UFMG

CURSO DE ESPECIALIZAÇÃO EM AUTOMAÇÃO INDUSTRIAL

Fundamentos e Aplicações de Sistemas de Automação

Módulo 4:
Semáforos, Eventos e Timers

Um inglês, mesmo que sozinho, forma uma fila ordenada de um indivíduo
(George Miges, How to be an Alien)

Professor: Constantino Seixas Filho

sábado, 7 de maio de 2005

1

Semáforos

UFMG

- Criados por Djkstra em 1968
- Inicialmente o objetivo era resolver o problema da exclusão mútua
- Hoje semáforos são um recurso mais poderoso e resolvem um grande espectro de problemas

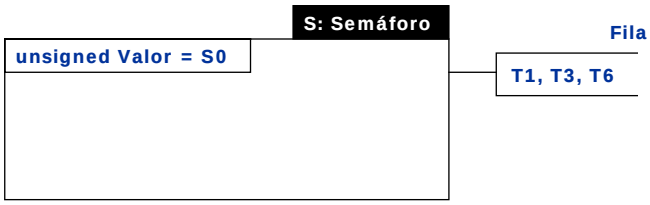
Semáforo
▪ Valor
▪ Wait() ▪ Signal()

- **Wait(S):** se S é positivo, o valor de S é decrementado. Se S é igual a zero, a thread é suspensa numa fila associada ao semáforo
- **Signal(S):** Acorda uma thread que esteja suspensa na fila do semáforo. Caso não haja nenhuma, o valor de S é incrementado

2

Fila

- **Fila pode ser:**
  - **FIFO: First In First Out**
  - **Fila de prioridade**



- **Wait(S):** **se S.Valor > 0**
  então S.Valor -= 1
  senão Suspende(S.Fila);
- **Signal(S):** **se Vazio(S.Fila)**
  então S.Valor += 1
  senão Acorda(Primeiro(S.Fila));

3

Tipo de Semáforo

- S0 = 1 – Semáforo binário ou semáforo de Djkstra
  Resolve problema da Exclusão Mútua
- S0 > 1 – Semáforo contador
  Resolve uma extensa gama de problemas

4

Solução da exclusão mútua com semáforos

```
// Variáveis globais:
Semáforo S = 1;

Thread T1;
{
    loop {
          Seção_Não_Crítica;
          Wait(S);          // Protocolo de Entrada
          Seção_Crítica;
          Signal(S);        // Protocolo de Saída
    } // end_loop
}   // ThreadT1
```

Invariantes:

- **S >= 0**
- **S = S0 - #Waits + #Signals**

5

Solução da exclusão mútua com semáforos

Invariantes:

- **S >= 0**
- **S = S0 - #Waits + #Signals**

Operação	#Waits	#Signals	S	S0 - #Waits + #Signals	Processos na Fila
Wait	1	0	2	2	0
Wait	2	0	1	1	0
Wait	3	0	0	0	0
Signal	3	1	1	1	0
Wait	4	1	0	0	0
Wait	4	1	0	0	1
Wait	4	1	0	0	2
Signal	5	2	0	0	1
Signal	6	3	0	0	0
Signal	6	3	1	1	0

6

Demonstração - Exclusão Mútua

- #P = #Wait - #Signal (1)
- S = S0 - #Wait + #Signal (2), invariante do semáforo

- Somando membro a membro as equações (1) e (2) obtemos:
  #P + S = S0 (3)

- Este é um resultado importante. Ele diz que o número de processos na seção crítica varia de 0 até um máximo que é valor de S0. Por isso estes semáforos são chamados de contadores, já que contam, geralmente em ordem decrescente, de um valor máximo S0 até 0, os processos que entraram em suas seções controladas

- No nosso caso específico S0 = 1.
  Logo: #P + S = 1 (4), ou S = 1 - #P
- Como S >= 0 $\Rightarrow$ 1- #P >=0
- $\Rightarrow$ #P <=1

c.q.d.

7

Demonstração – Ausência de deadlock

Vamos usar o princípio do absurdo:

- Vamos supor que exista deadlock. Então nenhum processo consegue entrar em sua na seção crítica, ficando preso em Wait(S)
  Daí concluímos que S = 0
- Como #P + S = 1, logo #P = 1 e existe um processo na seção crítica não existindo portanto deadlock

c.q.d.

8

Ausência de Inanição

- Se a fila for FIFO não ocorre inanição
- Se a fila for de prioridade: Processos de menor prioridade podem não ser atendidos, dependendo da taxa de chegada de processos de maior prioridade

9

Chegada na ausência de contenção

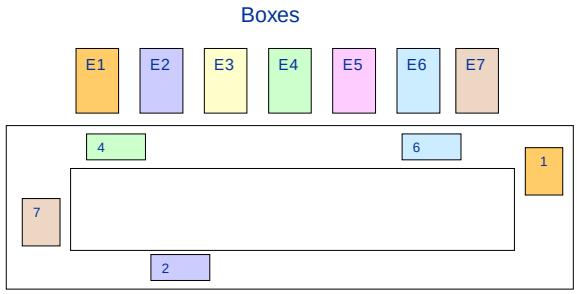
- Qualquer processo, que tente entrar na seção crítica com o semáforo igual a 1, consegue entrar livre de atrasos.

10

Contando com Semáforos

Problema:

- Você foi contratado para automatizar um treino de Fórmula 1. As regras estabelecidas pela direção da provas são simples:
  "No máximo 5 carros das 7 escuderias presentes podem entrar na pista simultaneamente, mas apenas um carro de cada equipe. O segundo carro deve ficar à espera, caso um companheiro de equipe já esteja na pista.



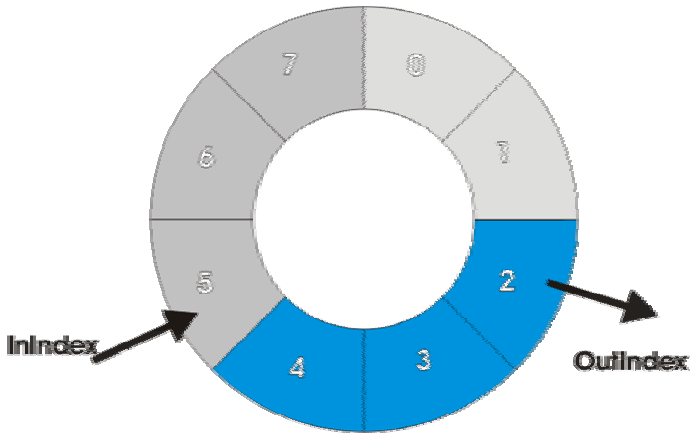
11

Solução com semáforos contadores

```
#define        EQUIPES        7

Semáforo CarrosNaPista = 5;
Semáforo Mutex[EQUIPES] = 1;        // Inicializa todo o vetor

Thread T1(int Equipe)
{
   loop {
        Wait(Mutex[Equipe]);        // Protocolo de Entrada
        Wait(CarrosNaPista);
        Entra_na_pista_e_treina();  // Seção crítica
        Signal(CarrosNaPista);      // Protocolo de Saída
        Signal(Mutex[Equipe]);
   } // end_loop
}  //  ThreadT1
```

12

Produtores e Consumidores



13

Incremento circular

- Uso de comando condicional
  indice = indice + 1;
  if (indice == N) indice = 0;
- Uso de expressão condicional
  indice = (indice == N-1) ? 0: indice + 1;
- Uso da função módulo
  indice = (indice +1) % N;

- Como fazer o decremento circular ?

14

Produtores e Consumidores - solução

```
data_type Buffer[N];              // Buffer circular
int InIndex = 0;                  // Índice para posição livre para inserção de dados
int OutIndex = 0;                 // Índice para posição ocupada contendo dado
Semaforo Cheios = 0;              // Contador de posições ocupadas
Semaforo Vazios = N;              // Contador de posições livres

Thread Produtor()
{       data_type Dado;
        loop {
                dado = ProduzDado();
                Wait(Vazios);                   // Espera que haja espaço livre
                Buffer[InIndex] = Dado;         // Insere dado no buffer
                InIndex = (InIndex + 1) % N;    // Aponta próxima posição livre
                Signal(Cheios);                 // Sinaliza Cheios
        } // end_loop
}       //  ThreadProdutor

Thread Consumidor()
{       data_type Dado;
        loop {
                Wait(Cheios);                   // Espera que haja dado para ser consumido
                Dado = Buffer[OutIndex];        // Retira dado do buffer
                OutIndex = (OutIndex + 1) % N;  // Aponta próxima posição ocupada
                Signal(Vazios);                 // Sinaliza Vazios
        } // end_loop
}       //  ThreadConsumidor
```

15

Propriedades

Pode-se provar que todas as propriedades de segurança e vitalidade são respeitadas:

- O produtor não pode inserir dados em um buffer cheio.
- O consumidor não pode consumir dados de um buffer vazio.
- Ausência de deadlock
- Não ocorre inanição de nenhum processo

Invariantes:

Seja #E o número de elementos no buffer num dado instante:

#E = Cheios

#E = N - Vazios

são invariantes no início de cada loop

16

O problema da escolha com semáforos

- Imagine a seguinte situação:
  Você dispõe de três impressoras em um sistema, o que permite que três processos possam imprimir simultaneamente. Nós desejamos obter uma das impressoras, numa certa ordem de preferência, logicamente escolhendo a impressora mais rápida, sempre que possível. Caso contrário ficamos com a que estiver disponível. Finalizada a impressão, a impressora é liberada.

17

O problema da escolha com semáforos

```
// Variáveis Globais
enum Status {LIVRE, OCUPADO};
Semáforo Cont = 3;
Semáforo Mutex = 1;
Status StatusImp[3]= {LIVRE, LIVRE, LIVRE}; // Impressoras em ordem decrescente de velocidade

Thread T1
{    int ImpIndex;
     loop {
          Wait(Cont);              // Protocolo de Entrada
          // Até 3 threads conseguiram passar até aqui agora só uma pode escolher
          Wait(Mutex);             // Ganha o direito de escolher
          for (int Indice=0; Indice <3; ++Indice) // ImpIndex= escolhe_impressora();
          if (StatusImp[Indice] == LIVRE) {
                    StatusImp[Indice] = OCUPADO;
                    ImpIndex = Indice;  break;
          } // end_if
          Signal(Mutex);           // Escolha completada
          Imprime(ImpIndex);       // Seção crítica
          Wait(Mutex);             // Libera impressora
          Status_imp[ImpIndex] =  LIVRE;
          Signal(Mutex);
          Signal(Cont);
     } // end_loop
}    //  ThreadT1
```

18

Equivalência entre semáforos contadores e semáforos binários

- Semáforos binários são mais ou menos poderosos que semáforos contadores ?
- Para provar que semáforos binários são equivalentes ou mais poderosos que semáforos contadores devemos provar que é possível criar semáforos contadores em um sistema que só tem semáforos binários
- Para provar que semáforos contadores são equivalentes ou mais poderosos que semáforos binários devemos provar que é possível criar semáforos binários a partir de semáforos contadores. Esta solução é trivial. Basta criar um semáforo contador com o valor inicial de 1
- Se a>=b e b >=a então ....

19

Equivalência entre semáforos contadores e semáforos binários

```
class Semaforo {
        private:
                SemaforoBinario Mutex;
                SemaforoBinario Delay;
                int Cont;
        public:
                Semaforo(int S0) {IniSemBin(Mutex, 1); IniSemBin(Delay, 1); Cont=S0; }
                ~Semaforo() {          }

                inline void Semaforo::WaitCont() {
                  Wait(Delay);          // se Cont = 0 fique esperando aqui
                  Wait(Mutex);          // decrementa Cont e testa se é zero com exclusão mútua
                  Cont = Cont - 1;
                  if (Cont > 0)         // se maior que zero mantenha porta de entrada aberta
                  Signal(Delay);
                  Signal(Mutex);
                }  // WaitCont

                inline void Semaforo::SignalCont() {
                  Wait(Mutex);          // incrementa cont e testa se acaba de se tornar 1
                  Cont = Cont + 1;      // com exclusão mútua
                  if (Cont == 1) Signal(Delay);     // maior que 1 pode abrir a porta de novo
                  Signal(Mutex);
                }  // SignalCont
} // Semaforo
```

20

O problema da alocação de recursos
O Jantar dos Filósofos

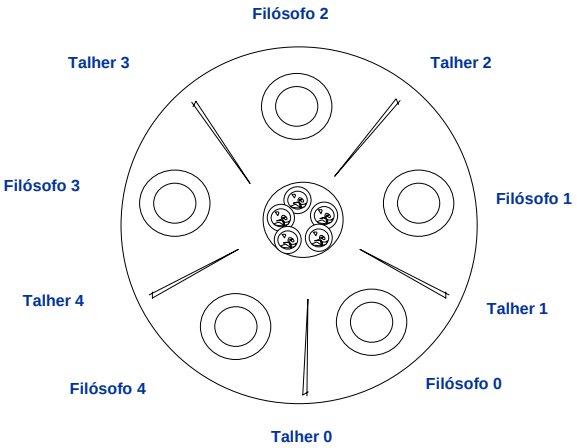
- Cinco filósofos estão sentados em torno de uma mesa circular, que tem em seu centro um prato inesgotável de sushis. Sobre a mesa, entre cada dois filósofos há um *hashi*. Cada filósofo, para comer, deve pegar dois talheres. Cada filósofo realiza um loop infinito em que pensa, toma os talheres um a um, come e devolve os talheres à mesa.

As regras a serem obedecidas são:

- P1: Dois filósofos não podem segurar um mesmo talher simultaneamente
- P2: O filósofo só come, quando tem dois talheres
- P3: Não deve haver deadlock, situação em que nenhum dos filósofos consegue comer
- P4: Não pode haver inanição (neste caso inanição propriamente dita), isto é, um filósofo querendo comer, deve eventualmente ter acesso aos dois talheres. Eventualmente aqui significa: o evento (comer) ocorre com certeza, em algum instante no futuro

21

O problema da alocação de recursos
O Jantar dos Filósofos



22

Jantar dos filósofos

```
Thread FilosofoN {
loop {
    Pensa;
    Conquista_talheres();   // Protocolo de entrada
    Come();
    Libera_talheres();      // Protocolo de Saída
    }      // end_loop
}
```


Primeira solução

```
Thread Filosofo(int i)        // i é o índice do filósofo
{   loop {
          Pensa();
          Wait(Talher[i]);              // Pega talher à sua esquerda
          Wait(Talher[(i+1) % 5]);      // Pega talher à sua direita
          Come();
          Signal(Talher[i]);            // Libera talher à esquerda
          Signal(Talher[(i+1) % 5]);    // Libera talher à direita
    } // end_loop
}   //  Filósofo
```

Qual o problema com esta solução ?

Solução Tanenbaum

```
#define    ESQUERDA            (i+N-1)%N
#define    DIREITA             (i+1)%N
enum status {PENSANDO, COM_FOME, COMENDO};
status Estado[N];
Semaforo Mutex = 1;
Semaforo S[N];                 // Inicialmente é 0

Thread Filosofo(int i)         // i é o índice do filósofo
{
    loop {
        Pensa();
        Pega_talheres(i);      // Pega ambos os talheres
        Come();
        Devolve_talheres(i);   // Devolve talheres
    } // end_loop
}   // Filósofo

void Pega_talheres(int i)
{
    Wait(Mutex);
    Estado[i] = COM_FOME;      // Anuncia que quer comer
    Test(i);                   // Pede talheres
    Signal(Mutex);
    Wait(S[i]);                // Fica a espera de talheres livres
}
```

25

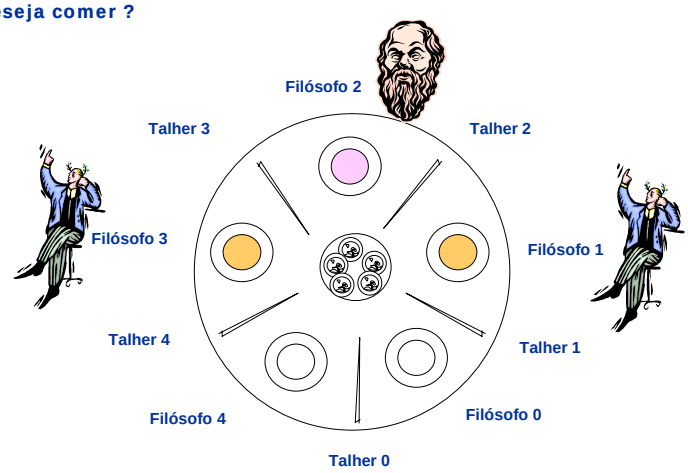
Solução Tanenbaum

```
void Devolve_talheres(int i)
{
    Wait(Mutex);
    Estado[i] = PENSANDO;      // Define novo estado
    Test(ESQUERDA);            // Verifica se o estado do filósofo à esquerda mudou
    Test(DIREITA);             // Verifica se o estado do filósofo à direita mudou
    Signal(Mutex);
}

void Test(int i) {
    if (Estado[i] == COM_FOME && Estado[ESQUERDA] != COMENDO && Estado[DIREITA] !=
    COMENDO) {
        Estado[i] = COMENDO;
        Signal(S[i]);          // Se o filósofo tem os dois talheres disponíveis, acorde-o
    } // if
}
```

26

Conspiração dos filósofos

- **O que acontece se os filósofos 1 e 3 combinarem de comer de forma sincronizada ? O filósofo 1 começa a comer e antes que termine, o filósofo 3 também começa a comer. O filósofo 1 para de comer, mas reassume antes que 3 termine e assim alternadamente. O que ocorre com o filósofo 2, que também deseja comer ?**



27

O problema dos leitores e escritores

Algumas regras básicas do problema são:

- Vários leitores podem efetuar a leitura concorrentemente.
- Se um escritor estiver acessando a base de dados, este acesso deve excluir qualquer outro escritor, e também qualquer leitor.
- Não deve haver deadlock
- Nenhum leitor e nenhum escritor deve sofrer de inanição.
- Na ausência de disputa, tanto leitores como escritores devem ter acesso simplificado.

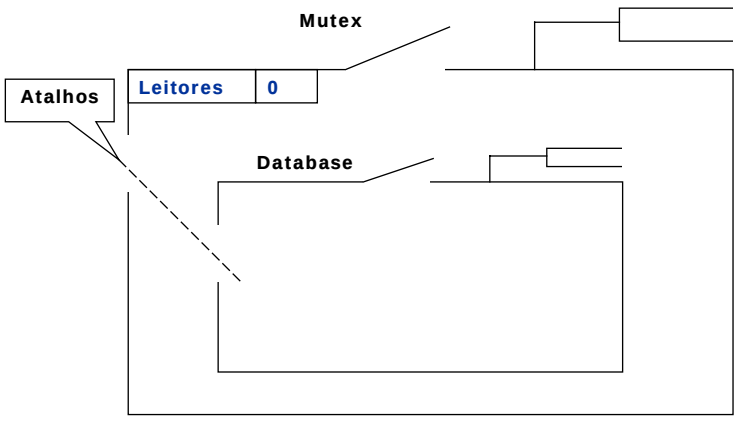
28

Leitores e Escritores

```
Semaforo Mutex = 1;      // Assegura exclusão mútua no acesso à variável leitores
Semaforo Database = 1;   // Organiza fila de espera de escritores e até um leitor
int Leitores = 0;        // Número de processos lendo ou querendo ler

Thread Leitor() {
     loop {
          Wait(Mutex);
          Leitores = Leitores + 1;   //  Mais um leitor deseja ler
          if (Leitores == 1) Wait(Database);      // Se é o primeiro espera Database livre
          Signal(Mutex);                          // senão tem leitor: pode entrar também
          LeDados();
          Wait(Mutex);
          Leitores = Leitores - 1;
          if (Leitores == 0) Signal(Database);    // Se é último leitor habilita um escritor
          Signal(Mutex);
          UsaDado();
     } // end_loop
}  // Leitor

Thread Escritor() {
     loop {
          ProduzDado();
          Wait(Database);          // Espera se tem alguém acessando o banco de dados
          EscreveDado();
          Signal(Database);
          }  // end_loop
}
```

29

Leitores e Escritores



Movimento de leitura: R1 R2 R3 W1 W2 R4 R5 W3 R6

30

Problema: O Jantar dos Selvagens [Andrew 91]

- Uma tribo de selvagens janta em conjunto, retirando missionários assados de um grande pote que comporta até M missionários. Quando um selvagem deseja comer ele se serve do pote, a menos que o pote esteja vazio. Se o pote está vazio, o selvagem acorda um cozinheiro e espera até que este tenha assado mais missionários e enchido o pote. Nos programas abaixo, substitua os marcadores em itálico por instruções em pseudo linguagem de modo a garantir a perfeita sincronização entre cozinheiro e selvagens.

```
Thread cozinheiro {
    loop {
      espera_ser_chamado;
      cozinha();
      enche_o_pote_e_avisa;
    } // end_loop
}

Thread selvagem {
    loop {
      descansa();
      serve_se;
      come();
    } // end_loop
}
```

31

Semáforos no WNT

```
HANDLE CreateSemaphore(
LPSECURITY_ATTRIBUTES lpSemaphoreAttributes,   // Atributos de segurança
LONG lInitialCount,                            // Valor Inicial
LONG lMaximumCount,                            // Valor Máximo
LPCTSTR lpName                                 // Nome do objeto
);
```

Retorno da função:

Status	Interpretação
Handle para o Semáforo criado	Sucesso
NULL	Falha

32

OpenSemaphore

```
HANDLE OpenSemaphore (
DWORD dwDesiredAccess,        // Atributos de segurança:
                              SEMAPHORE_ALL_ACCESS: o handle pode ser usado em qualquer função
                              SEMAPHORE_MODIFY_STATE: o handle retornado pode ser usado na função ReleaseSemaphore()
                              SYNCHRONIZE: o handle pode ser usado apenas nas funções Wait... e ReleaseSemaphore()
BOOL bInheritHandle,          // TRUE: handle será herdável
LPCTSTR lpName                // Nome do Semáforo a ser aberto
);
```

Retorno da função:

Status	Interpretação
Handle para o Semáforo aberto	Sucesso
NULL	Falha

33

Wait/Signal _> WaitForSingleObject/ReleaseSemaphore

```
DWORD WaitForSingleObject(
HANDLE hSemaphore,            //  Handle para o semáforo
DWORD dwMilliseconds          //  Tempo máximo que desejamos esperar
);
```

```
BOOL ReleaseSemaphore (
HANDLE hSemaphore,            // Handle para semáforo a ser incrementado
LONG lReleaseCount,           // Valor a ser adicionado ao valor corrente do semáforo
LPLONG lpPreviouCount         // Recebe valor anterior do semáforo
);
```

Retorno da função:

Status	Interpretação
TRUE	Sucesso
FALSE	Falha

34

O Problema do treino de fórmula 1 revisitado

```
#define WIN32_LEAN_AND_MEAN
#include <windows.h>
#include <stdio.h>
#include <stdlib.h>
#include <process.h>                    // _beginthreadex() e _endthreadex()
#include <conio.h>                      // _getch

#define _CHECKERROR   1                 // Ativa função CheckForError
#include "CheckForError.h"
// Casting para terceiro e sexto parâmetros da função _beginthreadex
typedef unsigned (WINAPI *CAST_FUNCTION)(LPVOID);
typedef unsigned *CAST_LPDWORD;
#define EQUIPES                         7
#define MAX_CARROS_PISTA                5
#define NUM_CARROS                      10          // número de carros na simulação
HANDLE hMutex[EQUIPES];
HANDLE hSemaphore;
DWORD WINAPI FuncCar(LPVOID);                       // declaração da função

int main()
{
    HANDLE hThreads[NUM_CARROS];
    DWORD dwThreadId;
    DWORD dwExitCode = 0;
    DWORD dwRet;
    int     nEquipe, nCar;
    char    BoxName[5];
```

35

O Problema do treino de fórmula 1 revisitado

```
    for (nEquipe=0; nEquipe<EQUIPES; ++nEquipe) {     // cria Mutexes, um para cada Equipe
        sprintf(BoxName, "Box%d", nEquipe);
        hMutex[nEquipe] = CreateMutex(NULL, FALSE, BoxName);
        CheckForError(hMutex[nEquipe]);
    } //for
    hSemaphore = CreateSemaphore(NULL,MAX_CARROS_PISTA,MAX_CARROS_PISTA,"MAX_CARROS");
    for (nCar=0; nCar<NUM_CARROS; ++nCar) {
        // cria threads, uma para cada carro
        nEquipe = rand() % EQUIPES;
        // cada thread representa um carro de uma equipe
        hThreads[nCar] = (HANDLE) _beginthreadex(
                NULL,  0,  (CAST_FUNCTION)FuncCar,
                (LPVOID)((nCar<<8) + nEquipe),
                0,  (CAST_LPDWORD)&dwThreadId );
        if (hThreads[nCar]) printf("Carro %2d Equipe %d Id= %0x \n", nCar, nEquipe, dwThreadId);
    } // for
    // Espera todas as threads terminarem
    dwRet = WaitForMultipleObjects(NUM_CARROS,hThreads,TRUE,INFINITE);
    CheckForError((dwRet >= WAIT_OBJECT_0) && (dwRet < WAIT_OBJECT_0 + NUM_CARROS));
    for (nCar=0; nCar<NUM_CARROS; ++nCar) {
        dwRet=GetExitCodeThread(hThreads[nCar], &dwExitCode);
        CheckForError(dwRet);
        CloseHandle(hThreads[nCar]);                // apaga referência ao objeto
    } // for
    for (nEquipe=0; nEquipe < EQUIPES; ++nEquipe)  CloseHandle(hMutex[nEquipe]);
    CloseHandle(hSemaphore);
    printf("\nAcione uma tecla para terminar\n");
    _getch();                                       // Pare aqui, caso não esteja executando no ambiente MDS
    return EXIT_SUCCESS;
    } // main
```

36

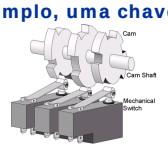
O Problema do treino de fórmula 1 revisitado

```
DWORD WINAPI FuncCarr(LPVOID id)
{
    LONG lOldValue;
    int       nCar, iTeam;

    iTeam= (DWORD)id % 256;
    nCar = (DWORD)id / 256;

    for (int i=0;i<3;++i) {  // dá 3 voltas na pista
            printf("Carro %d da Equipe %d quer treinar... volta %d \n", nCar, iTeam, i);
            WaitForSingleObject(hMutex[iTeam], INFINITE);
            WaitForSingleObject(hSemaphore, INFINITE);
            printf("Carro %d da Equipe %d treinando... volta %d \n", nCar, iTeam, i);
            Sleep(100*(rand() % 10));          // corre durante certo tempo
            ReleaseSemaphore(hSemaphore, 1, &lOldValue);
            ReleaseMutex(hMutex);
            printf("Carro %d da Equipe %d acabou de treinar... volta %d \n", nCar, iTeam, i);
    } // for
    _endthreadex(0);
    return(0);
} // FuncCar
```

37

Eventos

1. **Eventos de processo:**
   São gerados no campo, quando por exemplo, uma chave de nível alto de um silo é acionada



2. **Eventos de operação:**
   Têm origem no *cockpit* de operação, constituído pela Interface Humano Computador. O evento é gerado toda vez que o operador clica o botão de liga de uma janela de operação, ou quando aciona uma tecla no teclado funcional de um SDCD. Em sistema telecomandados o evento de operação pode ser enviado via a rede de comunicação de dados. Um operador situado a milhares de quilômetros de distância pode deflagrar todo o processo para, por exemplo, abrir uma comporta de uma usina hidrelétrica.

3. **Evento de tempo:**
   É gerado quando um intervalo de tempo especificado expira. Pode ser cíclico ou eventual.

38

Eventos

```
HANDLE Create Event(
LPSECURITY_ATTRIBUTES lpEventAttributes,   //  Atributos de segurança.
BOOL bManualReset,                         //  Reset Manual /Automático.
BOOL bInitialState,                        // Estado Inicial.
LPCTSTR lpName                             // Nome do objeto.
);
```

Retorno da função:

Status	Interpretação
Handle para o Evento criado	Sucesso
NULL	Falha

39

Eventos

```
BOOL SetEvent(HANDLE hEvent);     // Define o estado do evento como sinalizado

BOOL ResetEvent(HANDLE hEvent);   // Define o estado do evento como não sinalizado

BOOL PulseEvent(HANDLE hEvent);   // O evento se torna sinalizado e logo depois não
                                  // sinalizado. Ele ficará sinalizado pelo tempo
                                  // necessário para acordar as threads que estiverem
                                  // esperando pelo evento.
```

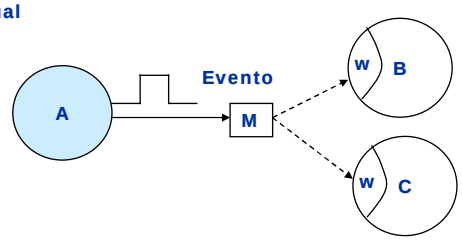
Retorno da função:

Status	Interpretação
TRUE	Sucesso
FALSE	Falha

40

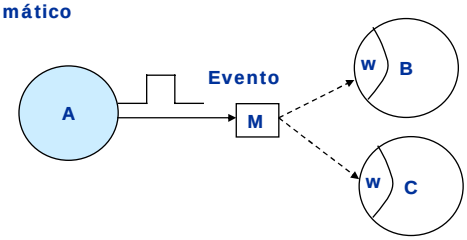
Eventos

Reset Manual



B e C acordam

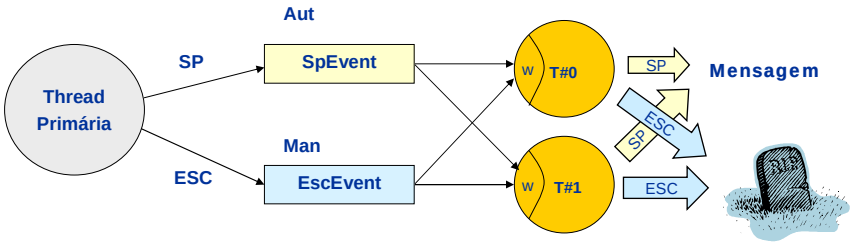
Reset Automático



A thread que fez Wait... primeiro acorda

41

Eventos - Exercício



Se o usuário pressionar espaço, será gerado um evento com reset automático através da função *PulseEvent()*. Uma das threads à espera do evento será sinalizada, acordará e imprimirá uma mensagem na tela, voltando ao loop de espera. Para terminar o processo o usuário deve pressionar Escape, para geração de um evento sem reset automático. Este evento acordará as duas threads e uma vez identificado, causará o término de ambas.

42

Exercício - Eventos

```
#define     SP                  0x20
#define     ESC                 0x1B
#define     NUM_THREADS         2
HANDLE hSpEvent;                // Handle para Evento
HANDLE hEscEvent;               // Handle para Evento Aborta
DWORD WINAPI WaitEventFunc(LPVOID);       // declaração da função

int main()
{
    HANDLE hThreads[NUM_THREADS];

    DWORD dwThreadId;
    DWORD dwExitCode = 0;
    DWORD dwRet;
    int i, nTecla;
    printf("Criando um objeto do tipo Evento com Reset Automatico\n");
    // apenas uma thread é acordada a cada pulso
    hSpEvent = CreateEvent(NULL, FALSE, FALSE,"SpEvento");
    CheckForError(hEvent);
    // Cria Evento com Reset Manual:
    // todas as threads são acordadas a cada pulso
    hEscEvent = CreateEvent(NULL, TRUE, FALSE,"EscEvento");
    CheckForError(hEscEvent);
```

43

Exercício - Eventos

```
    for (i=0; i<NUM_THREADS; ++i) {             // cria duas threads
        hThreads[i] = (HANDLE) _beginthreadex(
                NULL,  0,
                (CAST_FUNCTION) WaitEventFunc, (LPVOID)i,
                0,  (CAST_LPDWORD)&dwThreadId );
        if (hThreads[i]) printf("Thread %d criada Id= %0x \n", i, dwThreadId);
    }  // for

    do {    printf("Tecle <SP> para gerar evento ou <Esc> para terminar\n");
        nTecla = _getch();
        if (nTecla == SP) PulseEvent(hSpEvent);         // Gera 1 evento
        else if (nTecla == ESC) PulseEvent(hEscEvent);  // Termina threads
    } while (nTecla != ESC);
    // Espera todas as threads terminarem
    dwRet = WaitForMultipleObjects(NUM_THREADS,hThreads,TRUE,INFINITE);
    CheckForError(dwRet==WAIT_OBJECT_0);

    for (i=0; i<NUM_THREADS; ++i) {
        GetExitCodeThread(hThreads[i], &dwExitCode);
        CloseHandle(hThreads[i]);               // apaga referência ao objeto
    }  // for
    CloseHandle(hSpEvent);
    CloseHandle(hEscEvent);
    printf("\nAcione uma tecla para terminar\n");
    _getch();  // Pare aqui, caso não esteja executando no ambiente MDS – Microsoft Developer Studio
    return EXIT_SUCCESS;
}  // main
```

44

Exercício - Eventos

```
DWORD WINAPI WaitEventFunc(LPVOID id)
{
    HANDLE Events[2]= {hSpEvent, hEscEvent};
    DWORD ret;
    int nTipoEvento;

    do {
        printf("Thread %d esperando evento\n", id);
        ret=WaitForMultipleObjects(2, Events, FALSE, INFINITE);
        nTipoEvento = ret - WAIT_OBJECT_0;
        if (nTipoEvento == 0) printf("Evento %d \n", id);
    } while (nTipoEvento == 0);          // Esc foi escolhido
    printf("Thread %d terminando...\n", id);

    _endthreadex(0);
    return(0);
} // WaitEventFunc
```

45

Timers 1: Sleep

```
loop() {
    LeEntradas();
    ProcessaLógica();
    DefineSaídas();
    Sleep(tempo);
}
```

46

Timers 2: Uso de timeout

```
loop() {
    dwRet = WaitForSingleObject(hObjeto, tempo);
    if (dwRet == TIMEOUT) {
        // Realiza atividade associada a evento de tempo
    }
    else {
        // Realiza atividade associada a evento esperado
    }
}
```

Quais são as desvantagens desta temporização ?

47

Timers 3: Uso de WM_TIMER

- **Geram uma mensagem do tipo WM_TIMER ou chamam rotina callback**
- **Mensagens WM_TIMER são as de menor prioridade**
- **Apenas uma mensagem WM_TIMER pode estar na fila de cada vez**
- **Baseado em timer de HW que fornece ticks a cada 54.9 ms**

```
UINT SetTimer(
HWND hWnd,          // Handle para janela destino da mensagem
UINT nIDEvent,      // Identidade do temporizador.
UINT uElapsed,      // Valor a ser temporizado em ms.
TIMERPROC lpTimerFunc   // Endereço da rotina callback associada ao temporizador
);
```

Retorno da função:

Status	Interpretação
<>0	Valor do identificador do timer.
0	A função falhou na criação do timer.

48

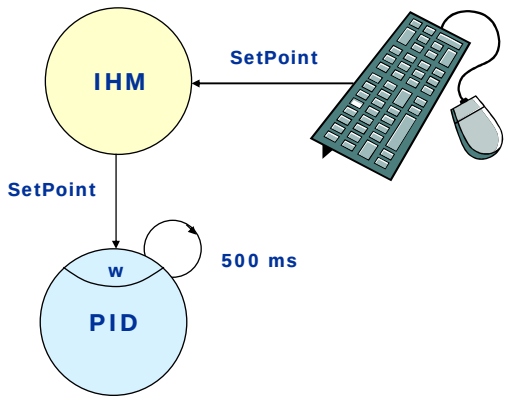
WM_TIMER

```
BOOL KillTimer
HWND hWnd,          // Handle para a janela que gerou o temporizador
UINT uIDEvent       // Identificador do timer
);
```

Retorno da função:

Status	Interpretação
<>0	SUCESSO
0	FALHA

Problema



WM_TIMER

```
#include "bGetFloat.h"

#define _CHECKERROR1            // Ativa função CheckForError
#include "CheckForError.h"

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

#define ESC                     0x1B
#define CR                      0x0D

HANDLE hEvent;                  // Handle para Evento
HANDLE hEscEvent;               // Handle para Evento Aborta
HANDLE hEventGotData;           // Handle para Evento "Peguei dado"

DWORD WINAPI PidControlFunc(LPVOID);       // declaração da função
VOID CALLBACK Pid(HWND hwnd, UINT uMsg, UINT idEvent, DWORD dwTime);  // Função PID

double SetPoint= 0.0;
double dInput;
```

51

WM_TIMER

```
int main()
{
    HANDLE hThread;
    DWORD dwThreadId;
    DWORD dwExitCode = 0;
    BOOL  bStatus;

    printf("Demonstrando o uso da funcao SetTimer com uma Working Thread\n");
    hEvent  = CreateEvent(NULL, FALSE, FALSE,"Evento");      // apenas uma thread é acordada a cada pulso
    CheckForError(hEvent);

    hEscEvent = CreateEvent(NULL, TRUE, FALSE,"EscEvento"); // todas as threads são acordadas
    CheckForError(hEscEvent);

    // Sincronismo: peguei dado
    hEventGotData= CreateEvent(NULL, FALSE, FALSE,"EscEventGotData");
    CheckForError(hEventGotData);

    hThread = (HANDLE) _beginthreadex(
                    NULL,   0,
                    (CAST_FUNCTION)PidControlFunc,  (LPVOID)0,
                    0,  (CAST_LPDWORD)&dwThreadId);
    CheckForError(hThread);
```

52

WM_TIMER

```
    do {    printf("\nEscreva novo valor de SetPoint: ");
            bStatus = bGetFloat(&dInput, 6); // Le string ou ESC/CR
            if (bStatus) {
                    SetEvent(hEvent);   // SetPoint mudou
                    WaitForSingleObject(hEventGotData, INFINITE);
                    ResetEvent(hEventGotData);
            }
            else SetEvent(hEscEvent);           // Termina threads
    } while (bStatus);

    // Espera thread terminar
    GetExitCodeThread(hThread, &dwExitCode);

    CloseHandle(hThread);          // apaga referência ao objeto
    CloseHandle(hEvent);
    CloseHandle(hEscEvent);
    CloseHandle(hEventGotData);

    printf("\nAcione uma tecla para terminar\n");
    _getch(); // Pare aqui, caso não esteja executando no ambiente MDS - Microsoft Developer Studio

    return EXIT_SUCCESS;
} // main
```

53

WM_TIMER

```
DWORD WINAPI PidControlFunc(LPVOID id)
{
    HANDLE Events[2]= {hEvent, hEscEvent};
    int     TipoEvento= 0;
    UINT    nTimerId;
    MSG     msg;
    DWORD  dwRet;
    // Esta não é uma GUI thread: a fila de mensagens deve ser criada artificialmente
    PeekMessage(&msg, NULL, 0, 0, PM_NOREMOVE);
    nTimerId = SetTimer( NULL,    // handle para janela
                         NULL,    // Identidade do temporizador
                         2000,    // Valor temporizado em ms
                         (TIMERPROC)Pid);    // Irá utilizar função callback
    do {              dwRet= MsgWaitForMultipleObjects(2, Events, FALSE, INFINITE, QS_TIMER);
            TipoEvento = dwRet - WAIT_OBJECT_0;
            if (TipoEvento == 0) {  // Novo valor de SP
                                    SetPoint= dInput;
                                    printf("\nNovo valor de set point: %6.2f", SetPoint);
                                    ResetEvent(hEvent);
                                    SetEvent(hEventGotData);
                                }
            else if (TipoEvento == 2) { // foi o Timer
                                    PeekMessage(&msg, NULL, 0, 0, PM_REMOVE);
                                    DispatchMessage(&msg); }  // Rotina callback será invocada
    } while (TipoEvento != 1);      // Esc foi escolhido
```

54

WM_TIMER

```
    KillTimer(0, nTimerId);                    // Desativa o temporizador

    printf("\nThread Pid terminando...\n");
    _endthreadex((DWORD) 0);
    return(0);
} // WaitEventFunc

VOID CALLBACK Pid(HWND hwnd, UINT uMsg, UINT idEvent, DWORD dwTime)
{   // Algoritmo Pid;
    //printf("\n SP=%6.2f ", SetPoint);
    MessageBeep(MB_OK) ;
};  // Pid
```

55

Waitable Timers

```
HANDLE CreateWaitableTimer(
LPSECURITY_ATTRIBUTES lpEventAttributes,    // Apontador para atributos de segurança.
BOOL bManualReset,                          // Reset Manual /Automático.
LPCTSTR lpTimerName);                       // Nome do objeto.
```

bManualReset

- **TRUE: Reset Manual.** O timer é um temporizador de notificação. Após o tempo expirar, ele ficará sinalizado e deverá ser reprogramado pela função *SetWaitableTimer()*. Todas as threads esperando pela sinalização do timer serão acordadas e se tornarão escalonáveis, tal como ocorre com um objeto do tipo evento de reset manual.
- **FALSE: Reset Automático.** O timer funciona como temporizador de sincronização. Ele ficará sinalizado até que uma thread realize uma operação de Wait no objeto. Apenas uma thread esperando pela sinalização do timer se tornará escalonável

Retorno da função:

Status	Interpretação
Handle para o Timer criado	Sucesso
NULL	Falha

56

Waitable Timer

```
HANDLE OpenWritableTimer(
DWORD dwDesiredAccess,          //  Atributos de segurança:
                                TIMER_ALL_ACCESS: o handle pode ser usado em qualquer função.
                                TIMER_MODIFY_STATE: o handle retornado pode ser usado na função
                                SetWaitableTimer() e CancelWaitableTimer().
                                SYNCHRONIZE: o handle pode ser usado apenas nas funções Wait…
BOOL bInheritHandle,            //  TRUE: handle será herdável
LPCTSTR lpTimerName             //  Nome do Timer a ser aberto
);
```

Retorno da função:

Status	Interpretação
Handle para o Timer aberto	Sucesso
NULL	Falha

57

SetWaitableTimer

```
BOOL SetWaitableTimer(
HANDLE hTimer,                          // Handle para timer
const LARGE_INTEGER *pDueTime,          // Quando o timer será sinalizado.
LONG lPeriod                            //  Período do timer em ms
                                           0: 1 vez
                                           >0: Periódico
PTIMERAPCROUTINE pfnCompletionRoutine,  // Apontador para a APC.
LPVOID lpArgToCompletionRoutine,        // Argumentos a serem passados para rotina.
BOOL fResume                            // Sai de modo Power saver
);
```

Retorno da função:

Status	Interpretação
TRUE	SUCESSO
FALSE	FALHA

58

Definição de um valor relativo para pDueTime

```
LARGE_INTEGER Atraso;
// Programa o temporizador para que a primeira sinalização ocorra 2s
// depois de SetWaitableTimer
// Define uma constante para acelerar cálculo do atraso e período
const int nMultiplicadorParaMs = 10000;
//  Use – para tempo relativo
//  typedef union _LARGE_INTEGER {
//  struct {
//    DWORD LowPart;
//    LONG HighPart;
//    };
//    LONGLONG QuadPart;
// } LARGE_INTEGER;
Atraso.QuadPart = -(2000 * nMultiplicadorParaMs);
// Dispara o temporizador depois de atraso e a cada 2s
SetWaitableTimer(hTimer, &Atraso, 2000, NULL, NULL, FALSE);
```

59

Definição de um valor absoluto para pDueTime

```
SYSTEMTIME st;
FILETIME ftLocal, ftUTC;
LARGE_INTEGER liUTC;
// Primeira sinalização deverá ocorrer em 1 de dezembro de 1999 às 17 horas (tempo local)
// Nós devemos definir uma estrutura do SYSTEMTIME
st.wYear  = 1999;    // Ano
st.wMonth= 12;       // Dezembro
st.wDayOfWeek= 0;  // Ignorado
st.wDay    = 1;       // Dia
st.wHour  = 17;       // Hora
st.wMinute  = 0;       // Minutos
st.wSecond  = 0;       // Segundos
st.wMilliseconds= 0; // Milisegundos
// A estrutura SYSTEMTIME é fácil de ser definida, mas é difícil de ser operada
// Nós devemos convertê-la para uma estrutura do tipo FILETIME
// A primeira conversão se dá para FILETIME local
SystemTimeToFileTime(&st, &ftLocal);
// Agora deve-se converter de FILETIME local para Coordinated Universal Time  (UTC) FILETIME UTC
LocalFileTimeToFileTime(&ftLocal, &ftUTC);
// Finalmente convertemos de  FILETIME UTC para LARGE_INTEGER para assegurar que a estrutura
     esteja alinhada numa fronteira de 64 bits.
liUTC.LowPart  = ftUTC.dwLowDateTime;
liUTC.HighPart = ftUTC.dwHighDateTime;
// Agora podemos disparar o temporizador: UFA !!!
SetWaitableTimer(hTimer, &liUTC, 2000, NULL, NULL, FALSE);
```

60

Waitable Timer – Cancel

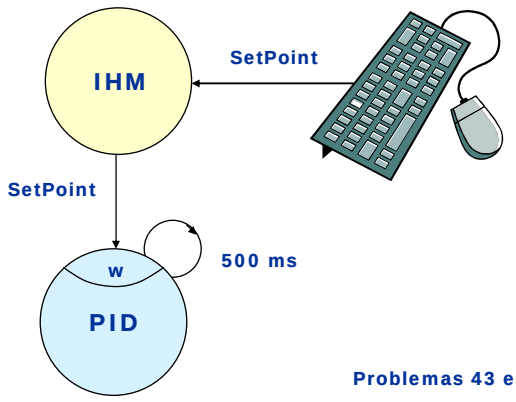
```
BOOL CancelWaitableTimer(HANDLE hTimer);   // Handle para timer
```

Retorno da função:

Status	Interpretação
!=0	SUCESSO
0	FALHA

61

Problema



Problemas 43 e 44

62

Waitable Timer - Exemplo

```
#define     ESC                     0x1B
#define     CR                      0x0D

HANDLE hEvent;                      // Handle para Evento
HANDLE hEscEvent;                   // Handle para Evento Aborta
HANDLE hTimer;                      // Handle para Timer

DWORD WINAPI PidControlFunc(LPVOID);        // declaração da função
VOID Pid(double );                          // Função PID
double SetPoint= 0.0;
double dInput;

int main()
{
    HANDLE hThread;
    DWORD dwThreadId;
    DWORD dwExitCode = 0;
    BOOL  bStatus;
    LARGE_INTEGER Preset;
    BOOL bSucesso;
    // Define uma constante para acelerar cálculo do atraso e período
    const int nMultiplicadorParaMs = 10000;

    printf("Criando um objeto do tipo Evento com reset automatico\n");
    // apenas uma thread é acordada a cada pulso
    hEvent              = CreateEvent(NULL, FALSE, FALSE,"Evento");
    CheckForError(hEvent);
    // todas as threads são acordadas a cada pulso
    hEscEvent           = CreateEvent(NULL, TRUE, FALSE,"EscEvento");
    CheckForError(hEscEvent);
```

3

Waitable Timer

```
// Cria timer com reset automático
hTimer = CreateWaitableTimer(NULL, FALSE, "MyTimer");
CheckForError(hTimer);

hThread = (HANDLE) _beginthreadex(
                    NULL,       0,
                    (CAST_FUNCTION)PidControlFunc,  (LPVOID)0,
                    0,          (CAST_LPDWORD)&dwThreadId );
if (hThread) printf("Thread Pid criada Id= %0x \n", dwThreadId);

// Programa o temporizador para que a primeira sinalização ocorra 2s
// depois de SetWaitableTimer
// Use - para tempo relativo
Preset.QuadPart = -(2000 * nMultiplicadorParaMs);
// Dispara timer
bSucesso = SetWaitableTimer(hTimer, &Preset, 5000, NULL, NULL, FALSE);
CheckForError(bSucesso);

do {
        printf("\nEscreva novo valor de SetPoint:");
        // le string ou ESC
        bStatus = bGetFloat(&dInput, 6);
        if (bStatus) PulseEvent(hEvent);  // SetPoint mudou
        else PulseEvent(hEscEvent);       // Termina threads
} while (bStatus);

// Espera thread terminar
GetExitCodeThread(hThread, &dwExitCode);
```

64

Waitable Timer

```
    CloseHandle(hThread);          // apaga referência ao objeto
    CloseHandle(hEvent);
    CloseHandle(hEscEvent);
    CloseHandle(hTimer);

    printf("\nAcione uma tecla para terminar\n");
    _getch();  // Pare aqui, caso não esteja executando no ambiente MDS
    return EXIT_SUCCESS;
}  // main

DWORD WINAPI PidControlFunc(LPVOID id)
{
    HANDLE Events[3]= {hEvent, hEscEvent, hTimer};
    DWORD ret;
    int TipoEvento= 0;

    do {    ret=WaitForMultipleObjects(3, Events, FALSE, INFINITE);
            TipoEvento = ret - WAIT_OBJECT_0;
            if (TipoEvento == 0){   // Novo valor de Pid
                    SetPoint= dInput;
                    printf("\nNovo valor de set point: %f", SetPoint);  }
            else if (TipoEvento == 2) Pid(SetPoint);          // ocorreu ativação de tempo
    } while (TipoEvento != 1);                                // Esc foi escolhido

    printf("\nThread Pid terminando...\n");
    _endthreadex((DWORD) 0);
    return(0);
} // WaitEventFunc
```

65

Waitable Timer

```
VOID Pid(double SetPoint)
{    // Algoritmo Pid;
     printf(" >");
};   // Pid
```

66

Timers Multimídia

- O HAL (*Hardware Abstraction Layer*) gera interrupções de tempo periodicamente para o kernel a cada 10 ou 15 ms.
- Como o HAL é aberto, cada projetista pode implementar o timer de uma forma diferente
- Alguns irão utilizar o chip de clock 8254, enquanto outros usarão o RTC para gerar as interrupções
- Funciona no Windows 95..XP

67

timeGetDevCaps

Indagar a precisão do relógio

```
MMRESULT timeGetDevCaps(
LPTIMECAPS ptc,     //  Endereço da estrutura TIMECAPS
                        typedef struct {
                            UINT wPeriodMin;  // limite mínimo suportado
                            UINT wPerio dMax; // limite máximo suportado
                        } TIMECAPS;
UINT cbtc           //  Tamanho em bytes da estrutura TIMECAPS
);
```

Retorno:

Status	Interpretação
TIMERR_NOERROR	Sucesso
TIMERR_STRUCT	Falha

68

Timer Multimídia

```
MMRESULT timeBeginPeriod(
UINT uPeriod                // Menor resolução do timer em milisegundos
);
```

```
MMRESULT timeEndPeriod(
UINT uPeriod                // Menor resolução do timer em milisegundos.
                            // Deve ser o mesmo da função timeBeginPeriod()
);
```

Retorno:

Status	Interpretação
TIMERR_NOERROR	Sucesso
TIMERR_NOCANDO	Falha . Resolução não pode ser atingida.

69

Timer Multimídia

```
MMRESULT timeSetEvent(
UINT uDelay,            // Atraso para ocorrência do Evento em ms. Deve estar dentro da faixa de valores
                        máximo e mínimo suportados pelo timer.
UINT uResolution,       // Resolução do timer em ms. Este valor não deve ser muito pequeno, porque
                        acarreta em overhead. O valor 0 assegura a máxima resolução possível.
LPTIMECALLBACK lpTimeProc,  // Função callback a ser chamada uma vez ou periodicamente, se fuEvent tiver o
                            flag TIME_CALLBACK_FUNCTION ativado, ou handle de um evento se
                            fuEvent tiver os flags TIME_CALLBACK_EVENT_SET ou
                            TIME_CALLBACK_EVENT_PULSE ativados.
DWORD dwUser,           // Dados para função callback
UINT fuEvent            // Tipo do Evento
                            TIME_ONESHOT – Evento ocorre uma vez após uDelay ms.
                            TIME_PERIODIC – Evento ocorre uma vez após uDelay ms e depois
                            periodicamente a cada ms.
                        // Flag determinando se lpTimeProc é uma função callback ou handle de um
                            evento:
TIME_CALLBACK_FUNCTION: Quando o tempo expira a função callback é
chamada (default)
TIME_CALLBACK_EVENT_SET: Quando o tempo expira, um evento é ativado
TIME_CALLBACK_EVENT_PULSE:  Quando o tempo expira, um evento é
pulsado
);
```

70

Timer Multimídia

```
MMRESULT timeKillEvent(
UINT uTimerID
);
```

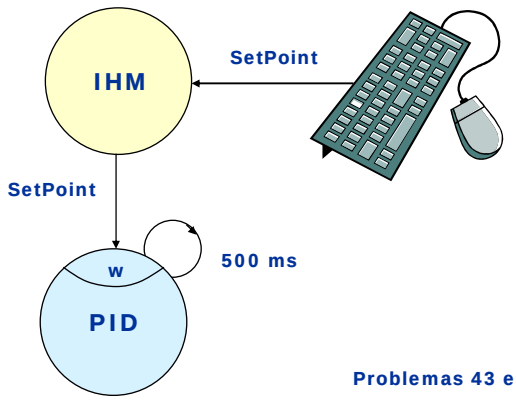
// Timer event a ser cancelado.

Retorno:

Status	Interpretação
TIMERR_NOERROR	Sucesso
MMSYSERR_INVALPARAM	Falha . Evento não existe.

71

Problema



Problemas 43 e 44

72

Timers Multimídia

```
#define     ESC                     0x1B
#define     CR                      0x0D
HANDLE hEvent;                              // Handle para Evento
HANDLE hEscEvent;                           // Handle para Evento Aborta
DWORD WINAPI PidControlFunc(LPVOID);        // declaração da função
void CALLBACK Pid(UINT, UINT, DWORD, DWORD, DWORD); // PID
DOUBLE SetPoint= 0.0;
DOUBLE dInput;

int main()
{
    HANDLE hThread;
    DWORD dwThreadId;
    DWORD dwExitCode = 0;
    BOOL  bStatus;

    hEvent = CreateEvent(NULL, FALSE, FALSE,"Evento"); // apenas uma thread é acordada a cada pulso
    CheckForError(hEvent);
    hEscEvent  = CreateEvent(NULL, TRUE, FALSE,"EscEvento"); // todas as threads são acordadas
    CheckForError(hEscEvent);
    // Cria controlador PID
    hThread = (HANDLE) _beginthreadex(NULL,   0,
            (CAST_FUNCTION)PidControlFunc,   (LPVOID)0, 0, (CAST_LPDWORD)&dwThreadId);
    CheckForError(hThread);
```


Timers Multimídia

```
            do {
            printf("\nEscreva novo valor de SetPoint:\n");
            // le string ou ESC
            bStatus = bGetFloat(&dInput, 9);
            if (bStatus) PulseEvent(hEvent);    // SetPoint mudou
            else PulseEvent(hEscEvent);         // Termina threads
    } while (bStatus);
    // Espera thread terminar
    GetExitCodeThread(hThread, &dwExitCode);
    CloseHandle(hThread);           // apaga referência ao objeto
    CloseHandle(hEvent);
    CloseHandle(hEscEvent);
    return EXIT_SUCCESS;
}  // main


DWORD WINAPI PidControlFunc(LPVOID id)
{
    HANDLE Events[2]= {hEvent, hEscEvent};
    DWORD ret;
    int TipoEvento= 0;
    MMRESULT TimerID;               // Identificador do timer
    TIMECAPS TimeCap;
    UINT MinPeriod;                 // Menor período do timer Multimídia
```

Timers Multimídia

```
// Programa timer
timeGetDevCaps(&TimeCap, sizeof(TIMECAPS));
MinPeriod = TimeCap.wPeriodMin;
printf("Periodo Minimo = %d ms\n", MinPeriod);
timeBeginPeriod(MinPeriod); // Define granularidade
TimerID = timeSetEvent(
        2000,                   // período em ms
        0,                      // resolução máxima
        Pid,                    // função callback,
        0,                      // Não passa dado para função callback
        TIME_PERIODIC);
do {  // observe: quando ocorre evento a temporização é perdida !
        ret=WaitForMultipleObjects(2, Events, FALSE, INFINITE);
        TipoEvento = ret - WAIT_OBJECT_0;
        if (TipoEvento == 0){   // Novo valor de Pid
                SetPoint= dInput;
                printf("Novo valor de set point: %6.2f\n", SetPoint);
        }
} while (TipoEvento != 1);      // Esc foi escolhido
// Cancela timer
timeEndPeriod(MinPeriod);
timeKillEvent(TimerID);
```

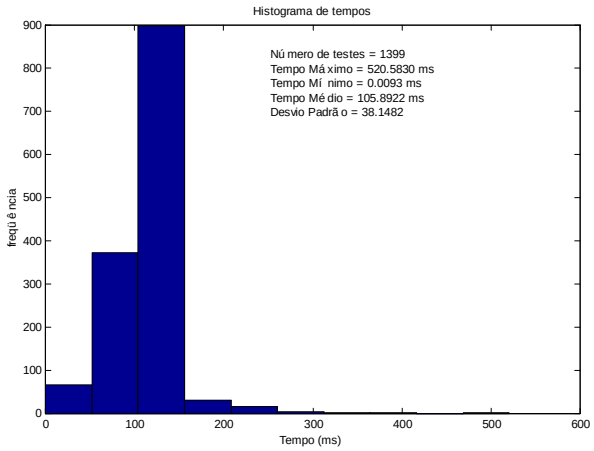

Timers Multimídia

```
    printf("Thread Pid terminando...\n");
    printf("\nAcione uma tecla para terminar\n");
    _getch();           // Pare aqui, caso não esteja executando no ambiente MDS
    _endthreadex((DWORD) 0);
    return 0;
} // WaitEventFunc


void CALLBACK Pid(UINT nTimerID, UINT wMsg, DWORD dwUser, DWORD dw1, DWORD dw2)
{   // Algoritmo Pid;
    printf("Pid rodando... SP= %6.2f\n", SetPoint);
};  // Pid
```

Monitoração de Performance

- Como saber se a performance de uma determinada diretiva se temporização atende a uma aplicação específica ?

Prioridade da Thread Lowest , Temporização desejada 100 ms



Monitoração de Performance

```
typedef struct {
    unsigned long low;
    unsigned long high;
} time_stamp;

#define CPUID  __asm __emit 0fh  __asm  __emit 0a2h
#define RDTSC  __asm __emit 0fh  __asm  __emit 031h

VOID GetTimeStampCounter(time_stamp &time)
{
    __asm {
          pushad
          CPUID              // garante serialização de instruções no Pentium II,...
          RDTSC              // Le Pentium Cycle counter
          mov ebx, time      // ebx é o endereço de time
          mov [ebx].low, eax // contador low
          mov [ebx].high,edx // contador high
          popad
    } // asm
} // GetTimeStampCounter
```

Monitoração de Performance

```
DOUBLE GetTimeInterval(time_stamp init_timer, time_stamp last_timer, unsigned
    MHz_frequency)
// Retorna intervalo de tempo em microsegundos
// MHz_frequency é a frequencia do clock em MHz
{
    DOUBLE          interval;           // intervalo de tempo em micro segundos
    __int64 cycles1, cycles2;           // número de ciclos de clock do Pentium
    cycles1 = ((unsigned __int64) init_timer.high <<32) | init_timer.low;
    cycles2 = ((unsigned __int64) last_timer.high <<32) | last_timer.low;
    interval = (double) (cycles2 - cycles1) / MHz_frequency;
    //printf("Intervalo = %f milisegundos\n", interval/1000);
    return interval;
} // GetTimeInterval
```

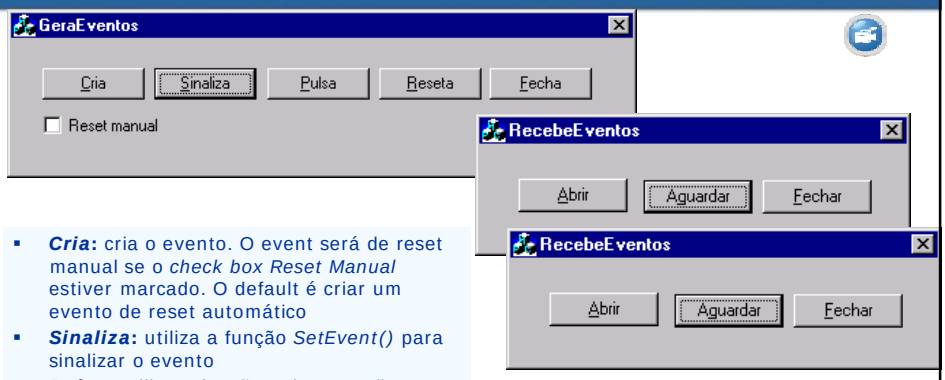
79

Monitoração de Performance

```
#define FREQUENCY           100     // freqüência do seu processador em MHz

// Programa46
int main()
{
    double interval;                // intervalo de tempo em micro segundo
    time_stamp init_timer, last_timer;
    GetTimeStampCounter(init_timer); // passagem por referência
    Sleep(300);
    GetTimeStampCounter(last_timer); // passagem por referência
    interval = GetTimeInterval(init_timer, last_timer, FREQUENCY);
    printf("Intervalo = %f milisegundos\n", interval/1000);
    return EXIT_SUCCESS;
}  // main
```

80

Exemplos MFC – Eventos



- ***Cria*:** cria o evento. O event será de reset manual se o *check box Reset Manual* estiver marcado. O default é criar um evento de reset automático
- ***Sinaliza*:** utiliza a função *SetEvent()* para sinalizar o evento
- ***Pulsa*:** utiliza a função *PulseEvent()* para pulsar o evento
- ***Reseta*:** utiliza a função *ResetEvent()* para resetar o evento
- ***Fecha*:** fecha o evento

- ***Abrir*:** utiliza a função *OpenEvent()* para obter um handle para o evento.
- ***Aguardar*:** utiliza a função *WaitForSingleObject()* para esperar por um evento. Quando o evento for recebido, exibe uma mensagem.
- ***Fechar*:** fecha o handle para o evento.

81

Exemplo MFC - Eventos

```
void CGeraEventosDlg::OnCria()
{
    BOOL bManual;
    bManual = (((CButton *) GetDlgItem (IDC_RESET_MANUAL))->GetState() &
    0x0003) == 1;
    /*
    Criar o evento. A variável bManual indica se ele deverá ser ou não manual. O evento
    deverá ser guardado em hEvento. O nome do evento deverá ser idêntico  ao que será
    utilizado pelo programa EsperaEventos.
    */
    hEvento = CreateEvent (NULL, bManual, FALSE, "evExemploEvento");
    if (hEvento == NULL)
        MensagemErro ("Erro ao criar evento");
}

void CRecebeEventosDlg::OnAbrir()
{
    // abrir o evento
    hEvento= OpenEvent(EVENT_ALL_ACCESS, FALSE, "evExemploEvento");
    if (hEvento == NULL)
        MensagemErro ("Erro ao abrir evento");
}
```

82

Exercícios

1) Comparar as seguintes diretivas de sincronização:

	Algoritmo de Peterson	Critical Sections	Mutex	Semáforos Binários
Vantagens				
Desvantagens				

83

Exercícios

2) Um semáforo apresenta um valor inicial de 2. Depois sofre 6 operações de wait efetuadas por 6 processos diferentes e duas operações de signal. Qual o seu valor final ?

3) Dar o valor verdade para as afirmativas abaixo:

( ) Se várias threads esperam por um evento com reset manual, então todas serão acordadas quando o evento ocorrer.

( ) Se várias threads esperam por um evento com reset automático, então apenas uma será acordada quando o evento ocorrer.

( ) Se uma thread não estiver presa numa função Wait quando a função *PulseEvent()* for chamada, então ela não será capaz de captar este evento

( ) É possível para uma thread perder eventos, quando os eventos são sinalizados com a função *PulseEvent()*.

( ) Um evento com reset manual ficará sinalizado se a função *SetEvent()* for chamada, e só passará ao estado não sinalizado após chamada da função *ResetEvent()*.

( ) Eventos são objetos do kernel e podem ou não ser nomeados

84

Muito Obrigado

Perguntas?

Constantino Seixas Filho

constantino.seixas@task.com.br

Problema: O Jantar dos Selvagens Solução 1

```
Semaphore PoteCheio = 0;
Semaphore PoteVazio = 0;
Semaphore Mutex = 1;
int Missionários = M;

Thread cozinheiro {
     loop {
       Wait(PoteVazio);
       cozinha();
       Missionários = M;
       Signal(PoteCheio);
     }  // end_loop
}

Thread selvagem {
     loop {
       descansa();
       Wait (Mutex);              // Entra na fila do caldeirão
       if (Missionários ==0) {
            Signal(PoteVazio);
            Wait(PoteCheio);
       }
       Missionários--;
       Signal(Mutex);
       come();
     }  // end_loop
}
```

Problema: O Jantar dos Selvagens Solução 2

```
Semaphore PoteCheio = 0;
Semaphore PoteVazio = 1;
int Missionários = 0;

Thread Cozinheiro {
     loop {
        Wait(PoteVazio);
        cozinha();
        Missionários = M;
        Signal(PoteCheio);
     }  // end_loop
}

Thread Selvagem {
     loop {
        descansa();
        Wait(PoteCheio);
        Missionários--;
        if (Missionários == 0)
             Signal(PoteVazio);
        Signal(PoteCheio);
        come();
     }  // end_loop
}
```

87