLE OpenFileMapping(
DWORD dwDesiredAccess,      // Tipo de acesso:
                            //   FILE_MAP_READ: leitura apenas
                            //   FILE_MAP_ALL_ACCESS: leitura e escrita
BOOL bInheritHandle,        // Indica se o handle será herdável
LPCTSTR lpName              // Apontador para o nome do objeto
);
```

Retorno da função:

Status	Interpretação
Handle válido	Sucesso.
NULL	Falha

75

MapViewOfFile

```
LPVOID MapViewOfFile(
HANDLE hFileMappingObject,  // Handle para objeto de arquivo mapeado
DWORD dwDesiredAccess,      // Modo de acesso
                            //   FILE_MAP_WRITE: escrita e leitura.
                            //   FILE_MAP_READ: apenas leitura.
DWORD dwOffsetHigh,         // Offset dentro da seção.
DWORD dwOffsetLow,          // Offset dentro da seção.
DWORD dwNumberOfBytesToMap  // Número de bytes a serem mapeados
                            //   0: Mapeia todo o arquivo.
);
```

Retorno da função:

Status	Interpretação
Valor de apontador válido	Sucesso.
NULL	Falha

76

UnmapViewOfFile / FlushViewOfFile

```
BOOL UnmapViewOfFile(
LPCVOID IpBaseAddress );   // Valor retornado pela função MapViewOfFile().
```

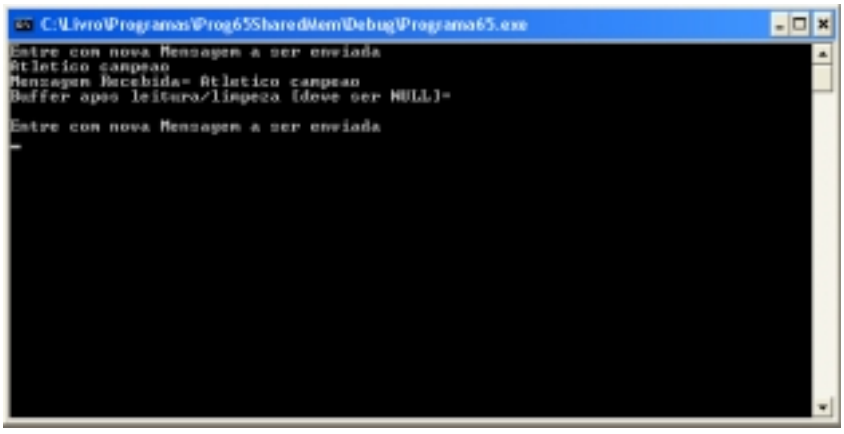
```
BOOL FlushViewOfFile(
LPCVOID IpBaseAddress,     // Valor retornado pela função MapViewOfFile().
DWORD dwNBytes);           // Número de bytes a serem escritos no disco.
                           // 0: Escreve todos os bytes.
```

Retorno:

Status	Interpretação
<> 0	Sucesso
0	Falha. Chame *GetLastError()* para obter mais informações.

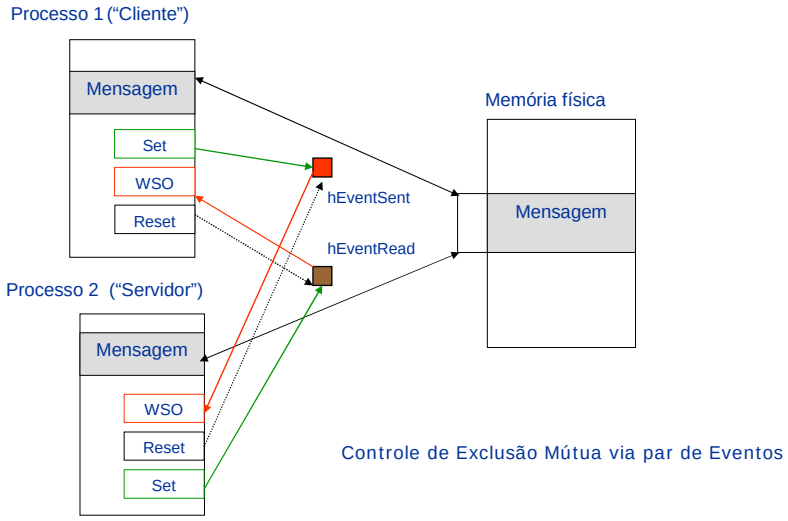
77

Memória Compartilhada - Exemplo



78

Memória Compartilhada - Exemplo



Controle de Exclusão Mútua via par de Eventos

79

Memória Compartilhada - Cliente

```
#define MSG_SIZE  128            // Tamanho máximo da mensagem a ser enviada
DWORD getstr(char *, DWORD);

int main()
{
    DWORD ExitCode = 0;
    BOOL bStatus;
    STARTUPINFO si;              // StartUpInformation para novo processo
    PROCESS_INFORMATION NewProcess;     // Informações sobre novo processo criado
    char Mensagem[MSG_SIZE]="";
    char *lpImage;               // Apontador para imagem local
    DWORD dwNumChar;
    HANDLE hSection;
    HANDLE hEventSent;
    HANDLE hEventRead;

    // Cria processo servidor
    ZeroMemory(&si, sizeof(si));
    si.cb = sizeof(si);  // Tamanho da estrutura em bytes
```

80

Memória Compartilhada - Cliente

```
bStatus = CreateProcess(
        "c:\\Livro\\Programas\\Programa65b\\Debug\\Programa65b.exe", // Nome
        NULL,       // linha de comando
        NULL,       // atributos de segurança: Processo
        NULL,       // atributos de segurança: Thread
        FALSE,      // herança de handles
        NORMAL_PRIORITY_CLASS,          // CreationFlags
        NULL,       // lpEnvironment
        "c:\\Livro\\Programas\\Programa65b\\Debug",   // diretório corrente do filho
        &si,        // lpStartUpInfo
        &NewProcess);           // lpProcessInformation
 CheckForError(bStatus);

 hSection= CreateFileMapping(
        (HANDLE)0xFFFFFFFF,
        NULL,
        PAGE_READWRITE,     // tipo de acesso
        0,                  // dwMaximumSizeHigh
        MSG_SIZE,           // dwMaximumSizeLow
        "MADONNA");         // Escolha o seu nome preferido
CheckForError(hSection);
```

81

Memória Compartilhada - Cliente

```
lpImage= (char *)MapViewOfFile(
        hSection,
        FILE_MAP_WRITE,     // Direitos de acesso: leitura e escrita
        0,                  // dwOffsetHigh
        0,                  // dwOffset Low
        MSG_SIZE);          // Número de bytes a serem mapeados
CheckForError(lpImage);
hEventSent= CreateEvent(NULL, FALSE, FALSE,"MsgAvailable");   // Cria evento com reset automático
CheckForError(hEventSent);
hEventRead= CreateEvent(NULL, FALSE, FALSE,"MsgRead");       // Cria evento com reset automático
CheckForError(hEventRead);
do {
        printf("Entre com nova Mensagem a ser enviada\n");
        dwNumChar = getstr(Mensagem, MSG_SIZE);
        printf("\n");
        // Escreve na memória compartilhada
        strcpy(lpImage, Mensagem);
        SetEvent(hEventSent); // Avisa processo B
        if (dwNumChar ==0) break;
        // Espera que processo B leia a mensagem
        WaitForSingleObject(hEventRead, INFINITE);
        ResetEvent(hEventRead);
        printf("Buffer= %s\n\n", lpImage);
} while(TRUE);
```

Neste ponto eu aborto o cliente

82

Memória Compartilhada - Cliente

```
    // Elimina mapeamento
    bStatus=UnmapViewOfFile(lpImage);
    CheckForError(bStatus);
    CloseHandle(hSection);
    CloseHandle(hEventSent);
    CloseHandle(hEventRead);
    CloseHandle(NewProcess.hProcess);
    CloseHandle(NewProcess.hThread);
    ExitProcess(0);
    return EXIT_SUCCESS;
}   // main
```

83

Memória Compartilhada - Servidor

```
#define MSG_SIZE     20
int main()
{
    BOOL bStatus;
    char *lpImage;                  // Apontador para imagem local
    HANDLE hSection;
    HANDLE hEventSent;
    HANDLE hEventRead;

    hSection= OpenFileMapping(
            FILE_MAP_ALL_ACCESS,
            FALSE,                  // Handle herdável
            "MADONNA");             // Escolha o seu nome preferido
    CheckForError(hSection);
    lpImage= (char *)MapViewOfFile(
            hSection,
            FILE_MAP_WRITE,         // Direitos de acesso: leitura e escrita
            0,                      // dwOffsetHigh
            0,                      // dwOffset Low
            MSG_SIZE);              // Número de bytes a serem mapeados
    CheckForError(lpImage);
    hEventSent= OpenEvent(EVENT_ALL_ACCESS, FALSE, "MsgAvailable");
    // Abre eventos criados por Processo65
```

84

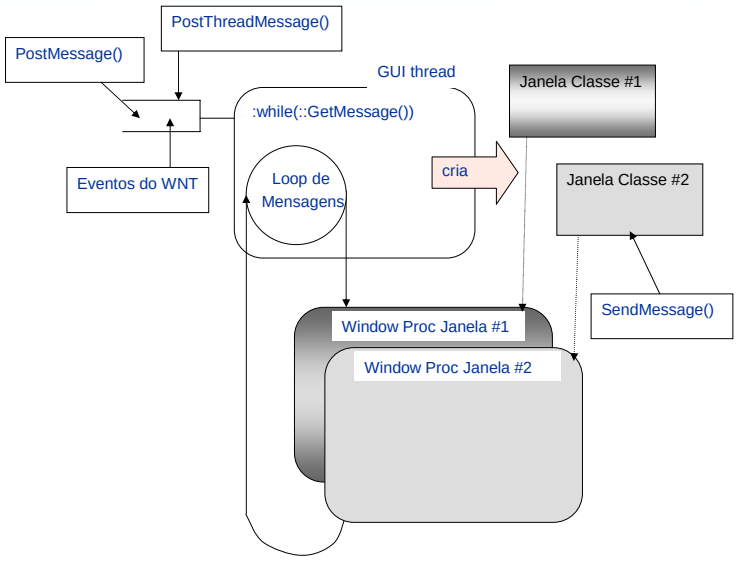
Memória Compartilhada - Servidor

```
    hEventRead= OpenEvent(EVENT_ALL_ACCESS, FALSE, "MsgRead"); // Cria evento com reset automático
    CheckForError(hEventRead);
    do {
            // Espera que processo A escreva mensagem
            WaitForSingleObject(hEventSent, INFINITE);
            ResetEvent(hEventSent);
            printf("Mensagem Recebida= %s\n", lpImage);
            if (strcmp(lpImage, "")==0) break;
            // Limpa memória compartilhada
            strcpy(lpImage, "");
            SetEvent(hEventRead);              // Avisa processo A

    } while(TRUE);
    // Elimina mapeamento
    bStatus=UnmapViewOfFile(lpImage);
    CheckForError(bStatus);
    CloseHandle(hSection);
    CloseHandle(hEventSent);
    CloseHandle(hEventRead);

    ExitProcess(0);
    return EXIT_SUCCESS;
}   // main
```

Neste ponto eu aborto o servidor

85

Mecanismo de Mensagens



86

Comunicação entre threads via fila de mensagens

O loop de espera de uma GUI thread é típico:

```
While (GetMessage(&msg, NULL, 0, 0)) {
    TranslateMesage(&msg);
    DispatchMessage(&msg);
}
```

A função *GetMessage()* retira, a cada loop, uma mensagem da fila e realiza o seu tratamento.
Cada mensagem segue o formato:

```
typedef struct tagMSG {     // msg
    HWND   hwnd;
    UINT   message;          // WM_LBUTTONDOWN | WM_PAINT ...
    WPARAM wParam;           // parâmetro de 32 bits que é função da mensagem
    LPARAM lParam;           // o mesmo
    DWORD  time;             // time stamp de quando a mensagem foi enfileirada
    POINT  pt;               // coordenadas do mouse:
} MSG;
```

pt é a coordenada da posição instantânea do mouse:

```
typedef struct POINT {
    LONG x;
    LONG y;
} POINT;
```

87

PostMessage

```
BOOL PostMessage(
HWND hWnd,          //  Handle da janela que receberá a mensagem.
UINT uMsg,          //  Mensagem a ser enviada
WPARAM wParam,      //  Primeiro parâmetro da mensagem
LPARAM lParam       //  Segundo parâmetro da mensagem
);
```

Comentários sobre os parâmetros:

hWnd — HWND_BROADCAST – mensagem de broadcast a ser enviada para todas as janelas do sistema, exceto janela filhas.
NULL – funciona como *ThreadPostMessage()*, com o primeiro parâmetro definido setado para o identificador da janela corrente.

uMsg — O valor de uMsg pode ser qualquer valor não utilizado pelo Windows, isto é, valores entre WM_USER e 0x7FFF ou entre WM_APP e 0xBFFF. A última faixa é a mais indicada.

Retorno da função:

Status	Interpretação
!= 0	Sucesso
0	Falha

88

PostMessage/SendMessage - Endereços

Faixa	Significado
0 a WM_USER-1	Mensagens reservadas para uso do Windows
WM_USER a 0x7FFFF	Mensagens que podem ser utilizada por classes privadas do Windows. Classes tais como BUTTON, LIST, LISTBOX e COMBOBOX utilizam mensagens nesta faixa.
WM_APP a 0xBFFF	Mensagens reservadas para uso futuro do Windows
0xC000 a 0xBFFF	Mensagens de strings para uso de aplicações
>0xFFFF	Reservado pelo Windows para uso futuro

89

SendMessage

```
BOOL SendMessage(
HWND hWnd,          // Handle da janela que receberá a mensagem.
UINT iMsg,          // Mensagem a ser enviada
WPARAM wParam,      // Primeiro parâmetro da mensagem
LPARAM lParam       // Segundo parâmetro da mensagem
);
```

Retorno da função:

Status	Interpretação
!= 0	Sucesso
0	Falha

90

PostThreadMessage

```
BOOL PostThreadMessage(
DWORD idThread,        //  Identificador da Thread
UINT Msg,              //  Tipo de mensagem a ser enviada: WM_XXX
WPARAM wParam,         //  Primeiro parâmetro da mensagem
LPARAM lParam          //  Segundo parâmetro da mensagem
);
```

Retorno da função:

Status	Interpretação
!=0	Sucesso
0	Falha *GetLastError():* INVALID_THREAD_ID: thread não existe ou não possui fila de mensagens.

91

PeekMessage

- **A função *PeekMessage()* pode ser usada por uma thread para forçar a criação da fila de mensagens, caso ela seja uma thread de trabalho e não uma GUI thread**

```
BOOL PeekMessage(
LPMSG lpMsg,           //  Endereço da estrutura para receber a mensagem
HWND hWnd,             //  Handle para a janela
UINT wMsgFiletrMin,    //  Primeira mensagem, por ex. WM_MOUSEFIRST
UINT wMsgFilterMax,    //  Última mensagem, por ex WM_MOUSELAST
UINT wRemoveMsg        //  PM_NOREMOVE – Não remove mensagem após leitura
                       //  PM_REMOVE – Remove mensagem após processamento
);
```

Retorno da função:

Status	Interpretação
!=0	Mensagem disponível
0	Mensagem não disponível

92

GetMessage

```
BOOL GetMessage(
LPMSG lpMsg,          // Endereço da estrutura para receber a mensagem
HWND hWnd,            // Handle para a janela
UINT wMsgFiletrMin,   // Primeira mensagem, por ex. WM_MOUSEFIRST, WM_KEYFIRST
UINT wMsgFilterMax    // Última mensagem, por ex WM_MOUSELAST, WM_KEYLAST
);
```

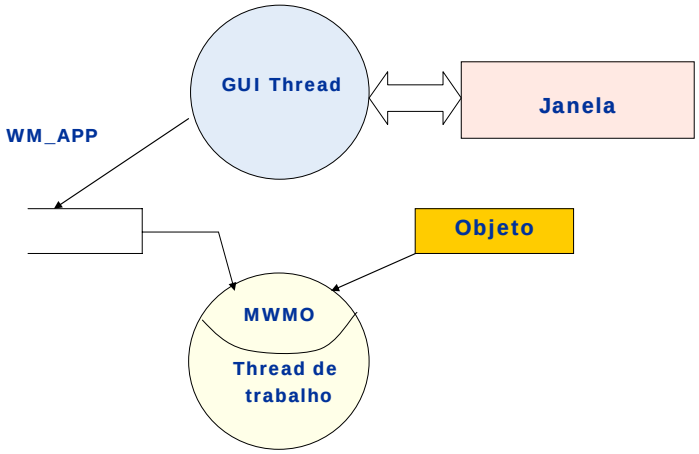
hWnd — NULL – *GetMessage* retira mensagens enviadas para qualquer janela pertencente à thread, e mensagens enviadas via função *PostThreadMessage()* da fila de mensagens.

Retorno da função:

Status	Interpretação
TRUE (>0)	Mensagem diferente de WM_QUIT
FALSE (0)	Mensagem recebida foi WM_QUIT
-1	ERRO

93

Esperando pela sinalização de objetos e por mensagens do Windows



94

MsgWaitForMultipleObjects

```
DWORD MsgWaitForMultipleObjects(
DWORD nCount,                 // Número de handles a esperar,  limitado por MAXIMUM_WAIT_OBJECTS
LPHANDLE pHandles,            // Vetor de handles para objetos do kernel
BOOL fWaitAll,                // TRUE: retorna se todos os handles forem sinalizados.
                              //   FALSE:  retorna se qualquer handle for sinalizado
DWORD dwMiliseconds,          // Tempo máximo que desejamos esperar
DWORD dwWakeMask              // Tipos de entradas do usuário a serem esperadas:
                              //    QS_ALLINPUT
                              //    QS_HOTKEY
                              //    QS_INPUT
                              //    QS_KEY
                              //    QS_MOUSE
                              //     QS_MOUSEBUTTON
                              //     QS_MOUSEMOVE
                              //     QS_PAINT
                              //     QS_POSTMESSAGE
                              //     QS_SENDMESSAGE
                              //     QS_TIMER
);
```

95

MsgWaitForMultipleObjects

Retorno da função:

bWaitAll	Status	Interpretação
TRUE FALSE	WAIT_OBJECT_0 WAIT_OBJECT_0 + Index	Todas threads retornaram Valor-WAIT_OBJECT_0 = índice da thread que retornou
	WAIT_ABANDONED_0 + x	Uma thread proprietária de um Mutex o abandona sem liberá-lo x é o índice da thread.
	WAIT_OBJECT_0 + nCount	Mensagem foi inserida na fila
	WAIT_TIMEOUT	Ocorreu *timeout*
	WAIT_FAILED	Função falhou

96

MsgWaitForMultipleObjects

```
#define WIN32_LEAN_AND_MEAN
#define _WIN32_WINNT  0x0400  // Ativa funções definidas após versão 4
#include <windows.h>
#include <process.h>              // _beginthreadex() e _endthreadex()
#include <stdio.h>
#include <stdlib.h>
#define _CHECKERROR1              // Ativa função CheckForError
#include "CheckForError.h"

DWORD WINAPI ThreadControlador(LPVOID);
BOOL bGetFloat(double *, int );
double Pid(double, double, double, double, double);

// Casting para terceiro e sexto parâmetros da função _beginthreadex
typedef unsigned (WINAPI *CAST_FUNCTION)(LPVOID);
typedef unsigned *CAST_LPDWORD;

#define WM_NOVO_SET_POINT     WM_APP + 0x0001
#define WM_NOVO_KP            WM_APP + 0x0002
#define WM_NOVO_TI            WM_APP + 0x0003
#define WM_NOVO_TD            WM_APP + 0x0004
HANDLE hEvent;
double dSetPoint;
```

97

MsgWaitForMultipleObjects

```
int main(VOID)
{
    HANDLE          hThread;
    DWORD           dwThreadId;
    DWORD           dwExitCode;
    BOOL            bStatus;
    // Cria um objeto do tipo evento para sincronizar duas threads
    // O evento será sinalizado quando a thrad estiver apta a receber mensagens
    hEvent = CreateEvent(NULL, TRUE, FALSE, NULL);
    hThread = (HANDLE) _beginthreadex(              // Cria thread controlador PID
          NULL,      0,
          (CAST_FUNCTION) ThreadControlador,  0,
          0,  (CAST_LPDWORD)&dwThreadId);
    CheckForError(hThread);
    if (hThread) printf("Thread criada Id= %0x \n", dwThreadId);
    WaitForSingleObject(hEvent, INFINITE);          // Espera sincronismo para iniciar comunicação

    for (int cont=0; cont<3 ;++cont) {
       do {
          printf("\nEntre com o novo valor do SetPoint: ");
          bStatus = bGetFloat(&dSetPoint, 8);       // le string ou ESC
          if (bStatus)  // Entrada válida
          PostThreadMessage(dwThreadId, WM_NOVO_SET_POINT, NULL, (LPARAM)&dSetPoint);
       } while (!bStatus);
    }  // for
```

98

MsgWaitForMultipleObjects

```
    // Encerra thread controladora
    PostThreadMessage(dwThreadId, WM_QUIT, NULL, NULL);
    // Espera controladora terminar
    WaitForSingleObject(hThread, INFINITE);
    GetExitCodeThread(hThread, &dwExitCode);
    CloseHandle(hThread);
    CloseHandle(hEvent);
    return EXIT_SUCCESS;
}

// Thread que implementa controlador PID
DWORD WINAPI ThreadControlador(LPVOID n)
{
    UNREFERENCED_PARAMETER(n);
    MSG msg;
    DWORD dwRet;
    double dSetPoint, dKp, dTd, dTi, dPV;
    LARGE_INTEGER Preset;
    HANDLE hTimer;                          // Handle para Timer
    // Define uma constante para acelerar cálculo do atraso e período
    const int nMultiplicadorParaMs = 10000;
    // Cria fila de mensagens
    PeekMessage(&msg, NULL, 0, 0, PM_NOREMOVE);
    // Avisa que está apto a receber mensagens
    SetEvent(hEvent);
```

MsgWaitForMultipleObjects

```
    // Define timer para gerar evento a cada 1s
    hTimer = CreateWaitableTimer(NULL, FALSE, "MyTimer");   // Cria timer com reset automático
    CheckForError(hTimer);
    // Programa o temporizador para que a primeira sinalização ocorra 2s
    // depois de SetWaitableTimer.  Use - para tempo relativo
    Preset.QuadPart = -(2000 * nMultiplicadorParaMs);
    SetWaitableTimer(hTimer, (const LARGE_INTEGER *)&Preset, 5000, NULL, NULL, FALSE); // T = 2s

    for (; ;)
    {
        DwRet = MsgWaitForMultipleObjects(1, &hTimer, FALSE, INFINITE, QS_POSTMESSAGE);
        if (dwRet == WAIT_OBJECT_0)  Pid(dSetPoint, dKp, dTd, dTi, dPV); // foi o timer
        else {      // mudou algum parâmetro
        PeekMessage(&msg, NULL, 0, 0, PM_REMOVE);
        double dParametro= *((double *)msg.lParam);
        switch(msg.message)
        {
        case WM_NOVO_SET_POINT:
                printf("\nControlador: novo SP = %f\n", dParametro);
                dSetPoint = dParametro;
                break;
        case WM_NOVO_KP:
                printf("\nControlador: novo KP = %f\n", dParametro);
                dKp = dParametro;
                break;
```

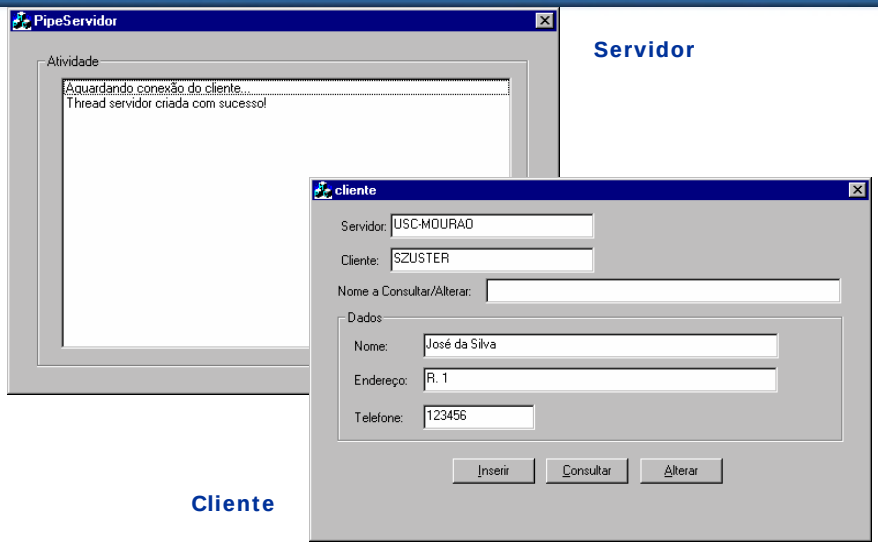
MsgWaitForMultipleObjects

```
            case WM_NOVO_TI:
                        printf("\nControlador: novo TI = %f\n", dParametro);
                        dTi = dParametro;
                        break;
            case WM_NOVO_TD:
                        printf("\nControlador: novo TD = %f\n", dParametro);
                        dTd = dParametro;
            case WM_QUIT:
                        break;
            default:
                        DispatchMessage(&msg);
            } // switch
      } // else
      } // for
      CloseHandle(hTimer);
      return 0;
}

double      Pid(double dSetPoint, double dKp, double dTd, double dTi, double dPV)
{     double dMV;       // variável manipulada
      printf("> ");
      // Calcula variável manipulada
      return(dMV);
}  // Pid
```

Nova maneira de pedir o término de uma thread

101

Exemplos em MFC - Pipes

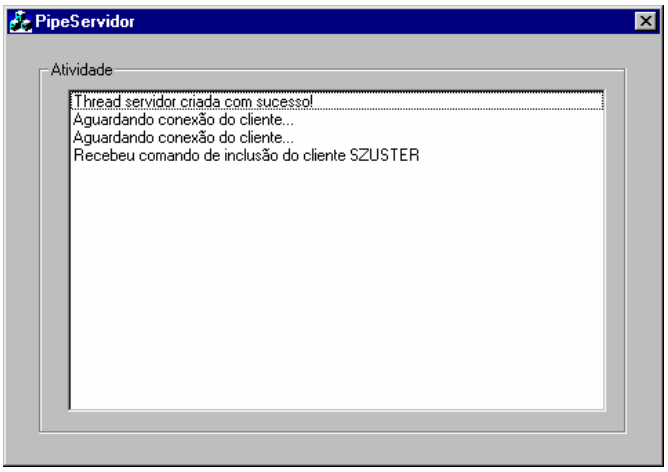


Servidor

Cliente

102

Exemplos em MFC - Pipes



103

Exemplos em MFC – Mailslots implementando um Chat

104

Exercícios

1) Classifique os exemplos de comunicação abaixo como síncrono (S) ou assíncrono (A) em analogia aos sistemas de comunicação entre processos:

( ) Dar um telefonema
( ) Enviar um e-mail
( ) Enviar um e-mail e depois correr para a porta do destinatário da mensagem até ele ler o e-mail (parece incrível, mas acontece).
( ) Colocar uma carta no correio
( ) Passar um telegrama
( ) Namorar
( ) Deixar um recado em um serviço de secretária eletrônica
( ) Colocar um aviso num mural
( ) Deixar uma mensagem para um professor no seu escaninho
( ) O serviço de Datagrama de uma rede de comunicação de dados
( ) O serviço de Circuito Virtual de uma rede de comunicação de dados

105

Exercícios

2) Numerar a segunda coluna de acordo com a primeira:

Utilização	Mecanismo de IPC
1. Deseja-se uma comunicação que funcione via rede para o WNT e Windows 95. A comunicação será unidirecional onde diversos clientes enviam mensagens para um servidor.	( ) Memória compartilhada
2. Comunicação bidirecional entre dois processos. Pode ser usada via rede. É a forma preferencial de comunicação cliente-servidor no WNT. O Windows 95 pode funcionar como cliente, mas não como servidor	( ) Pipes não nomeados
3. Comunicação unidirecional e local entre dois processos.	( ) Pipes nomeados
4. Deseja-se um mecanismo rápido de IPC para uso em uma mesma máquina. Deve funcionar no WNT e Windows 95.	( ) WM_COPYDATA
5. Único método de comunicação que funciona entre threads de 32 e 16 bits.	( ) Mailslots

106

Muito Obrigado

Perguntas?

Constantino Seixas Filho

constantino.seixas@task.com.br