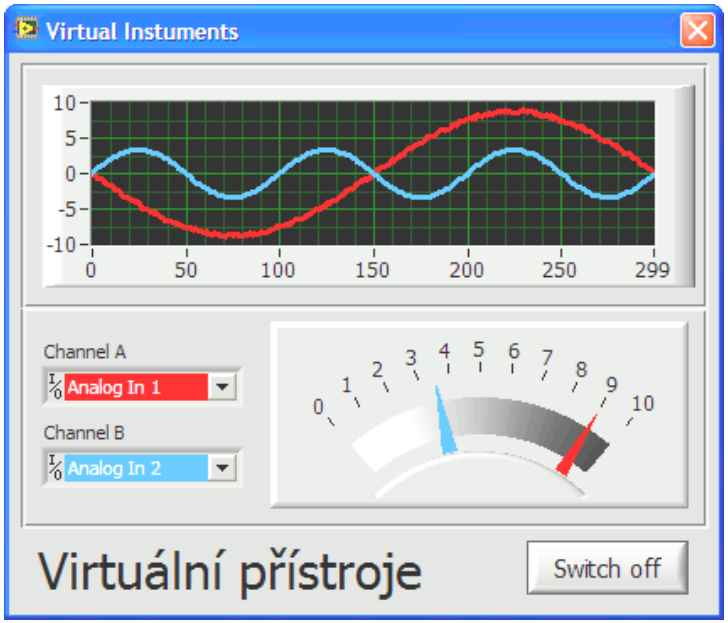
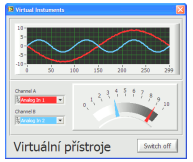
Virtuální přístroje

Operační systémy, vývoj aplikačních programů, speciální techniky programování



Použitá literatura



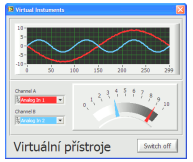
[1] Stallings, W.: Operating Systems. Internals and Design Principles. 4th Edition. Prentice Hall, New Jersey, 2001.

[2] Silberschatz, A. – Galvin, P. B. - Gagne, G.: Operating System Concepts. 6th Edition. John Wiley & Sons, 2003.

[3] Tanenbaum, A.: Modern Operating Systems. Modern Operating Systems. Prentice Hall, New Jersey, 2008.

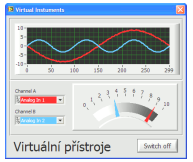
[4] Richter, J.: Windows pro pokročilé a experty. Computer Press, 1997.

Úvod



- Jaké technické prostředky (HW) použijeme? (viz první přednáška)
- Jaké operační systémy (OS) zvolíme?
- Jaké vývojové prostředky pro tvorbu aplikačních programů nasadíme?
- Jaké techniky programování budeme využívat?

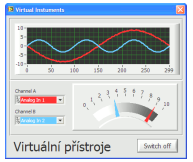
Operační systémy (OS) - 1



Operační systém je software který:

- řídí provádění uživatelských programů,
- funguje jako rozhraní mezi aplikačním programem a fyzickými prostředky (resources) počítače,
- spravuje všechny fyzické prostředky počítače, vytváří lepší, jednodušší, přehlednější prostředí pro efektivní využití počítače.

Operační systémy (OS) - 2



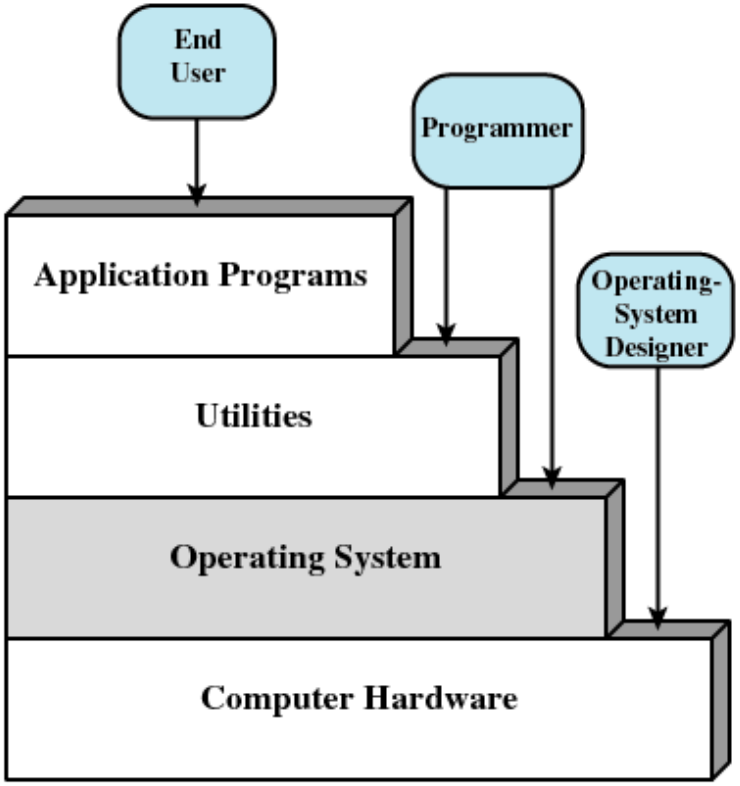
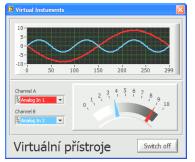


Figure 2.1 Layers and Views of a Computer System

[1] Stallings, W.: Operating Systems. 4th Ed.Prentice Hall, New Jersey, 2001.

Typy OS vhodné pro virtuální přístroje



OS pro osobní počítače

příklady: Windows 95/98/ME, Windows 2000/XP/VISTA/Win2007, Macintosh OS,Linux.

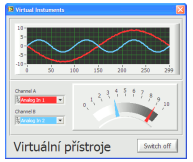
RTOS (Real Time OS)

Speciální aplikace (průmysl, doprava, vědecké experimenty, „vestavěné" aplikace)
Hard/soft RTOS
příklady: VxWorks, RT Linux, RTX, PharLap, … (a většina *Embedded OS*)

Vestavěné OS (Embedded OS)

OS pro komerční i nekomerční aplikace (TV přijímače, mobilní telefony, pračky, digitální fotoaparáty, aj.)
OS pro speciální paměťové karty, PDA, aj.
většinou mají vlastnosti RTOS
příklady: uCLinux, FreeRTOS, Android, QNX, Symbian

OS pro „distribuované“ aplikace



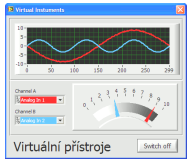
Paralelní systémy

Multiprocesorové systémy pro speciální účely
Clusters

Distribuované systémy

Využívají vhodné sítě LAN, WAN, aj.
Architektury klient-server, *peer-to-peer*

Vývojové prostředky



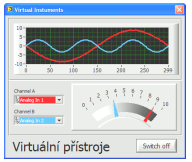
Vývojové systémy na bázi textově orientovaných programovacích jazyků:

- LabWindows/CVI
- MS .NET (resp. Visual C/C++)

Vývojové systémy na bázi grafického programování:

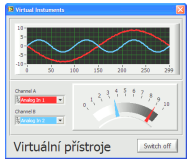
- LabVIEW
- Agilent VEE
- Matlab/Simulink

Procesy a vlákna – základní témata



- Procesy/vlákna
- Výhody a nevýhody použití vláken
- Jaké operační systémy podporují multithreading?
- Synchronizace vláken
- Techniky programování

Procesy a vlákna

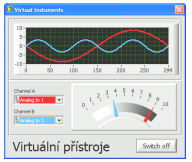


Proces (*process*) - běžící program (=„instance programu“).

Proces vlastní:

- privátní adresový prostor
- systémové prostředky (soubory, dynamicky alokovanou paměť, synchronizační prostředky apod.)
- nejméně jedno vlákno

Procesy a vlákna

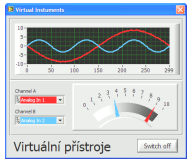


Vlákno (*thread*) – vzniká při inicializaci procesu (primární vlákno). Primární vlákno může vytvářet vlákna sekundární.

Vlákna:

- sdílejí privátní adresový prostor procesu
- mohou přistupovat ke globálním proměnným a systémovým prostředkům procesu
- každé vlákno má vlastní zásobník a vlastní kontext

Multitasking
Multiprocessing / multithreading



[1] Stallings, W.: Operating Systems. 4th Ed.Prentice Hall, New Jersey, 2001.

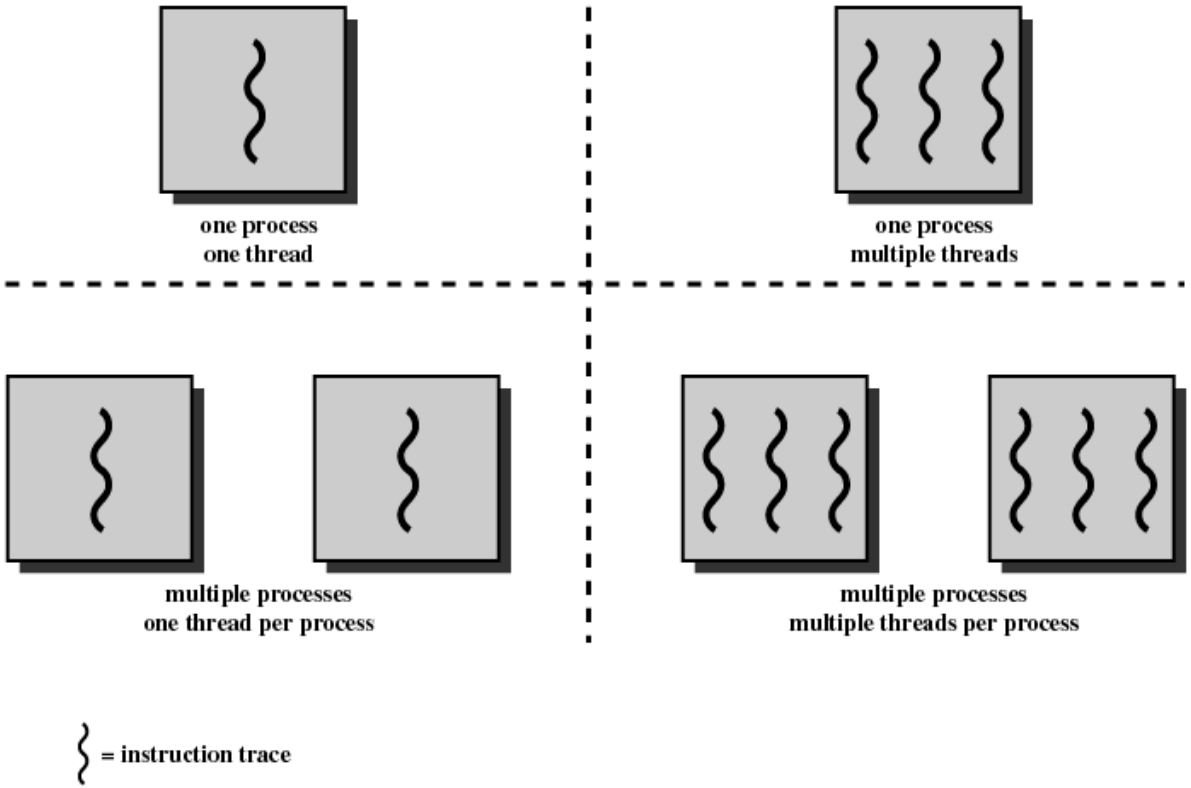
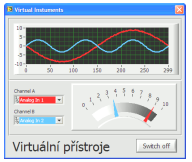


Figure 4.1 Threads and Processes [ANDE97]

Multiprocessing / multithreading



[1] Stallings, W.: Operating Systems. 4th Ed.Prentice Hall, New Jersey, 2001.

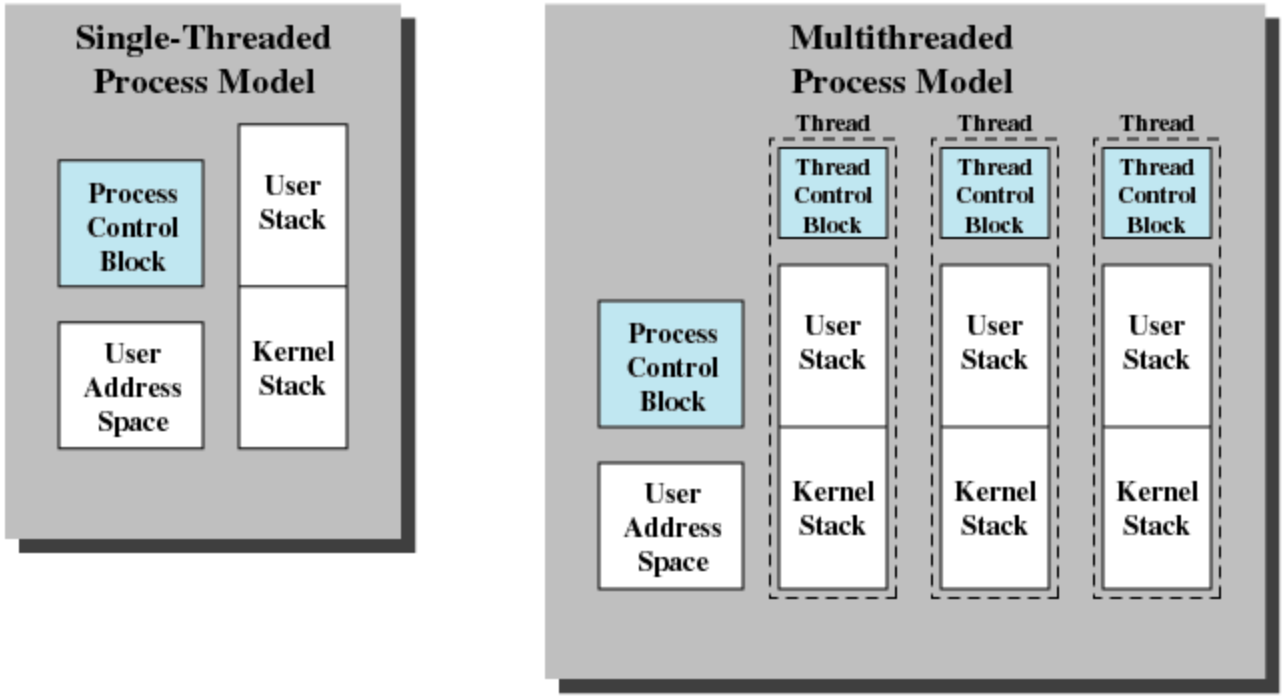
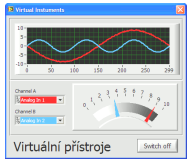
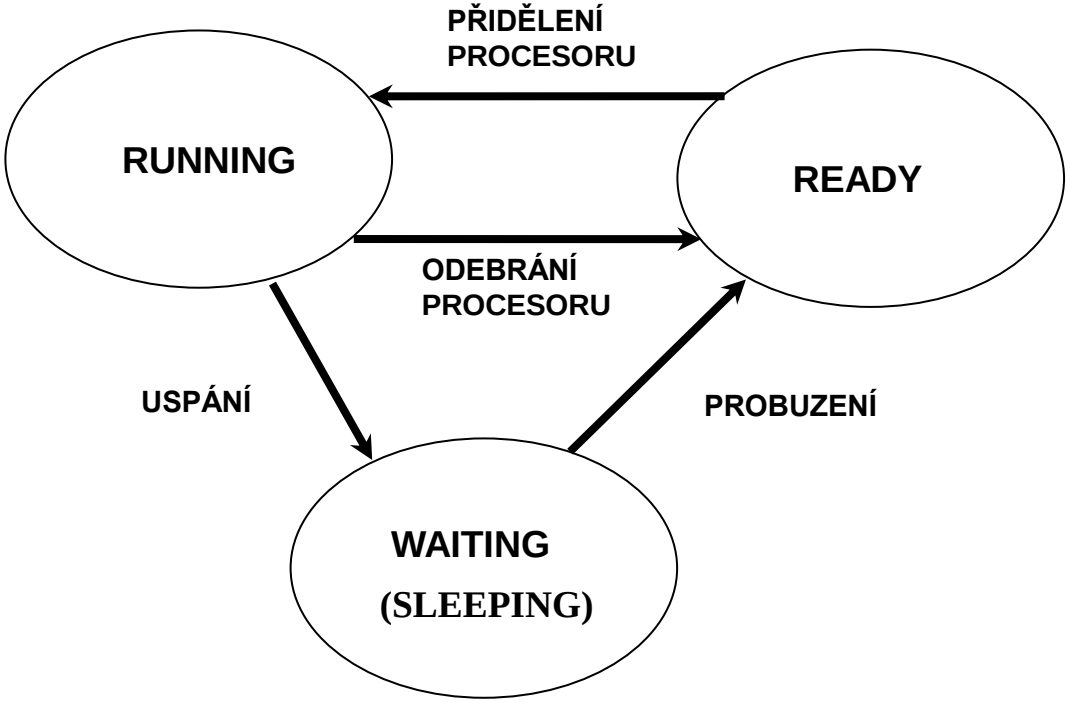


Figure 4.2 Single Threaded and Multithreaded Process Models

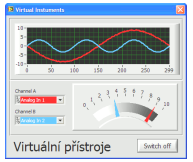
Multitasking



Stavy vlákna (procesu) ve víceúlohovém OS

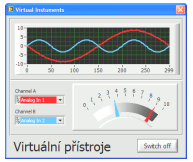


Výhody vláken



- Vícevláknový proces není blokován při zablokování jednoho z vláken (např. při čekání na událost)
- Paměť a systémové prostředky jsou sdíleny (na rozdíl od procesů), zvyšuje se efektivita jejich využití, zjednodušuje se přenos dat mezi vlákny (např. použití globálních proměnných ale POZOR na kritické sekce!!! v programu)
- Přepínání kontextu vláken je výrazně rychlejší než přepínání kontextu u procesů
- Efektivní provádění pomalých I/O operací (I/O operace se mohou překrývat s výpočetními operacemi)
- Výrazné zrychlení programu v případě multiprocesorové architektury (= počítačový systém obsahuje několik procesorů nebo vícejádrové procesory); vlákna běží skutečně paralelně (NE pseudoparalelně!!!)
- Lepší strukturování programu (Ale s rozumem!!! Nepřehánět počet vláken, zvyšuje časovou režii při přepínání kontextu)

Nevýhody vláken



- Obtížnější programování
- Nutnost synchronizace při přístupu ke sdíleným prostředkům (globálním proměnným, periferiím)
- Velmi obtížné až nereálné ladění programu

MULTITASKING / MULTITHREADING
Programování vláken ve Win32

Vytvoření vlákna:

```
HANDLE CreateThread(
    LPSECURITY_ATTRIBUTES  lpThreadAttributes,
    DWORD  dwStackSize,
    LPTHREAD_START_ROUTINE  lpStartAddress,
    LPVOID  lpParameter,
    DWORD  dwCreationFlags, LPDWORD  lpThreadId
    );
```

Programování vláken ve Win32

Ukončení vlákna:

VOID **ExitThread** (DWORD dwExitCode);

Ukončení z jiného vlákna:

BOOL **TerminateThread** (
HANDLE hThread, DWORD dwExitCode);

Programování vláken ve Win32

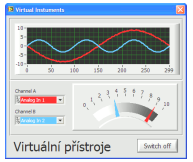
Pozastavení vlákna:

DWORD **SuspendThread**(HANDLE *hThread*);

Opětovné spuštění vlákna:

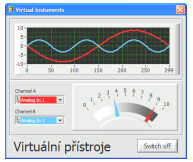
DWORD **ResumeThread**(HANDLE *hThread*);

Přístup ke sdíleným prostředkům



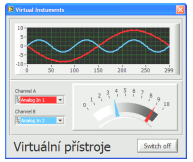
- operace jsou prováděny ve více krocích
- při souběžném přístupu může dojít k jejich přerušení
- vlákno je přeplánováno uprostřed operace
- výsledek závisí na pořadí provádění vláken
- nastává tzv. ***race condition (chyba souběhu)***

Řešení chyby souběhu
Kritická sekce



- dokud není operace dokončena, není možné začít jinou, bez ohledu na počet kroků (atomická změna datové struktury)
- je nutné identifikovat **kritické sekce programu**
- systém zajistí ***vzájemné vyloučení*** *(mutual exlusion) – kód kritické sekce může provádět jen jedno vlákno !!*

Realizace vzájemného vyloučení



Vlákno 1

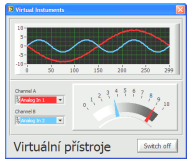
```
p1_locked = TRUE;
turn = P2;
while (p2_locked && turn = P2);

//critical section
p1_locked = FALSE;
```

Vlákno 2

```
p2_locked = TRUE;
turn = P1;
while (p1_locked &&
        turn == P1);
//critical section
p2_locked = FALSE;
```

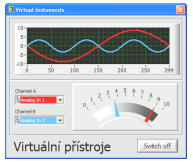
Synchronizace vláken - 1



Cíle:

- zabránit konfliktům mezi vlákny při přístupu ke sdíleným prostředkům (paměti, souborům, periferním zařízením, ap.)
- pozastavení běhu vláken, dokud nejsou splněny specifikované podmínky nebo nenastane určená událost;
- zajistit vykonání určitých programových sekvencí v požadovaném pořadí.

Synchronizace vláken - 2



Optimální mechanismus synchronizace vláken je založen na možnosti převedení vlákna do **stavu čekání** resp. spánku (*waiting*, *sleeping*), kdy mu není přidělován čas procesoru (viz obr. Stavy vlákna …).

V tomto stavu vlákno setrvá tak dlouho, dokud nebudou splněny požadované podmínky (např. ukončení jiného vlákna nebo procesu, nenulový stav semaforu, signalizovaná událost, uplynutí nastaveného časového intervalu apod.).

Jakmile podmínky nastanou, systém vlákno probudí, tzn. přesune je (v závislosti na jeho prioritě) do příslušné fronty aktivních procesů připravených na přidělení procesoru.

Synchronizace pomocí objektů jádra ve Win32

Synchronizace vláken pomocí funkce:

DWORD **WaitForSingleObject** (
HANDLE hObject, DWORD dwTimeout);

DWORD **WaitForMultipleObjects** (
DWORD cObjects, CONST HANDLE *lphObjects,
BOOL fWaitAll, DWORD dwTimeout);

hObject = „manipulátor" (*handle*) synchronizačního objektu (mutex, semafor, událost, soubor, proces, vlákno, …)

Synchr. objekt je v **signalizovaném** nebo **nesignalizovaném stavu!**

Mutex (Mutual Exclusion)

- synchronizační objekt typu „binární semafor“
- signalizován, pokud není vlastněn žádným vláknem
- typicky se používá pro přístup do „kritické sekce“

Vytvoření mutexu:

```
HANDLE CreateMutex (
    LPSECURITY_ATTRIBUTES lpMutexAttributes,
    BOOL bInitialOwner,   LPCTSTR lpName );
```

Otevření mutexu:

```
HANDLE OpenMutex (
    DWORD fdwAccess,     BOOL fInherit,
    LPCTSTR lpszMutexName );
```

Uvolnění mutexu:

```
BOOL ReleaseMutex ( HANDLE hMutex );
```

Příklad vícevláknového programu - 1

```
#include <windows.h>
HANDLE  hThreads[2];
HANDLE  g_hMutex;
DWORD  dwThreadId1, dwThreadId2;

int WinMain(...)       {

// Vytvoření mutexu
g_hMutex = CreateMutex (NULL, FALSE, NULL);
```

Příklad vícevláknového programu - 2

```
// Vytvoření vláken a jejich okamžité spuštění
hThreads[0] = CreateThread (NULL, 0, FirstThread, NULL, 0,
    &dwThreadId1);
hThreads[1] = CreateThread (NULL, 0, SecondThread, NULL, 0,
    &dwThreadId2);

// Čekání na ukončení vláken „FirstThread“ a „SecondThread“;
// primární vlákno je pozastaveno
WaitForMultipleObjects (2,hThreads,TRUE,INFINITE);

// Uzavření manipulátorů obou vláken
CloseHandle (hThreads[0]);
CloseHandle (hThreads[1]);
}
```

Příklad vícevláknového programu - 3

```
DWORD  WINAPI FirstThread (LPVOID lpvThreadParm) {
    BOOL fDone = FALSE;
    DWORD dw;

    while (!fDone) {
                    // čekání na signalizaci mutexu, vlákno je
                       pozastaveno
                    dw = WaitForSingleObject(g_hMutex, INFINITE);

                    if (dw == WAIT_OBJECT_0)  {
                    // mutex byl signalizován, vlákno může pokračovat
                      ...
                    // hlavní programová sekvence vlákna „FirstThread“
                      ...
                    // uvolnění mutexu
                    ReleaseMutex (g_hMutex);
                    }
          }
}
```

Příklad vícevláknového programu - 4

```
DWORD  WINAPI SecondThread (LPVOID lpvThreadParm) {
    BOOL fDone = FALSE;
    DWORD dw;

    while (!fDone) {
                    // čekání na signalizaci mutexu, vlákno je
                       pozastaveno
                    dw = WaitForSingleObject(g_hMutex, INFINITE);

                    if (dw == WAIT_OBJECT_0)  {
                    // mutex byl signalizován, vlákno může pokračovat
                      ...
                    // hlavní programová sekvence vlákna „SecondThread“
                      ...
                    // uvolnění mutexu
                    ReleaseMutex (g_hMutex);
                    }
          }
}
```

Semafory - 1

- synchronizační objekty používané např. ke sledování počtu volných systémových prostředků.
- hodnota semaforu se pohybuje od nuly do specifikované maximální hodnoty.
- semafor je signalizován, je-li jeho hodnota větší než nula, nesignalizován, je-li hodnota nulová.
- při volání funkce WaitForSingleObject (hSemaphore, …) systém zjistí, zda je hodnota semaforu větší než nula a pak ji dekrementuje o jedničku
- k uvolnění semaforu (tzn. inkrementaci jeho hodnoty) se používá volání funkce ReleaseSemaphore()
- ReleaseSemaphore() má jiné chování než ReleaseMutex(). Může být volána libovolným vláknem, které má k objektu semaforu přístup. Hodnota semaforu je inkrementována o hodnotu danou argumentem cReleaseCount.

Semafory - 2

HANDLE CreateSemaphore (

LPSECURITY_ATTRIBUTES lpSemaphoreAttributes,

LONG lInitialCount, LONG lMaximumCount,

LPCTSTR lpName);

HANDLE OpenSemaphore (

DWORD fdwAccess, **BOO**L fInherit,

LPCTSTR lpszSemName);

BOOL ReleaseSemaphore (

HANDLE hSemaphore, **LONG** cReleaseCount,

LPLONG lplPreviousCount);

Příklad

```
HANDLE CreateNewSemaphore(LPSECURITY_ATTRIBUTES lpsa,
      LONG cInitial, LONG cMax, LPTSTR lpszName)
{
  HANDLE hSem;
  hSem = CreateSemaphore(
    lpsa,           //  pointer na „security attributes“
    cInitial,       //  počáteční hodnota
    cMax,           //  maximální hodnota
    lpszName)       //  jméno semaforu

  if (hSem != NULL &&
       GetLastError() == ERROR_ALREADY_EXISTS) {

     // semafor již existuje, funkce vrátí NULL
  CloseHandle(hSem);
  return NULL;
  }
  return hSem;  // vrátí handle nově vytvořeného semaforu
}
```

Události (*Events*)

- Události se zpravidla používají k oznámení, že má být nebo byla ukončena nebo zahájena určitá činnost.
- Rozlišují se dva typy událostí: ručně resetované a automaticky resetované.

Vytvoření události:

```
HANDLE CreateEvent (
    LPSECURITY_ATTRIBUTES  lpEventAttributes,
    BOOL  bManualReset, BOOL  bInitialState,
    LPCTSTR  lpName );
```

Ručně resetované události

- do signalizovaného stavu jsou uvedeny voláním funkce:

  BOOL SetEvent (**HANDLE** hEvent);
- události nejsou prostřednictvím funkcí WaitForSingleObject(), WaitForMultipleObject() převedeny automaticky zpět do nesignalizovaného stavu (jako u mutexu).
- v signalizovaném stavu událost zůstává tak dlouho, dokud ji některé z vláken explicitně nepřepne do nesignalizovaného stavu. K přepnutí (resetování) se používá funkce:

  BOOL **ResetEvent** (HANDLE *hEvent*);
- posloupnost volání SetEvent(), uvolnění všech čekajících vláken a volání ResetEvent() je možné nahradit jedinou funkcí:

  BOOL **PulseEvent** (HANDLE *hEvent*);

Automaticky resetované události

- s automaticky resetovanými událostmi je možné pracovat podobně jako s ručně resetovanými- jiný je mechanismus resetování!!!
- do signalizovaného stavu jsou automaticky resetované události uvedeny voláním funkce SetEvent();
- oproti ručně resetovaným událostem jsou **automaticky resetované události** převedeny systémem bezprostředně po probuzení čekajícího vlákna zpět do nesignalizovaného stavu; v běhu může pokračovat pouze jediné vlákno ze všech, která na událost čekají; ostatní vlákna zůstávají pozastavena;
- použití funkce ResetEvent() je zbytečné.

Multithreading v LabWindows/CVI

Thread Pool

```
CmtScheduleThreadPoolFunction
    (DEFAULT_THREAD_POOL_HANDLE,
    DataAcqThreadFunction, NULL, &functionId);

int CVICALLBACK ThreadFunction (void *functionData);
```

Asynchronous Timer

Multithreading v LabWindows/CVI – příklad použití Thread Pool

```
int CVICALLBACK DataAcqThreadFunction (void *functionData);
int main(int argc, char *argv[])
{
  int panelHandle;
  int functionId;
  if (InitCVIRTE (0, argv, 0) == 0)
  return -1; /* out of memory */
  if ((panelHandle = LoadPanel(0, "DAQDisplay.uir",
          PANEL)) < 0)
  return -1;
  DisplayPanel (panelHandle);
  CmtScheduleThreadPoolFunction
      (DEFAULT_THREAD_POOL_HANDLE,
       DataAcqThreadFunction, NULL, &functionId);
  RunUserInterface ();
```

Příklad - pokračování

```
 DiscardPanel (panelHandle);
 CmtWaitForThreadPoolFunctionCompletion
      (DEFAULT_THREAD_POOL_HANDLE, functionId, 0);
 return 0;
}

int CVICALLBACK DataAcqThreadFunction (void *functionData)
{
 while (!quit) {
  Acquire(. . .);
  Analyze(. . .);
 }
 return 0;
}
```

Ochrana dat

- **Thread Lock**
- **Thread-Safe Variables**
- **Thread-Safe Queue**

Příklad

- Viz materiály NI