

```

/*****
/*
/*----- D I N N E R . C -----*/
/* Task      : Variant of the dining philosophers (borrowed from
/*            Dijkstra's Problem of the Philosophers)
/*-----*/
/* Authors    : Michael Tischer and Bruno Jennrich
/* developed on : 08/03/1995
/* last update  : 08/21/1995
/*****
#include <windows.h>
#include "turtle32.h"

#include "resource.h"

/*== Constants and Macros ==*/
#define WM_STARTDINNER WM_USER + 1          /* Dinner begins */
#define MAX_PHILOSOPHERS 20                /* Our table has room for a max. of 20 */

#define THINKING 0                          /* The status of a proper philosopher */
#define HUNGRY 1
#define EATING 2

#define SIZ_SPAGHETTI 100                  /* How big is a portion of spaghetti? */

/* Who is a philosopher's left or right neighbor? -----*/
#define LEFT_NEIGHBOR( i ) ( iNumPhilosophers + ( i - 1 ) ) \
                                % iNumPhilosophers
#define RIGHT_NEIGHBOR( i ) ( ( i + 1 ) % iNumPhilosophers )

/*= Global Variables =====*/
int iNumPhilosophers = 0;                  /* How many accepted the invitation? */
int nState[ MAX_PHILOSOPHERS ];           /* current state of a philosopher
/* (EATING, THINKING, HUNGRY ) */

int nSpaghettis[ MAX_PHILOSOPHERS ];       /* How full is the plate? */

TURTLECONTEXT TC;                          /* TurtleContext for graphical output */

HWND hMainDialog;                          /* Output window for turtle */

HANDLE hThreadPhilosophers[ MAX_PHILOSOPHERS ]; /* Philosopher threads */
HANDLE hSemPhilosophEats[ MAX_PHILOSOPHERS ]; /* Does he have 2 forks? */

HANDLE hMutexStateChange;                  /* Mutex serialized access */

tate array                                  /* to nS

BOOL bStopped;                             /* Output being blocked by main thread? */
HANDLE hMutexStartStop;                     /* Output blocker mutex */

/*****
/* DrawPhilosopher : Draws the given philosophers
/*-----*/
/* Parameter:      pTC - TurtleContext
/*                iNum - Which philosopher?
/*                bDraw - Draw or "blind" passage
/* Return value: none
/*****
void DrawPhilosopher( PTURTLECONTEXT pTC, int iNum, BOOL bDraw )
{
    int HatEdge      = 6000;
    int HatDiagonal  = 8485;                /* SQRT(2) * HatEdge */
    int Plate        = 2500;
    char Buffer[ 10 ];
    int i;

    /* If a thread owns the following mutex, only this thread
    /* can draw, since the subsequent Waits() for a previously
    /* received mutex are granted immediately. (see IDC_STARTSTOP)
    WaitForSingleObject( hMutexStartStop, INFINITE );

    turtleSetPen( pTC, RGB( 0,0,0 ), 1);

```

```

/* Head for location of philosopher -----*/
turtleMoveTo( pTC, 0, 0, FALSE );
turtleSetAngle( pTC, ((360.0f/iNumPhilosophers)*iNum));
turtleForward( pTC, (10000.0f-(Plate+100.0f))/iNumPhilosophers, FALSE );

/* Draw plate -----*/
turtleCircle( pTC, ( float )(Plate/iNumPhilosophers), bDraw );

if( bDraw )
{
    /* Output number of remaining spaghetti noodles -----*/
    wsprintf( Buffer, "%3d ", nSpaghetthis[ iNum ] );
    turtleTextOut( pTC, Buffer );
}

/* When the philosopher eats, two black forks are drawn,          */
/* otherwise they are drawn in the background color of the       */
/* window.                                                         */
if( nState[ iNum ] == EATING )
    turtleSetPen( pTC, RGB( 0,0,0 ), 1);
else
    turtleSetPen( pTC, GetSysColor( COLOR_3DFACE ), 1 );

for( i = 0; i < 2; i++ )
{
    turtlePush( pTC );
    turtleRotate( pTC,i?-90.0f:90.0f);
    turtleForward(pTC,(float)(Plate+5)/iNumPhilosophers,FALSE);
    turtleRotate(pTC,i?180.0f:0.0f);
    turtleRotate(pTC,90.0f);
    turtleForward(pTC,(float)Plate/iNumPhilosophers,FALSE);
    turtleRotate(pTC,180.0f);
    turtleForward(pTC,(float)Plate/iNumPhilosophers, bDraw);

    turtlePush(pTC);
    turtleRotate(pTC,180.0f);
    turtleForward(pTC,(float)(Plate/4)/iNumPhilosophers,FALSE);
    turtleRotate(pTC,180.0f);

    turtlePush(pTC);
    turtleRotate(pTC,160.0f);
    turtleForward( pTC,(float)(Plate/3)/iNumPhilosophers,bDraw);
    turtlePop(pTC);

    turtlePush(pTC);
    turtleRotate(pTC,-160.0f);
    turtleForward(pTC,(float)(Plate/3)/iNumPhilosophers,bDraw);
    turtlePop(pTC);

    turtlePop(pTC);

    turtleForward(pTC,(float)Plate/iNumPhilosophers,bDraw);
    turtlePop(pTC);
}

turtleSetPen(pTC,RGB(0,0,0),1);

turtleForward(pTC,(float)Plate/iNumPhilosophers,FALSE);
turtleForward(pTC,(Plate+100.0f)/iNumPhilosophers,FALSE);

/* Hat */
turtleForward(pTC,
              (float)(HatDiagonal/2)/iNumPhilosophers,
              FALSE );
turtleCircle(pTC,(float)((HatEdge/4)/iNumPhilosophers),bDraw);

switch( nState[ iNum ] )
{
    case EATING:
        turtleTextOut( pTC, "X");
        break;
    case HUNGRY:
        turtleTextOut( pTC, " ! ");
        break;
    case THINKING:
        turtleTextOut( pTC, "?");
}

```

```

    break;
}

turtleForward(pTC,
              (float)(HatDiagonal/2)/iNumPhilosophers,
              FALSE );

turtleRotate( pTC, 135.0f );    /* 180 - 45 */
turtleForward( pTC,(float)HatEdge/iNumPhilosophers,bDraw);
turtleRotate( pTC, 90.0f );
turtleForward( pTC,(float)HatEdge/iNumPhilosophers,bDraw);
turtleRotate( pTC, 90.0f );
turtleForward( pTC,(float)HatEdge/iNumPhilosophers,bDraw);
turtleRotate( pTC, 90.0f );
turtleForward( pTC,(float)HatEdge/iNumPhilosophers,bDraw);

ReleaseMutex( hMutexStartStop );
}

/*****
/* Eating : The second favorite occupation of a philosopher - right
/*          after thinking.
/*-----*/
/* Parameter:    nPhilosoph - Which philosopher?
/* Return value: none
*****/
void Eating( int nPhilosoph )
{
    int iNumSpaghetti;

    srand( GetTickCount() );    /* Initialize random number generator */

    /* 1 - 20 spaghetti at one bite -----*/
    iNumSpaghetti = ( ( 19 * rand() ) / RAND_MAX ) + 1;

    /* Any spaghetti left on the plate? -----*/
    while( nSpaghettis[ nPhilosoph ] && iNumSpaghetti )
    {
        nSpaghettis[ nPhilosoph ]--;
        iNumSpaghetti--;
        DrawPhilosopher( &TC, nPhilosoph, TRUE );    /* Draw */
        Sleep( ( 500 * rand() ) / RAND_MAX );    /* Digesting spaghetti */
    }
}

/*****
/* Test: Checks whether a philosopher is allowed to eat, since his
/*       two neighbors aren't eating.
/*-----*/
/* Parameter:    iPhilosoph - Which philosopher?
/* Return value: none
*****/
void Test( int iPhilosoph )
{
    /* The philosopher can only eat if his two neighbors on the left
    /* and right are not eating
    if( ( nState[ iPhilosoph ] == HUNGRY ) &&
        ( nState[ LEFT_NEIGHBOR( iPhilosoph ) ] != EATING ) &&
        ( nState[ RIGHT_NEIGHBOR( iPhilosoph ) ] != EATING ) )
    {
        /* Test() is called in an hMutexStateChange context,
        /* therefore you can describe nState safely
        nState[ iPhilosoph ] = EATING;

        /* Release semaphore so that thread can access it
        ReleaseSemaphore( hSemPhilosophEats[ iPhilosoph ], 1, NULL );
    }
}

/*****
/* PutDownForks : A philosopher puts down his forks and allows
/*               his neighbors access to their semaphores.
/*-----*/
/* Parameter:    iPhilosoph - Which philosopher?
/* Return value: none
*****/

```

```

void PutDownForks( int iPhilosoph )
{
    WaitForSingleObject( hMutexStateChange,      /* Save access to state */
                        INFINITE );              /* array */

    nState[ iPhilosoph ] = THINKING;             /* Change array entry */

    DrawPhilosopher( &TC, iPhilosoph, TRUE );   /* Draw */

    /* Allow left and right neighbors access to their semaphores */
    Test( LEFT_NEIGHBOR( iPhilosoph ) );
    Test( RIGHT_NEIGHBOR( iPhilosoph ) );

    ReleaseMutex( hMutexStateChange );           /* Release state array */
}

/*****
/* TakeForks      : A philosopher tries to take his fork and one of
/*                  his neighbor's forks.
/*-----*/
/* Parameter:      iPhilosoph - Which philosopher?
/* Return value: none
*****/
void TakeForks( int iPhilosoph )
{
    WaitForSingleObject( hMutexStateChange,      /* State array free? */
                        INFINITE );
    nState[ iPhilosoph ] = HUNGRY;              /* Change state */
    DrawPhilosopher( &TC, iPhilosoph, TRUE );   /* Draw */

    Test( iPhilosoph );                        /* Release custom semaphore */
    ReleaseMutex( hMutexStateChange );          /* Release state array */

    /* May I finally eat? -----*/
    WaitForSingleObject( hSemPhilosophEats[ iPhilosoph ], INFINITE );
}

/*****
/* fnPhilosoph : The mechanical philosopher
/*-----*/
/* Parameter:      nPhilosoph - Which philosopher?
/* Return value: none
*****/
DWORD WINAPI fnPhilosoph( LPVOID nPhilosoph )
{
    srand( GetTickCount() );                  /* Initialize random generator */
    while( nSpaghettis[ ( int ) nPhilosoph ] > 0 )
    {
        /* You may think for up to one second! -----*/
        Sleep( ( ( 1000 * rand() ) / RAND_MAX ) );
        TakeForks( ( int ) nPhilosoph );
        Eating( ( int ) nPhilosoph );
        PutDownForks( ( int ) nPhilosoph );
    }
    return 0L;
}

/*****
/* fnMenuWakeUp : This thread function waits for all philosophers
/*                  to finish eating, in order to release the threads
/*                  and semaphores. Also, the Window menu is
/*                  enabled.
/*-----*/
/* Parameter:      hWnd - Window handle of main window
/* Return value: none
*****/
DWORD WINAPI fnMenuWakeUp( LPVOID hWnd )
{
    int i;
    DWORD dw;

    /* Disable menu bar of main window -----*/
    EnableMenuItem( GetMenu( (HWND) hWnd ), 0,
                    MF_BYPOSITION|MF_DISABLED|MF_GRAYED );
    SetMenu( hWnd, GetMenu( hWnd ) );
    /* Wait until all the philosophers have finished -----*/

```

```

dw = WaitForMultipleObjects( iNumPhilosophers ,
                             hThreadPhilosophers,
                             TRUE,
                             INFINITE );

for( i = 0 ; i < iNumPhilosophers ; i++ )      /* Release handles */
{
    CloseHandle( hSemPhilosophEats[ i ] );      /* Delete semaphores */
    CloseHandle( hThreadPhilosophers[ i ] );    /* Delete threads */
}

CloseHandle( hMutexStateChange );
/* Enable menu -----*/
EnableMenuItem( GetMenu((HWND)hWnd ), 0,MF_BYPOSITION|MF_ENABLED );
SetMenu( hWnd, GetMenu( hWnd ) );
return 0;
}

/*****
/* DialogProc : Window function of the dialog box */
/*-----*/
/* Parameter:      Default dialog box parameters */
/* Return value:   Default dialog box return value */
*****/
BOOL CALLBACK DialogProc( HWND hWnd, UINT uMsg, WPARAM wP, LPARAM lP )
{
    switch( uMsg )
    {
        case WM_INITDIALOG:
            bStopped = FALSE;
            hMutexStartStop = CreateMutex( NULL, FALSE, NULL );
            hMainDialog = hWnd; /* Note window handle in global variables */
            turtleInit( &TC ); /* Initialize turtle */
            turtleSetWindow( &TC, hMainDialog ); /* Set turtle window */
            break;

        case WM_SIZE: /* When resizing, begin redrawing */
            InvalidateRect( hWnd, NULL, TRUE );
            UpdateWindow( hWnd );
            break;

        case WM_PAINT: /* Display all the philosophers in current state */
        {
            int i;

            PAINTSTRUCT ps; /* Remove WM_PAINT from message */
            BeginPaint( hWnd, &ps ); /* loop and remove Clip-Rects */
            EndPaint( hWnd, &ps );

            for( i = 0; i < iNumPhilosophers ; i++ )
                DrawPhilosopher( &TC, i, TRUE );
        }
            break;

        case WM_COMMAND:
            if( ( LOWORD( wP ) >= ID_DINNER_WITH2 ) && /* Invitation to */
                ( LOWORD( wP ) <= ID_DINNER_WITH2 + 18 ) ) /* dinner? */
            {
                /* How many philosophers were invited? -----*/
                iNumPhilosophers = ( LOWORD( wP ) - ID_DINNER_WITH2 ) + 2;
                InvalidateRect( hWnd, NULL, TRUE );
                UpdateWindow( hWnd );

                /* Start dinner -----*/
                PostMessage( hWnd, WM_STARTDINNER, 0, 0 );
            }
            else
                switch( LOWORD( wP ) )
                {
                    case IDC_STARTSTOP: /* Start/Stop button pressed ? */
                    {
                        if( bStopped ) /* Dinner is stopped */
                        {
                            bStopped = FALSE;
                            /* Continue dinner by releasing the mutex -----*/
                            ReleaseMutex( hMutexStartStop );
                            SetWindowText( GetDlgItem( hWnd, IDC_STARTSTOP ), "Pause" );
                        }
                    }
                }
            }
    }
}

```

[illegible]

```

        /* Lower the priority a little. After all, eating philoso- */
        /* phers aren't all that important either.                */
        SetThreadPriority( hThreadPhilosophers[ i ],
                           THREAD_PRIORITY_BELOW_NORMAL );
    }
    /* The following thread intercepts the end of the meal and    */
    /* provides for the deallocation of the philosopher threads   */
    /* and semaphores.                                             */
    CreateThread( NULL,
                 0,
                 fnMenuWakeUp,
                 ( LPVOID )hWnd,
                 0,
                 &dwThreadId );
}
break;
}
return FALSE;
}

/*****
/* WinMain : Main entry point of application                      */
/*-----*/
/* Parameter:      Default WinMain parameters                    */
/* Return value:   Default WinMain return value                  */
*****/
int WINAPI WinMain( HINSTANCE hinstExe,
                   HINSTANCE hinstPrev,
                   LPSTR      lpszCmdLine,
                   int        nCmdShow )
{
    MSG msg;

    /* This application uses a dialog box as the main window -----*/
    hMainDialog = CreateDialog( NULL,
                               MAKEINTRESOURCE( IDD_MAINWINDOW ),
                               NULL,
                               DialogProc );

    /* Application message loop -----*/
    while( GetMessage( &msg, hMainDialog, 0, 0 ) )
    {
        TranslateMessage( &msg );
        DispatchMessage( &msg );
    }
    return 0;
}

```