Vlákna a WinAPI

Podpora

- preemptivní multitasking
- preemptivní multithreading
- podpora kooperativních fibres jádrem
- řečeno jazykem C, vlákno je funkce, která běží paralelně s main
- single procesor
- SMP
- NUMA

Process

- virtuální adresový prostor
- systémové zdroje
- bezpečnostní kontext
    - kdokoliv nemůže provádět cokoliv
- jedinečný identifikátor
- spustitelný kód
- prioritní třída – vlákna mohou mít prioritu pouze v rozsahu prioritní třídy procesu
- má alespoň jedno, primární, vlákno, které může vytvářet vlákna – threads a fibers

Thread

- entita v rámci procesu, kterou plánuje OS
- všechny vlákna sdílí paměťový prostor a zdroje svého procesu
- vlastní handler vyjímek
- priorita
  - OS zajišťuje
    - Inverze priorit
    - Dynamic boost
    - Foreground/Input
  - včetně real-time

- jedinečný identifikátor
- TLS – Thread Local Storage
  - Data, která jsou specifická/lokální pro daný thread
    - __declspec(thread) int number;
  - Možnost, jak si thread může zabezpečit jedinečný přístup ke svým datům
  - Každý proces má k dispozici několik TLS slotů, které mohou být použity jeho thready
  - Možnost využití ke zpracování vyjímek

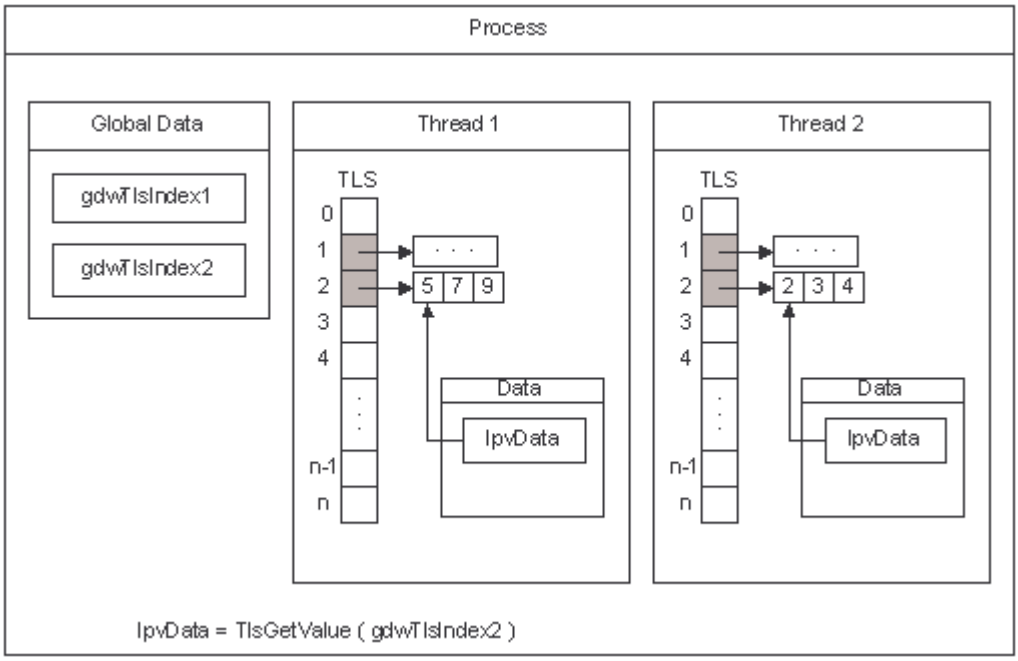
```
program ExplainTLS;

var       global:integer;
threadvar local:integer;  //TLS

procedure X;
var stack:integer;//take local,ale ne TLS

begin
end;
```

- global – jednu proměnnou sdílí všechna vlákna
- local – každé vlákno má svou vlastní, využito TLS
- stack – každé volání rutiny má vlastní proměnnou



http://msdn2.microsoft.com/en-us/library/ms686749.aspx

- POSIX: pthread_key_create

- Kontext
- Zásobník
- Může mít vlastní bezpečnostní kontext – možnost převzetí role někoho jiného (impersonating)
- Lze spustit na dostupných procesorech
  - Get/SetProcessAffinityMask

Job Object

- možnost sloučit několik procesů dohromady a spravovat je jako celek
- operace provedená na jednom objektu ovlivňuje ostatní

Fiber

- běží v kontextu vlákna
- plánuje ho thread procesu
- jeden thread může naplánovat několik fibres
- FLS – Fibre Local Storage
  - Analogie k TLS
  - FLS je asociováno se threadem
    - Lazy init
- Má „malý“ kontext (v porovnání s kontextem threadu)
  - Zásobník, podmnožinu registrů a inicializační data
- Má přístup do TLS threadu v jehož kontextu běží
- Nemá prioritu

Stavy vlákna

- Ke zjednodušení lze použít čtyř stavový model ready, running, waiting, terminated
- Ve skutečnosti lze ale např. s pomocí WMI zjistit následující:
  - ThreadState
    - Initialized — „sjelo z výrobní linky jádra OS“
    - Ready — připraveno ke spuštění na některém z procesorů
    - Running — běží
    - Standby — bude spuštěno, vždy pouze jen jedno vlákno
    - Terminated — RIP
    - Waiting — není připraveno běžet, až bude, bude naplánováno

        - Transition — vlákno čeká na něco jiného než je procesor
        - Unknown — (klientu WMI) Neznámý stav

    - ExecutionState
        - Unknown
        - Other
        - Ready
        - Running
        - Blocked
        - Blocked/Suspended
        - Suspended/Ready

    - Status
        - OK
        - Error
        - Degraded
        - Unknown
        - Pred Fail
        - Starting
        - Stopping
        - Service

    - ThreadWaitReason
        - 21 možností
        - Executive, FreePage, PageOut, PoolAllocation

Funkce řízení běhu vlákna

- CreateThread
- CreateRemoteThread
  - V adresovém prostoru jiného procesu
- Suspend, Resume
- Kill, Terminate, Exit
- Sleep
- Get/SetPriority
- SetThreadStackGuarantee
- http://msdn2.microsoft.com/en-us/library/ms684847.aspx

Thread Pool

- množina vláken, která zpracovává asynchronní události pro process
- asociován s
  - frontou úkolů ke zpracování
  - waitable handles
  - časovačem
  - I/O
- farmer-worker model
- výhodné např.
  - pro aplikace pro distribuované vyhledávání v síti
  - místo vytváření a rušení vláken, která běží jen krátkou dobu

User-Mode Scheduling

- light-weight mechanismus jako fibres
  - možnost pro aplikaci, jak si plánovat svoje vlákna nezávisle na systémovém plánovači
    - tj. ušetří se režie potřebná k přepínání mezi uživatelským režimem a režimem jádra
    - tj. jsou efektivnější než ThreadPool, protože ten vyžaduje přepnutí
- na rozdíl od fibres má každý UMS thread svůj kontext
- Doporučeno jen pro aplikace, u kterých se předpokládají vysoké nároky na výpočetní výkon
- Dostupné od Windows 7 a Server 2008 R2
  - Navíc pouze 64-bitové verze
- UMS aplikace
  - Má svůj UMS plánovač, který
    - Vytvoří jeden UMS scheduler thread pro každý procesor, na kterém může běžet UMS thread
  - Vytvoří UMS thready
    - Konverzí z „normálních“ threadů
  - Spravuje frontu threadů, které dokončily činnost v režimu jádra
    - OS posílá oznámení o takových threadech
      - Aby si je UMS aplikace mohla naplánovat podle svého
  - Provádí úklid po UMS threadech
  - UMS thread by neměl předpokládat, kdo ho plánuje
  - APC zamyká kontext UMS threadu a pak ho nelze naplánovat
- Pozor na sdílení systémových zámků mezi UMS work threadem a UMS Schedulerem –např. loader LoadLibrary
  - Worker ho uzamkne a Scheduler se už nevzbudí

Synchronizace

- probíhá pomocí synchronizačních objektů
  - Event
    - Může být buď pulsní, tj. překlopí se do nonsignaled, jakmile je zpracována wait funkcí
    - Nebo zůstane signaled
  - Mutex
  - Semafor
  - Časovač waitable timer
  - Oznámení o změně – Change notification
  - Standardní vstup
  - Job
  - Memory resource notification - zápis do fyzické paměti
  - Proces
  - Thread
  - Lze čekat i na handle souboru, roury, komunikačního zařízení, ale není doporučováno
    - Místo toho se pro I/O používá asynchronní události – OVERLAPPED

- Objekt je buď signaled, nebo nonsignaled
- Čeká se pomocí tzv. wait funkcí
  - Single-Object
    - SignalObjectAndWait
    - WaitForSingleObject/Ex
  - Multiple-Object
    - WaitForMultipleObjects/Ex
    - MsgWaitForMultipleObjects/Ex

- Alertable Wait
  - Čekání pomocí SignalObjectAndWait a *Ex se ukončí i v případě, že systém dokončil I/O operaci, nebo má nastat APC (asynchronous procedure call) pro dané vlákno

- Registered Wait
  - RegisterWaitForSingleObject
  - UnregisterWaitEx
  - Určeno pro thread pool

- Wait funkce se
  - buď vrátí hned
    - testovala se jen podmínka, ale nečekalo se
    - nebo se mělo čekat, ale podmínka je splněna
  - vrátí se po splnění podmínky
  - vypršel timout, lze čekat i do nekonečna

```
Watcher=FindFirstChangeNotification(path,
        TRUE,
          FILE_NOTIFY_CHANGE_FILE_NAME or
          FILE_NOTIFY_CHANGE_DIR_NAME  or
          FILE_NOTIFY_CHANGE_LAST_WRITE);
       //last write to detect file writes

...

//Wait for a change using a thread pool
RegisterWaitForSingleObject(WatcherWait,
Watcher, WatcherCallback,
Self, INFINITE, WT_EXECUTELONGFUNCTION);
```

Synchronizace a I/O

- operace jako ReadFile jsou buď blokující – synchronní
- nebo jsou asynchronní – například zápis do transakční roury přes síť může trvat dlouho a co dělat, než proběhne?

- Vlákno mezitím může něco udělat
  - Konec operace může zjistit buď pomocí alertable wait funkce
  - Nebo se bude periodicky dotazovat pomocí
    - GetOverlappedResult
    - HasOverlappedIoCompleted
  - Případně může celou operaci zrušit
    - CancelIo

APC

- Asynchronous Procedure Call
- Rutina, která se provede v kontextu daného threadu
- Každý thread má svou frontu APC, jednotlivé APC jsou plánovány plánovačem namísto normálního pokračování non-APC kódu vlákna

- Provedou se pouze tehdy, když je thread v alertable state
  - Volání alertable wait funkce, nebo SleepEx
- Jsou dva druhy
  - User APC
  - Kernel APC

- ReadFileEx, SetWaitableTimer a WriteFileEx jsou realizovány pomocí APC
- Způsob, jakým vykonat požadovanou funkci v kontextu požadovaného vlákna
  - QueueUserAPC
  - Nelze vstoupit do alertable state v APC => Stack overflow

Další možnosti synchronizace

- Kritická sekce
  - Enter, TryEnter, Leave

- Podmínkové proměnné
  - Nelze je sdílet mezi procesy
  - Jsou asociované buď s kritickou sekcí, nebo se slim lock
  - Podporují vzbuzení jednoho, nebo všech čekajících vláken
    - Tj. vhodné pro vlastní implementaci bariéry
  - Čekáním na podmínkovou proměnnou se vlákno vzdá zámku kritické sekce
    - při vzbuzení ho znovu získá

- Slim Locks
  - Slim Reader/Writer (SRW) Locks
  - Optimalizované pro případy, kdy kritická sekce, nebo mutex, představují příliš velkou režii
  - U vstupu do kritické sekce chybí informace, co se v ní bude dít
  - U slim locku se to dá poznat podle volané funkce a systém tak optimalizuje

  - Vyplatí se v případech, kdy se chráněná data více čtou než zapisují
  - 2 režimy ve kterých vlákno může přistupovat k datům
    - Sdílený - read-only; AcquireSRWLockShared
    - Exkluzivní – jenom jedno vlákno může zapisovat; AcquireSRWLockExclusive
    - Viz protected type Ady
  - SRW Lock má velikost pointru, takže nemůže obsahovat příliš informací a tedy ani nemůže být přidělen rekurzivně

- Jednorázová inicializace
  - Může být jedna datová struktura, kterou se může pokoušet inicializovat několik vláken, ale smí být inicializována pouze jednou – tj. pouze jedním vláknem
  - Synchronní
    - InitOnceBeginInitialize
    - InitOnceComplete
    - InitOnceExecuteOnce
    - InitOnceCallback
  - Asynchronní
    - INIT_ONCE_ASYNC

- Interlocked Variable Access
  - Atomické operace nad 32-bitovými proměnnými pod 32-bitovým systémem; stejně pro 64-bitů
  - Operace nad proměnnou větší než registr nejsou atomické – např. 64 bitů na 32-bitovém procesoru
  - Increment, Decrement
  - Exchange, ExchangeAdd, CompareExchange
  - And, Or, Xor

- Interlocked Single Linked List
  - Atomické operace nad jednosměrným seznamem

- Fronta časovače
  - Umožňuje naplánovat vyvolání dané callback funkce v zadanou dobu
  - CreateTimerQueue

Rizika synchronizace na víceprocesorových systémech

- uspořádání přístupu do paměti
  - zápis do paměti je cachován pro zvýšení výkonu
  - čtení dat jde z cache
  - procesor provádí optimalizaci přístupu k datům spekulativním načítáním do cache, když předpokládá, že budou žádána
  - paměťové operace mohou být prováděny mimo pořadí pro zvýšení výkonu

  - řešením je paměťová bariéra, podpora ze strany procesoru – jak říci, kdy jsou která data k dispozici
    - acquire – před další instrukcí v zapsaném kódu
    - release – po další instrukci v zapsaném kódu
    - fence – dostupné v době provedení instrukce
    - funkce WinAPI se o to postarají

- http://msdn2.microsoft.com/en-us/library/ms686355.aspx

- Jde o to, jak uspořádat přístup do paměti
  - Aneb, jak se implementuje volatile

Zprávy

- možnost zasílání zpráv do front
- asynchronní doručení z hlediska příjemce
- je zajištěna synchronizace, konzistence, při doručení do fronty zpráv příjemce
- PostMessage – neblokující odeslání zprávy
- SendMessage – odesílající je blokován do té doby, než je zpráva zpracována a vrácena návratová hodnota
    - InSendMessage
    - SendMessageTimeout

- Get/PeekMessage – vyzvednutí/přečtení zprávy z fronty
- WaitMessage – vlákno se naplánuje až po přijetí zprávy
- PostThreadMessage – doručení do fronty zpráv konkrétního threadu

IPC

- lze synchronizovat i ta vlákna, která se nacházejí v různých procesech
- DuplicateHandle
    - duplikuje stávající handle objektu tak, aby ho mohl použít i jiný proces
        - např. OpenSemaphore

    - výměna informací vyžaduje nějakou formu interakce mezi procesy
        - pojmenované objekty
            - roury
            - sdílená paměť – CreateFileMapping
                - lze využít pro blackboard

- lze získat handle pojmenovaného objektu v jiném procesu, když se předem zná jeho jméno
  - takto lze vzájemně synchronizovat i 32 a 64 bitové procesy

- zaslání zprávy
  - WM_COPYDATA se nemusí omezit jen na parametry, ale může poslat i větší paměťový blok

- Dědičnost
  - CreateProcess – při vytváření dalšího procesu lze specifikovat, zda má zdědit platné handles rodičovského procesu
    - Potřebné informace lze sdělit například příkazovou řádkou

- Sdílený segment DLL
  - Jednu DLL může načíst více procesů
  - Vytvoří se nová sekce v DLL
    - Vedle .text, .bss., atd.
  - Podle flagů OS pozná, že pro ni má vytvořit sdílený segment
  - Data v tomto segmentu jsou pak sdílená pro všechny procesy, které danou DLL načtou
    - Pochopitelně bez synchronizace
      - Je třeba dodat

```
//.DEF file
SECTIONS
.MYDATA
 READ WRITE SHARED
```

```
#pragma data_seg(".MYDATA")
int myint = 0;
#pragma data_seg()
#pragma comment(linker, "/SECTION:.MYDATA,RWS")
```

Rendez-Vous s WinAPI

- Je možné implementovat se všemi vymoženostmi
- Za pomoci dual-interfaces, variants, vlastního skriptu a RTTI je možné vytvořit implementaci, která bude ještě věrněji imitovat zápis v Adě
  - kosmetická záležitost

```
type
  PEntryCallDescriptor =
                    ^TEntryCallDescriptor;
  TEntryCallDescriptor = record
    ID:TEntryCallID;
    Event:THandle;
    Arguments:pointer;
  end;


  TRendezVousServer = class(TThread)
  protected
    FServerState:TServerState;
    FSleepingEvent:THandle;//manual reset
    FEntryCalls:TEntryCallsContainer;
    function GetAcceptableEntryCall:
                     PEntryCallDescriptor;
  public
    procedure Execute; override;
  published
    procedure EntryCall1(AParamSet1);
    procedure EntryCall2(AParamSet2);
  end;
```

```
procedure TRendezVousServer.
                    EntryCall1(AParamSet1);
var descriptor:TEntryCallDescriptor;
    event:THandle;

begin
  event:=CreateEvent
                (nil, true, false, nil);
  //not signaled, manual reset event
  descriptor:=
       CreateAndFillEntryCallDescriptor(
        EntryCall1ID, AParamSet1, event);

  //event is not signaled
  FEntryCalls.Add(@descriptor);
     //assume that this method takes care
     //about critical section handling

  //Wakeup the server if necessary
  SetEvent(FSleepingEvent);
  //and wait for the result
  //if the result is already signaled,
  //it is OK as we use manual-reset event
  if WaitForSingleObject(event, INFINITE)
                      <>WAIT_OBJECT_0 then
    raise ERendezVousException.Create(…);
  CloseHandle(event);
end;
```

```
procedure TRendezVousServer.Execute;
var descriptor:PEntryCallDescriptor;

begin
  BuildUpStateTransitions;
    //could be a parsed script, which
    //defines an order and conditions
    //to be met for each particular
    //entry call - GetAcceptableEntryCall
    //picks available entry call based
    //on FServerState

  repeat
    descriptor:=GetAcceptableEntryCall;
    if descriptor<>nil then
      begin
        //perform the action
        ExecuteEntryCall(descriptor);
        //unblock waiting client
        SetEvent(descriptor^.Event);
        //and remove the descriptor from
        //the queue
        RemoveDescriptor(descriptor);
           //assume this handles critical
           //section
      end else
        TakeDefaultAction;
     //Both, Execute* and Take*,
     //procedures may modify and
     //depends on FServerState

  until (FServerState = ssTerminate) or
         Terminated;
end;
```

```
//An application demonstrating the
//farmer-worker model with WinAPI.
//KIV/PPR Example.

//It generates an ascending array of
//numbers and sums them up in threads.


program fwsum;

{$APPTYPE CONSOLE}

uses
  SysUtils,
  Farmer in 'Farmer.pas',
  Abstraction in 'Abstraction.pas',
  Worker in 'Worker.pas',
  waste in 'waste.pas',
  Strings in 'Strings.pas';

function GetDefault
              (ADefault:integer):integer;
  //Reads a number from the standard
  //input. If the read string is not
  //a suitable number, it returns
  //ADefault.

var s:string;
    i:integer;
```

```
begin
  readln(s);
  val(s, result, i);
  if (i<>0) or (result<1) then
result:=ADefault;
end;

var ValuesCount, WorkersCount,
WasteLevel, i:integer;
    Farmer:TFarmer;
    tmp:Double;


begin
  //Ask for the input parameters
  write(Format(rsEnterValuesSize,
               [dfValuesSize]));
  ValuesCount:=GetDefault(dfValuesSize);

  write(Format(rsEnterWorkersCount,
               [dfWorkersCount]));
  WorkersCount:=
              GetDefault(dfWorkersCount);
  write(Format(rsEnterWastingLevel,
                [dfWastingLevel]));
  WasteLevel:=GetDefault(dfWastingLevel);


  //Create the farmer
  writeln(rsCreatingFarmer);
  writeln(rsGeneratingValues);
  Farmer:=TFarmer.Create(ValuesCount);
```

```
  //Create workers and let them work
  writeln(rsEmployingWorkers);
  for i := 1 to WorkersCount do
    TWorker.Create(Farmer, WasteLevel);
      //Each worker will free itself on
      //exit, so we don't have to remeber
      //the pointers to free the memory.

  //Wait, until all workers had done
  //their jobs.
  Farmer.WaitForWorkers;

  //Display the output.
  tmp:=Farmer.Sum; //just to fool the
                   //Format routine
  writeln(Format(rsTotalSum, [tmp]));

  //And release the farmer as well.
  Farmer.Free;

  //For a convenience, wait for the
  //readln, when running from windowed
  //IDE.
  writeln(rsPressEnter);
  readln;
end.
```

```
unit Abstraction;

interface

const
  WorkSegmentation = 100;
    //the size of a job unit for a worker
  dfValuesSize     = 4000;
  dfWorkersCount   = 4;
  dfWastingLevel   = 32;
   //how many excessive
   //instruction-blocks to execute

type
  PValue = ^TValue;
  TValue = integer;
  //whatever you like, with wasting,
  //integer is enough with the wasting

  TAFarmer = class
    //The abstract farmer - methods
    //visible to workers.
  public
    function AskForValues(
                          out Values:PValue;
                          out Limit:PValue):
                 boolean; virtual; abstract;
      //Returns true if the caller has to
      //compute, or false if the caller
      //has to terminate. Values is the
      //first value, ALimit is the last
      //value.
```

```
    procedure ReportSum(ASum:TValue);
                          virtual; abstract;
      //To report a sum of values
      //assigned with AskForValues.

    procedure LogMessage(
                    const AMessage:String);
                          virtual; abstract;
      //Self-explaining:-)

    procedure AddWorkerRef;
                          virtual; abstract;
    procedure ReleaseWorker;
                          virtual; abstract;
      //worker counting aka reference
      //counting
  end;

implementation

end.
```

```
unit Worker;

interface

uses Classes, Abstraction;

type
  TWorker = class(TThread)
  protected
    FFarmer:TAFarmer;
    FWasteLevel:integer;
  public
    constructor Create(AFarmer:TAFarmer;
                       AWasteLevel:integer);
                              reintroduce;

    procedure Execute; override;
  end;

implementation

uses SysUtils, Strings, Waste;

constructor TWorker.Create(
                           AFarmer:TAFarmer;
                       AWasteLevel:integer);
begin
  FFarmer:=AFarmer;
  FWasteLevel:=AWasteLevel;
  inherited Create(False);
  FreeOnTerminate:=true;
   //so we don't have to bother with Free
end;
```

```
procedure TWorker.Execute;
var cont:boolean;
    value, limit:PValue;
    sum:TValue;

    tmp:Double;

begin
  FFarmer.AddWorkerRef;  //Register
                //the worker - AddRef
  try
    repeat
      cont:=FFarmer.AskForValues(value,
                                 limit);
      if cont then
        begin
          //We've got some values to sum.
          sum:=0;

          //Sum the values
          While integer(value)<=
                integer(limit) do
            begin
              sum:=sum+value^;
              inc(value);

              //And waste the CPU
              //a little-bit, so we get
              //ran on all processors,
              //if there's enough
              //of threads spawned.
              WasteCPU(
                    random(FWasteLevel));
            end;
```

```
            //Summed all up => report the
            //intermediate result
            FFarmer.ReportSum(sum);

            //And log the progress.
            tmp:=sum; //just to fool
                      //the Format routine
            FFarmer.LogMessage(
                  Format(rsWorkerProgress,
                         [ThreadID, tmp]));
          end;

      until Terminated or (not cont);
        //as long as we have to compute

    finally
      FFarmer.ReleaseWorker;   //Unregister
                       //the worker - Release
    end;
end;

end.
```

```
unit Farmer;

interface

uses
 Windows, Classes, SyncObjs, Abstraction;

type
  TFarmer = class(TAFarmer)
    //Note that the farmer is expected
    //to run in the primary thread.
  protected
    FValues:PValue;    //First of values,
                       //which are stored
                       //in the memory
    FCommitted:PValue;//A value to be
                       //assigned next to
                       //a worker
    FLimit:PValue;     //Last of the
                       //stored values
    FSum:TValue;       //The so-far known
                       //sum of all values

    FDataGuard,      //Guarding access to
                     //FCommitted and FSum
    FMessageGuard:TCriticalSection;
            //Guarding access to messages

    FWait,       //Manual-reset event
       //signalizing all workers are done
    FMessageAdded:THandle;//Manual-reset
     //event, new message to be displayed
    FWorkerCount:integer; //Number of
       //registered threads (to wait for)
```

```
    FMessages:TStringList;//Messages to
                          //be displayed

    procedure GenerateValues(
                ASize:integer); dynamic;
       //Generates values to be summed up
  public
    constructor Create(ACount:integer);
    destructor Destroy; override;

    function AskForValues(
              out Values:PValue;
              out ALimit:PValue):boolean;
              override;
    procedure ReportSum(ASum:TValue);
                                 override;
    procedure LogMessage(
        const AMessage:String); override;
    procedure AddWorkerRef; override;
    procedure ReleaseWorker; override;

    procedure WaitForWorkers;
    property Sum:TValue read FSum;
  end;

implementation

uses SysUtils, Strings;
```

```
constructor TFarmer.Create(
                              ACount:integer);
begin
  //Allocate the memory
  GetMem(FValues, sizeof(TValue)*ACount);
  FLimit:=PValue(integer(FValues)+
                sizeof(TValue)*(ACount-1));
  FCommitted:=FValues;

  //Init what's necessary
  GenerateValues(ACount);
  FSum:=0;
  FWorkerCount:=0;

  //Create events
  FWait:=CreateEvent(nil, true,
                     false, nil);
  FMessageAdded:=CreateEvent(nil, true,
                             false, nil);

  //critical sections
  FDataGuard:=TCriticalSection.Create;
  FMessageGuard:=TCriticalSection.Create;

  //and the message holder as well
  FMessages:=TStringList.Create;
end;
```

```
destructor TFarmer.Destroy;
begin
  FDataGuard.Free;
  FMessageGuard.Free;

  CloseHandle(FWait);
  CloseHandle(FMessageAdded);

  FreeMem(FValues);
  FMessages.Free;

  inherited;
end;

procedure TFarmer.GenerateValues(
                              ASize:integer);
var i:integer;
   index:PValue;
begin
  index:=FValues;
  for i:=1 to ASize do
    begin
      index^:=i;
      inc(index);
    end;
end;
```

```
function TFarmer.AskForValues(
                out Values:PValue;
                out ALimit:PValue):boolean;
var tmp:integer;
begin
  FDataGuard.Acquire;
  try
    //Are there some values left
    //for the calling worker?
    result:=integer(FCommitted)<=
            integer(FLimit);
    if result then
       begin
          Values:=FCommitted;
          inc(FCommitted,
              WorkSegmentation);
            //FComitted points to next
            //number to be assigned
             //next time

          ALimit:=FCommitted;
          dec(ALimit);
          if integer(ALimit)>
             integer(FLimit) then
             ALimit:=FLimit;
            //ALimit points to the last
            //number of assigned values

          tmp:=integer(FLimit)-
               integer(ALimit);
          LogMessage(
                 Format(rsFarmerProgress,
                        [tmp]));
       end;
```

```
  finally
    FDataGuard.Release;
  end;
end;

procedure TFarmer.ReportSum(ASum:TValue);
begin
  FDataGuard.Acquire;
  try
    FSum:=FSum+ASum;
  finally
    FDataGuard.Release;
  end;
end;

procedure TFarmer.WaitForWorkers;

procedure DumpMessages;
var i:integer;
begin
  //Just write all messages
  //we have buffered
  FMessageGuard.Acquire;
  try
    for i:=0 to FMessages.Count - 1 do
      writeln(FMessages[i]);
    FMessages.Clear;

    ResetEvent(FMessageAdded);
            //one excessive reset only
  finally
    FMessageGuard.Release;
  end;
end;
```

```
const
  hiWait = 0;
  hiMsg  = 1;

var Handles:array[0..1] of THandle;
    res:integer;

begin
  //We are the main thread, and as a such
  //one, we will serialize the output of
  //messages to the standard output

  Handles[hiWait]:=FWait;
  Handles[hiMsg]:=FMessageAdded;

  repeat
    res:=WaitForMultipleObjects(
                Length(Handles), @Handles,
                False, INFINITE);
      //The code gets here as soon as all
      //workers are done, or there is
      //a new message to be displayed.
      //Because of thread switching,
      //several messages may get buffered
      //prior getting here.
    DumpMessages;

  until res=WAIT_OBJECT_0+hiWait;
end;
```

```
procedure TFarmer.AddWorkerRef;
begin
  InterlockedIncrement(FWorkerCount);
end;

procedure TFarmer.ReleaseWorker;
begin
  InterlockedDecrement(FWorkerCount);
  if FWorkerCount<1 then SetEvent(FWait);
end;

procedure TFarmer.LogMessage(
                    const AMessage:String);
begin
  FMessageGuard.Acquire;
  try
    FMessages.Add(AMessage);
    SetEvent(FMessageAdded);
  finally
    FMessageGuard.Release;
  end;
end;

initialization
  Randomize;
end.
```

```
Enter the number of values (default 4000): 600
Enter the number of workers (default 4):
Enter the maximum level of a CPU-time wasting
(default 32):
Creating Farmer...
Generating values...
Employing workers...
Farmer: 2000 bytes to go...
Farmer: 1600 bytes to go...
Farmer: 1200 bytes to go...
Farmer: 800 bytes to go...
Worker 1F4 reported an intermediate sum of 5050
Farmer: 400 bytes to go...
Worker 1F4 reported an intermediate sum of 45050
Farmer: 0 bytes to go...
Worker 1F4 reported an intermediate sum of 55050
Worker 200 reported an intermediate sum of 35050
Worker CC4 reported an intermediate sum of 15050
Worker 210 reported an intermediate sum of 25050
Total sum of generated values: 180300
Press [Enter] to continue...
```