

# Sincronizarea threadurilor

---

CURS NR. 6

# Concurența

---

## Execuție **intercalată** și **suprapusă**

- **Intercalarea** în timp a execuției oferă iluzia execuției simultane pe sisteme monoprosesor
  - Introduce o încărcare suplimentară (schimbările de context) dar
  - Oferă eficiență sporită în procesare și în structurarea programului
- În cazul sistemelor multiprosesor este posibilă nu doar intercalarea dar și **suprapunerea** execuțiilor (paralelism real)
- Ambele sunt exemple de execuție concurentă
- Ambele prezintă probleme similare
  - Date de concurență

## **Viteza relativă de execuție** a proceselor nu este predictibilă

- Rezultă din caracteristicile de bază a multiprogramării
- Depinde de activitățile celorlalte procese
- De modul de gestiune a întreruperilor de către SO
- De politica de planificare folosită de SO

Execuția concurentă necontrolată poate duce la **rezultate imprevizibile** și **stări inconsistente** ale resurselor partajate

# Exemplu simplu: incrementarea unui contor

Incrementarea concurentă a unui contor partajat de threaduri

Resursă critică	Thread 1	Thread 2
<code>int x=0;</code>	<code>x++;</code>	<code>x++;</code>

Instrucțiunea de incrementare se descompune în mai multe operații

Rezultatul execuției poate fi

```
registru1 = x
registru1 = registru1 + 1
x = registru1
```

Thread 1	Thread 2	r1	r2	x
<code>r1 = x</code>		0		0
<code>r1 = r1+1</code>		1		0
<code>x = r1</code>		1		1
	<code>r2 = x</code>		1	1
	<code>r2 = r2+1</code>		2	1
	<code>x = r2</code>		2	2

- Sau în urma unei planificări preemptive, rezultatul poate fi:

Thread 1	Thread 2	r1	r2	x
<code>r1 = x</code>		0		0
<code>r1 = r1+1</code>		1		0
	<code>r2 = x</code>	1	0	0
	<code>r2 = r2+1</code>	1	1	0
<code>x = r1</code>		1	1	1
	<code>x = r2</code>	1	1	1

# Concepte fundamentale

---

## Condiție de cursă (race condition)

- Situație în care mai multe procese (threaduri) citesc și scriu o dată partajată și **rezultatul final depinde de ordinea relativă a execuțiilor**

## Resursă critică

- **Resursă accesată** de mai multe procese în mod **concurent**

## Regiune critică (Critical section)

- Constă dintr-un număr de **instrucțiuni** consecutive în care **se accesează resurse critice în mod concurent**

## Excludere mutuală

- În orice moment doar **un singur proces** poate fi în regiunea critică
  - Cât timp un proces este în regiunea critică, nici un alt proces nu poate intra

## Operație atomică

- Acțiune implementată ca o secvență de **instrucțiuni indivizibile**
- Principiu: Totul sau nimic → este garantată execuția secvenței în totalitate sau deloc

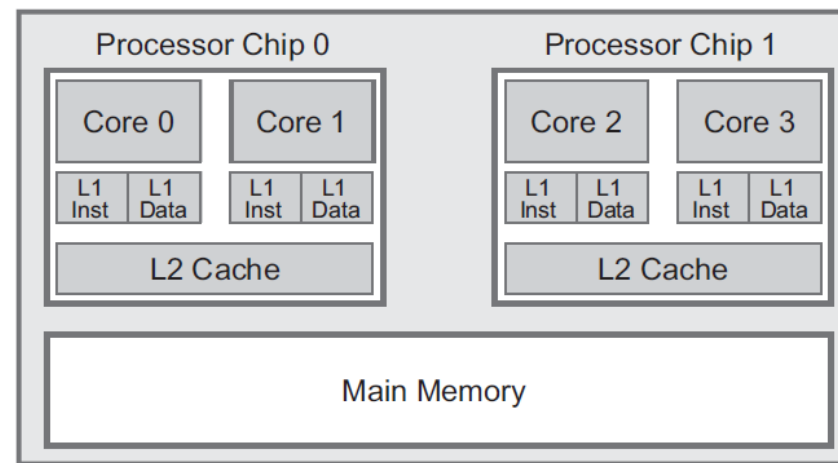
## Necesitatea sincronizării se datorează faptului că operațiile asupra resurselor partajate nu sunt atomice

- Pot fi întrerupte fără finalizarea actualizării resurselor partajate

# Calificatorul de tip `volatile`

Calificatorul de tip `volatile` indică compilatorului că

- Valoarea variabilei `poate fi modificată în mod concurent din afară` – de un alt thread, de sistemul de operare, de hardware
  - Valoarea variabilei să fie întotdeauna citită din locația de memorie și scrisă înapoi în memorie după modificare
    - Fără optimizări de acces (optimizarea implică păstrarea în regiștrii procesorului și citirea ulterioară din registru)
  - Obs. Dacă o structură este declarată volatilă atunci fiecare componentă este volatilă
- 
- Însă scrierea în memorie impusă de `volatile` nu garantează că valoarea este vizibilă tuturor procesoarelor din sistem
    - Valoarea scrisă în memoria cache proprie core-ului



Memory System Architecture

Soluție: Bariere de memorie – furnizate de mecanisme de sincronizare

# Thread-safety

---

## Nu toate funcțiile sunt thread-safe

- Vezi biblioteca C din CRT
  - Motiv: păstrează rezultate intermediare în date stocate static sau global
    - Două threaduri concurente pot accesa simultan biblioteca rezultând în modificare simultană a datelor globale ale bibliotecii
  - Exemple: errno, strtok, tmpfile, asctime, etc.

## Funcție thread-safe

- În gestiunea datelor partajate se garantează execuția în siguranță a instrucțiunilor chiar și în context multithreading
  - când sunt accesate în mod concurent de mai multe threaduri

## Recomandări pentru thread-safety

- Valorile specifice threadului să fie stocate în stiva proprie threadului sau în structuri accesibile doar threadului
  - să nu fie declarate globale sau statice
- Valorile partajate să fie declarate globale sau statice și să fie protejate de mecanisme de sincronizare care crează barieră de memorie

# Mecanisme de sincronizare

---

## Mecanisme de sincronizare în spațiul utilizator

- Familia de funcții **Interlocked**
  - Potrivite pentru operații atomice pe o singură valoare întreagă
    - Operații foarte rapide – performanță mare
- **Secțiuni critice**
  - Potrivite pentru execuția în excludere mutuală a unui bloc de instrucțiuni
    - Seamănă cu mutex-urile dar pot sincroniza doar threadurile din cadrul aceluiași proces
      - NU sunt obiecte kernel, NU sunt vizibile în afara procesului
  - Avantaj simplitate și performanță

## Mecanisme de sincronizare în spațiul kernel – folosind obiecte kernel

- Mutex-uri
- Semafoare
- Evenimente

# Familia de funcții interlocked

Adresele parametrilor trebuie să fie aliniate adecvat în memorie, altfel funcțiile pot eșua !

Soluție simplă pentru incrementarea, decrementarea sau interschimbarea **atomică** a unor valori întregi

- **Adunare atomică** – fără întrerupere - a două valori întregi pe 32 respectiv 64 de biți

```
LONG InterlockedExchangeAdd(
    LONG volatile *Addend,
    LONG Value );
```

Pointer la variabila care se incrementează

```
LONGLONG InterlockedExchangeAdd64(
    LONGLONG volatile *Addend,
    LONGLONG Value );
```

incrementul

- Returnează valoarea inițială de la adresa Addend

- **Incrementare / Decrementare atomică**

```
LONG InterlockedIncrement( LONG volatile *Addend );
LONG InterlockedDecrement( LONG volatile *Addend );
```

```
LONG InterlockedIncrement64( LONG volatile *Addend );
LONG InterlockedDecrement64( LONG volatile *Addend );
```

- Returnează valoarea incrementată / decrementată a variabilei de la adresa Addend

```
// Define a global variable.
long g_x = 0;

DWORD WINAPI ThreadFunc1(PVOID pvParam) {
    g_x++;
    return(0);
}

DWORD WINAPI ThreadFunc2(PVOID pvParam) {
    g_x++;
    return(0);
}
```



```
// Define a global variable.
long g_x = 0;

DWORD WINAPI ThreadFunc1(PVOID pvParam) {
    InterlockedExchangeAdd(&g_x, 1);
    return(0);
}

DWORD WINAPI ThreadFunc2(PVOID pvParam) {
    InterlockedExchangeAdd(&g_x, 1);
    return(0);
}
```

Toate threadurile trebuie să incrementeze variabila folosind doar aceste funcții



# Familia de funcții interlocked

## Schimbarea atomică a valorii unui întreg

```
LONG InterlockedExchange(  
    LONG volatile *Target,  
    LONG          Value );
```

Valori pe 32 de biți

```
LONGLONG InterlockedExchange64(  
    LONGLONG volatile *Target,  
    LONGLONG          Value );
```

Valori aliniate pe 64 de biți

```
PVOID InterlockedExchangePointer(  
    PVOID volatile *Target,  
    PVOID          Value );
```

Valori aliniate pe 32 sau 64 de biți

- Returnează valoarea inițială
- Importante în implementarea spinlock-urilor
  - Ex. Excludere mutuală prin testarea repetată dacă resursa este folosită
    - Threadul setează valoarea la TRUE și while verifică valoarea precedentă
      - Dacă a fost TRUE se rămâne în while verificând mai departe → **așteptare ocupată** ! (în timp ce așteaptă threadul consumă timp de procesare)
      - Dacă a fost FALSE se iese din while, threadul are acces exclusiv la resursă
    - La final setează valoarea la FALSE – indicând ca nu mai are nevoie de resursă

```
// Global variable indicating whether a shared resource is in use or not  
BOOL g_fResourceInUse = FALSE; ...  
  
void Func1() {  
    // Wait to access the resource.  
    while (InterlockedExchange(&g_fResourceInUse, TRUE) == TRUE)  
        Sleep(0);  
    // Access the resource.  
    ...  
    // We no longer need to access the resource.  
    InterlockedExchange(&g_fResourceInUse, FALSE);  
}
```

## Compară și setează în mod atomic (trei variante – ca mai sus)

```
LONG InterlockedCompareExchange(  
    PLONG volatile *pDestination,  
    LONG          lExchange,  
    LONG          lComparand );
```

Compară \*pDestination cu lComparand

- Dacă sunt egale \*pDestination devine Exchange
- Dacă nu, \*pDestination nu se schimbă

Returnează valoarea inițială \*pDestination

Toate acestea  
în mod atomic

# Secțiuni critice: CRITICAL\_SECTION

Secțiunea critică este o regiune de cod care poate fi executată doar în **excludere mutuală**

- Dacă un thread execută secțiunea de cod, toate celelalte threaduri sunt reținute de la intrarea în executarea aceleași secțiuni
- SO poate întrerupe și relua execuția threadului (preemptie) dar garantează excluderea mutuală

CRITICAL\_SECTION este o structură alocată global (sau local, sau dinamic și transmisă threadurilor)

- Toate threadurile din proces pot accesa structura – utilizat pentru **sincronizarea threadurilor din cadrul unui proces**
- CRITICAL\_SECTION nu este partajat între procese – nu poate sincroniza threaduri din procese diferite

## Cod fără sincronizare

```
const int COUNT = 1000;
int g_nSum = 0;

DWORD WINAPI FirstThread(PVOID pvParam) {
    g_nSum = 0;
    for (int n = 1; n <= COUNT; n++) {
        g_nSum += n;
    }
    return(g_nSum);
}

DWORD WINAPI SecondThread(PVOID pvParam) {
    g_nSum = 0;
    for (int n = 1; n <= COUNT; n++) {
        g_nSum += n;
    }
    return(g_nSum);
}
```

Două threaduri  
actualizează concurrent  
valoarea partajată

## Cod cu sincronizare

```
const int COUNT = 10;
int g_nSum = 0;

CRITICAL_SECTION g_cs;

DWORD WINAPI FirstThread(PVOID pvParam) {
    EnterCriticalSection(&g_cs);
    g_nSum = 0;
    for (int n = 1; n <= COUNT; n++) {
        g_nSum += n;
    }
    LeaveCriticalSection(&g_cs);
    return(g_nSum);
}

DWORD WINAPI SecondThread(PVOID pvParam) {
    EnterCriticalSection(&g_cs);
    g_nSum = 0;
    for (int n = 1; n <= COUNT; n++) {
        g_nSum += n;
    }
    LeaveCriticalSection(&g_cs);
    return(g_nSum);
}
```

Toate threadurile (care doresc să acceseze resursa partajată) trebuie să cunoscă adresa structurii CRITICAL\_SECTION

Înainte de utilizare, structura CRITICAL\_SECTION trebuie inițializată !

# Secțiuni critice: CRITICAL\_SECTION

## Declararea secțiunii critice

- Declarare globală (cel mai adesea) sau locală, dinamică, etc. dar adresa structurii să fie cunoscută tuturor threadurilor care vor să folosească secțiunea critică
  - Tipul `CRITICAL_SECTION`

## Inițializarea secțiunii critice

- Structura `CRITICAL_SECTION` trebuie inițializată înainte de utilizare
  - Altfel rezultatele sunt nedefinite

```
void WINAPI InitializeCriticalSection(  
    LPCRITICAL_SECTION lpCriticalSection );
```

## Cerere de intrare în secțiunea critică

- Funcția testează dacă alt thread este înăuntrul secțiunii critice
  - Dacă NU permite threadului intrarea și actualizează structura
  - Dacă DA, threadul este pus în așteptare (nu e busy-waiting)
    - urmând să fie semnalat când secțiunea se eliberează
- Secțiunile critice sunt recursive
  - Dacă threadul din interiorul secțiunii critice cere din nou intrarea, atunci se incrementează contorul de accesări (din structură) și threadul este lăsat să-și continue execuția

```
void WINAPI EnterCriticalSection(  
    LPCRITICAL_SECTION lpCriticalSection );
```

## Eliberarea secțiunii critice

- Funcția decrementează contorul de accesări din partea threadului
  - Dacă contorul devine 0 – se actualizează structura să reflecte că nici un thread nu execută secțiunea critică și revine
  - Dacă contorul > 0 se decrementează contorul și revine – fără să semnaleze eliberarea

```
void WINAPI LeaveCriticalSection(  
    LPCRITICAL_SECTION lpCriticalSection );
```

## Ștergerea secțiunii critice

```
void WINAPI DeleteCriticalSection(  
    LPCRITICAL_SECTION lpCriticalSection );
```

# Secțiuni critice: CRITICAL\_SECTION

```
// Global variable  
CRITICAL_SECTION CriticalSection;
```

→ **Declararea** structurii ca variabilă globală – toate threadurile din proces o pot accesa

```
int main(void) {
```

```
...  
// Initialize the critical section one time only.  
InitializeCriticalSection(&CriticalSection);  
...
```

→ **Inițializarea** componentelor din structură  
- pas esențial înainte de utilizarea secțiunii critice

```
// Release resources used by the critical section object.  
DeleteCriticalSection(&CriticalSection);
```

→ **Ștergerea** structurii când nu se mai accesează resursa partajată pe care o protejează

```
DWORD WINAPI ThreadProc(LPVOID lpParameter) {
```

```
...  
// Request ownership of the critical section.  
EnterCriticalSection(&CriticalSection);
```

**Intrare** în secțiunea critică: **Blocarea secțiunii critice**

- Threadul se blochează dacă alt thread folosește deja secțiunea
- Mai multe threaduri pot fi în așteptare pentru eliberarea secțiunii

```
// Access the shared resource.
```

```
// Release ownership of the critical section.  
LeaveCriticalSection(&CriticalSection);
```

**Părăsirea** secțiunii critice: **Deblocarea secțiunii critice**

- Decrementează contorul de accesări pt thread → dacă 0, se scoate un thread din așteptare (dacă sunt mai multe – nu știm care va intra)

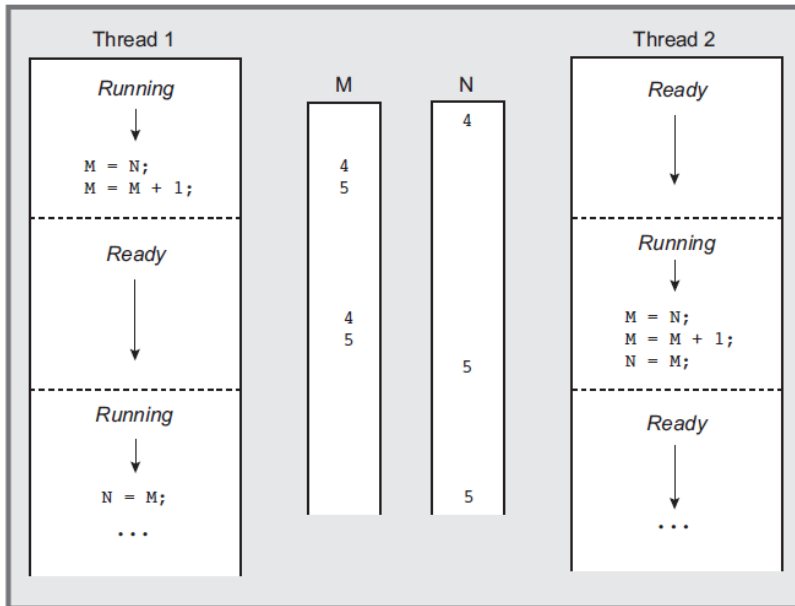
```
...  
return 1;  
}
```

## Alte funcții

- Încercarea blocării secțiunii critice
  - Returnează TRUE dacă a reușit blocarea secțiunii critice, altfel returnează FALSE și revine imediat (threadul nu intră în așteptare)
- Fiecare TryEnterCriticalSection revenit cu succes trebuie să aibă o pereche LeaveCriticalSection

```
BOOL WINAPI TryEnterCriticalSection( LPCRITICAL_SECTION lpCriticalSection );
```

# Secțiuni critice cu spinlock



Unsynchronized Threads Sharing Memory

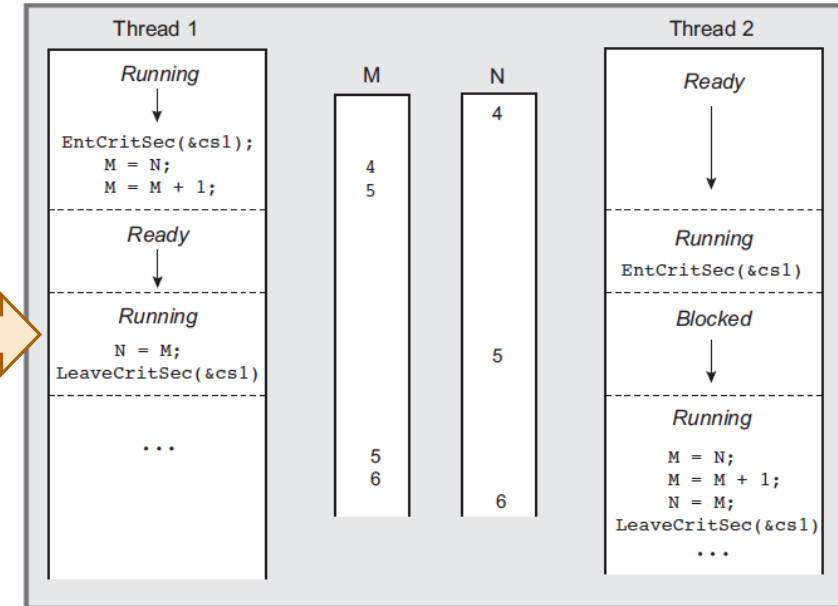
```
#define N_MAX 10000
// Global variable
CRITICAL_SECTION cs;
volatile DWORD N = 0;

DWORD WINAPI ThreadProc(LPVOID lpParameter) {
    DWORD M;
    // Request ownership of the critical section.
    EnterCriticalSection(&cs);
    // Access the shared resource.
    if (N < N_MAX) { M = N; M += 1; N = M; }

    // Release ownership of the critical section.
    LeaveCriticalSection(&cs);
    ...
}

int main(void) {
    // Initialize the critical section one time only.
    if (!InitializeCriticalSectionAndSpinCount(
        &CriticalSection,
        0x00000400)) return;

    ...
    // Create threads executing ThreadFunc
    ...
    // Release resources used by the critical section object.
    DeleteCriticalSection(&cs);
}
```



Synchronized Threads Sharing Memory

## Secțiuni critice care încorporează **spinlock**

- Motivație: Intrarea în așteptare este costisitoare (tranziția din sp utilizator în sp kernel) și threadul care deține secțiunea o poate elibera foarte curând (posibil mai repede decât durata de plasare în așteptare a primului thread)
  - La apelul EnterCriticalSection, dacă secțiunea este deținută de alt thread, înainte de a intra în așteptare se testează în mod repetat (de un număr stabilit de ori) dacă secțiunea nu a fost eliberată și abia după aceea intră în așteptare (dacă e nevoie)

```
BOOL WINAPI InitializeCriticalSectionAndSpinCount(
    LPCRITICAL_SECTION lpCriticalSection,
    DWORD dwSpinCount );
```

# Secțiuni critice

---

O singură structură `CRITICAL_SECTION` **protejează o singură resursă** partajată – sau mai multe resurse partajate care se accesează întotdeauna împreună

- Fiecare acces la resursa partajată trebuie să fie în cadrul unei regiuni protejate de secțiunea critică asociată acelei resurse
  - Altfel resursa poate ajunge într-o stare inconsistentă

Dacă avem mai multe resurse partajate de protejat → avem nevoie de mai multe secțiuni critice

Secțiunea critică **nu este obiect kernel** – se gestionează în spațiul utilizator

- Performanță mai bună decât mutex-urile care oferă aproximativ aceeași funcționalitate dar sunt obiecte kernel

Obiectul `CRITICAL_SECTION` este un mecanism puternic și performant pentru sincronizare, dar

- Nu poate semnala (signal) un alt thread
- Nu are o funcționalitate de time-out
- Soluție: obiectele de sincronizare furnizate de kernel rezolvă aceste neajunsuri

# Problema Producător-Consumator

---

## Exemplu de sincronizare folosind secțiuni critice

### Variantă a problemei clasice Producător-Consumator

- Pe lângă threadul principal mai sunt două threaduri: Producătorul respectiv Consumatorul
- **Producătorul**
  - creează periodic mesaje - un tablou de numere
- **Consumatorul**
  - la cererea utilizatorului – afișează datele curente
    - La cererea utilizatorului – înseamnă cu unele mesaje nu vor fi cerute și vor fi “pierdute”
  - Cerințele pentru afișare
    - Datele cele mai recente care sunt complete
    - Nici o dată să nu fie afișată de mai multe ori
    - Nu se vor afișa date în timp ce producătorul le actualizează
    - Nu se vor afișa date vechi
- Pentru verificarea integrității datelor, Producătorul include un checksum al datelor produse
- Consumatorul validează checksum-ul pentru datele preluate (consumate)
- Threadurile actualizează statistici partajate despre numărul total de mesaje produse, consumate și pierdute

# Problema Producător-Consumator

```
/* Chapter 9. simplePC.c */
/* Maintain two threads, a Producer and a Consumer */
/* The Producer periodically creates checksummed mData buffers, */
/* or "message block" which the Consumer displays when prompted */
/* by the user. The consumer reads the most recent complete */
/* set of mData and validates it before display */

#include "Everything.h"
#include <time.h>
#define DATA_SIZE 256

typedef struct MSG_BLOCK_TAG { /* Message block */
    CRITICAL_SECTION mGuard; /* Guard the message block structure */
    DWORD fReady, fStop;
    /* ready state flag, stop flag */
    volatile DWORD nCons, mSequence; /* Message block mSequence number */
    DWORD nLost;
    time_t mTimestamp;
    DWORD mChecksum; /* Message contents mChecksum */
    DWORD mData[DATA_SIZE]; /* Message Contents */
} MSG_BLOCK;

/* One of the following conditions holds for the message block */
/* 1) !fReady || fStop */
/* nothing is assured about the mData OR */
/* 2) fReady && mData is valid */
/* && mChecksum and mTimestamp are valid */
/* Also, at all times, 0 <= nLost + nCons <= mSequence */

/* Single message block, ready to fill with a new message */
MSG_BLOCK mBlock = { 0, 0, 0, 0, 0 };

DWORD WINAPI Produce (void *);
DWORD WINAPI Consume (void *);
void MessageFill (MSG_BLOCK *);
void MessageDisplay (MSG_BLOCK *);
```

```
int _tmain (int argc, LPTSTR argv[]){
    DWORD status;
    HANDLE hProduce, hConsume;

    /* Initialize the message block CRITICAL SECTION */
    InitializeCriticalSection (&mBlock.mGuard);

    /* Create the two threads */
    hProduce = (HANDLE)_beginthreadex (NULL, 0, Produce, NULL, 0, NULL);
    if (hProduce == NULL)
        ReportError (_T("Cannot create Producer thread"), 1, TRUE);

    hConsume = (HANDLE)_beginthreadex (NULL, 0, Consume, NULL, 0, NULL);
    if (hConsume == NULL)
        ReportError (_T("Cannot create Consumer thread"), 2, TRUE);

    /* Wait for the Producer and Consumer to complete */
    status = WaitForSingleObject (hConsume, INFINITE);
    if (status != WAIT_OBJECT_0)
        ReportError (_T("Failed waiting for Consumer thread"), 3, TRUE);

    status = WaitForSingleObject (hProduce, INFINITE);
    if (status != WAIT_OBJECT_0)
        ReportError (__T("Failed waiting for Producer thread"), 4, TRUE);

    // Release resources used by the critical section object.
    DeleteCriticalSection (&mBlock.mGuard);

    _tprintf (_T("Producer and Consumer threads have terminated\n"));
    _tprintf (_T("Messages Produced: %d, Consumed: %d, Lost: %d.\n"),
        mBlock.mSequence, mBlock.nCons, mBlock.mSequence - mBlock.nCons);

    return 0;
}
```



# Problema Producător-Consumator

```
DWORD WINAPI Produce (void *arg)
/* Producer thread - Create new messages at random intervals */
{
    srand ((DWORD)time(NULL)); /* Seed the random # generator */

    while (!mBlock.fStop) {
        /* Random Delay */
        Sleep(rand()/100);

        /* Get the buffer, fill it */

        EnterCriticalSection (&mBlock.mGuard);

        __try {
            if (!mBlock.fStop) {
                mBlock.fReady = 0;
                MessageFill (&mBlock);
                mBlock.fReady = 1;
                InterlockedIncrement (&mBlock.mSequence);
            }
        }
        __finally { LeaveCriticalSection (&mBlock.mGuard); }
    }
    return 0;
}
```

```
DWORD WINAPI Consume (void *arg)
{
    CHAR command, extra;

    /* Consume the NEXT message when prompted by the user */
    while (!mBlock.fStop) { /*This is the only thread accessing stdin, stdout */
        _tprintf (_T("\n**Enter 'c' for Consume; 's' to stop: "));
        _tscanf (_T("%c%c"), &command, &extra);
        if (command == _T('s')) {
            /* ES not needed here. This is not a read/modify/write.
             * The Producer will see the new value after the Consumer returns */
            mBlock.fStop = 1;
        }
        else if (command == _T('c')) { /* Get a new buffer to Consume */
            EnterCriticalSection (&mBlock.mGuard);

            __try {
                if (mBlock.fReady == 0)
                    _tprintf (_T("No new messages. Try again later\n"));
                else {
                    MessageDisplay (&mBlock);
                    mBlock.nLost = mBlock.mSequence - mBlock.nCons + 1;
                    mBlock.fReady = 0; /* No new messages are ready */
                    InterlockedIncrement(&mBlock.nCons);
                }
            }
            __finally { LeaveCriticalSection (&mBlock.mGuard); }
        }
        else {
            _tprintf (_T("Illegal command. Try again.\n"));
        }
    }
    return 0;
}
```

# Problema Producător-Consumator

```
void MessageFill (MSG_BLOCK *msgBlock)
{
    /* Fill the message buffer, and include mChecksum and mTimestamp*/
    /* This function is called from the Producer thread while it */
    /* owns the message block mutex*/

    DWORD i;

    msgBlock->mChecksum = 0;
    for (i = 0; i < DATA_SIZE; i++) {
        msgBlock->mData[i] = rand();
        msgBlock->mChecksum ^= msgBlock->mData[i];
    }
    msgBlock->mTimestamp = time(NULL);
    return;
}

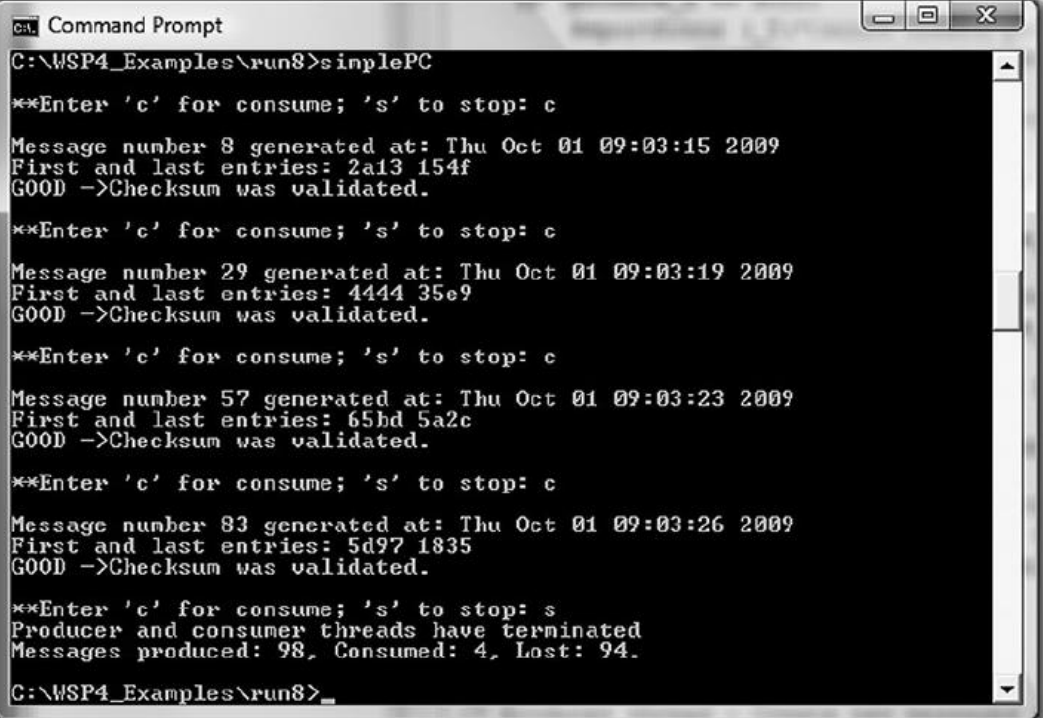
void MessageDisplay (MSG_BLOCK *msgBlock)
{
    /* Display message buffer, mTimestamp, and validate mChecksum*/
    /* This function is called from the Consumer thread while it */
    /* owns the message block mutex*/
    DWORD i, tcheck = 0;

    for (i = 0; i < DATA_SIZE; i++)
        tcheck ^= msgBlock->mData[i];

    _tprintf (_T("\nMessage number %d generated at: %s"),
        msgBlock->mSequence, _tctime (&(msgBlock->mTimestamp)));
    _tprintf (_T("First and last entries: %x %x\n"),
        msgBlock->mData[0], msgBlock->mData[DATA_SIZE-1]);

    if (tcheck == msgBlock->mChecksum)
        _tprintf (_T("GOOD ->mChecksum was validated.\n"));
    else
        _tprintf (_T("BAD ->mChecksum failed. message was corrupted\n"));

    return;
}
```



```
Command Prompt
C:\WSP4_Examples\run8>simplePC

**Enter 'c' for consume; 's' to stop: c
Message number 8 generated at: Thu Oct 01 09:03:15 2009
First and last entries: 2a13 154f
GOOD ->Checksum was validated.

**Enter 'c' for consume; 's' to stop: c
Message number 29 generated at: Thu Oct 01 09:03:19 2009
First and last entries: 4444 35e9
GOOD ->Checksum was validated.

**Enter 'c' for consume; 's' to stop: c
Message number 57 generated at: Thu Oct 01 09:03:23 2009
First and last entries: 65bd 5a2c
GOOD ->Checksum was validated.

**Enter 'c' for consume; 's' to stop: c
Message number 83 generated at: Thu Oct 01 09:03:26 2009
First and last entries: 5d97 1835
GOOD ->Checksum was validated.

**Enter 'c' for consume; 's' to stop: s
Producer and consumer threads have terminated
Messages produced: 98, Consumed: 4, Lost: 94.
C:\WSP4_Examples\run8>
```

simplePC: Periodic Messages, Consumed on Demand

# Problema Producător-Consumator

---

## Comentarii pe marginea codului

- Structura de secțiune critică este parte din obiectul (blocul de mesaj) pe care îl protejează
- Fiecare acces la blocul de mesaj se face în cadrul unei secțiuni critice
  - O excepție: setarea flagului de oprire de către Consumator – la recepționarea comenzii de la utilizator
- Producătorul știe că trebuie să se oprească doar din valoarea flagului de terminare din blocul de mesaj (flag care se poate seta de către Consumator)
- Consumatorul știe că un nou mesaj este produs doar din valoarea flagului ready (flag setat de Producător)
  - În lipsa obiectelor de tip eveniment, threadul nu poate semnaliza celălalt thread (și `TerminateThread` are efecte secundare nedorite) → threadurile trebuie să colaboreze și să nu fie blocate, pentru a putea testa flagul
- Valorile protejate de funcții `Interlocked` sunt declarate volatile
  - `mData` și `mChecksum` nu sunt volatile deoarece sunt modificate doar în interiorul unor secțiuni critice
- Handler-uri de terminare sunt folosite pentru a asigura că secțiunea critică este eliberată
  - Obs. Instrucțiunea `__try` trebuie să fie imediat după `EnterCriticalSection`
- Funcțiile `MessageFill` și `MessageDisplay` sunt apelate doar în cadrul unor secțiuni critice

# Problema cititori - scriitori

Un set de date sunt partajate între mai multe procese

## Două tipuri de threaduri

- Cititori – doar citesc datele (nu le modifică)
- Scriitori – modifică datele

## Reguli

- Mai mulți cititori pot citi simultan datele
- La un moment dat doar un scriitor poate modifica datele
- Dacă un scriitor scrie date, atunci nici un cititor nu are voie să citească

	Reader	Writer
Reader	OK	No
Writer	No	No

Copyright © University of Illinois CS 241 Staff

# Slim Reader-Writer Locks (SRW)

## Similare în scop cu secțiunile critice

- Protejarea unei singure resurse partajate în cazul accesului concurent

## Diferite față de secțiunile critice

- Fac diferențiere între modurile de acces
  - **Acces partajat** – doar în **citire**
    - Threadurile care doresc doar să citească concurent valoarea resursei partajate – numiți **Cititori** (Readers)
      - Accesul în citire poate fi acordat mai multor threaduri simultan – nu apare coruperea datelor
  - **Acces exclusiv** – asigură acces în excludere mutuală pentru **scriere** (și citire)
    - Threadurile care doresc să actualizeze (modifice) valoarea valorii partajate – numiți **Scriitori** (Writers)
      - Nevoia de sincronizare apare când un thread dorește scrierea resursei – necesită acces în excludere mutuală

## Alternativele de sincronizare care oferă acces exclusiv – secțiunile critice, mutex-urile – pot provoca înfometare

- Dacă mai multe threaduri Cititor și Scriitor accesează resursa și threadurile Cititor rulează aproape continuu iar Scriitorii apar rar

## SRW sunt proiectate să fie performante în ceea ce privește viteza și spațiul de memorie ocupat

- Dimensiune similară unui pointer
  - Avantaj: actualizare rapidă a stării lacătului
  - Dezavantaj: puține informații de stare pot fi stocate → SRW nu pot fi obținute în mod recursiv
    - Un thread care deține lacătul în mod partajat – în citire – nu își poate promova modul în deținere exclusivă – pentru scriere

# Slim Reader-Writer Locks (SRW)

## Alocare și inițializare

- Alocarea structurii SRWLOCK
- Inițializare dinamică folosind funcția `InitializeSRWLock`, sau
- Inițializarea statică prin atribuirea constantei **SRWLOCK\_INIT** la variabila de tip structură

```
VOID WINAPI InitializeSRWLock(  
    PSRWLOCK SRWLock );
```

→ Pointer la structura SRWLOCK

## Obținerea lacătului

- În mod exclusiv

```
VOID WINAPI AcquireSRWLockExclusive(  
    PSRWLOCK SRWLock );
```

↓ Pointer la structura SRWLOCK

- În mod partajat

```
VOID WINAPI AcquireSRWLockShared(  
    PSRWLOCK SRWLock );
```

↓ Pointer la structura SRWLOCK

## Eliberarea lacătului

- Obținut în mod exclusiv

```
VOID WINAPI ReleaseSRWLockExclusive(  
    PSRWLOCK SRWLock );
```

↓ Pointer la structura SRWLOCK

- Obținut în mod partajat

```
VOID WINAPI ReleaseSRWLockShared(  
    PSRWLOCK SRWLock );
```

↓ Pointer la structura SRWLOCK

Nu necesită distrugere explicită – sistemul rezolvă automat eliberarea

# Mutex-uri

---

Mutex-ul are funcționalitate similară cu cea a secțiunilor critice

- Avantaje
  - Este un **obiect kernel**
  - Poate sincroniza **threaduri din procese distincte**
  - Poate avea nume – accesibil și threadurilor din procese distincte: permite sincronizare între procese diferite
  - Este **semnalizat la abandonare** de către un thread care se termină
  - Are funcționalitate de **time-out**
- Dezavantaj
  - Performanță mai slabă decât secțiunile critice

## Operațiunile esențiale

- **Blocarea** mutex-ului folosind funcțiile de așteptare pe handle-ul mutexului - **WaitForSingleObject** sau **WaitForMultipleObjects**
- **Deblocarea** / eliberarea mutex-ului: **ReleaseMutex**
- Deținere recursivă
  - Dacă blocarea este recursivă, threadul trebuie să deblocheze mutex-ul de atâtea ori de câte ori l-a blocat

# Mutex-uri

## Crearea unui mutex

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

Atribute de securitate  
Dacă e NULL handle-ul nu poate fi moștenit

Dacă este TRUE - Threadul apelant intră în posesia mutex-ului

Numele obiectului mutex. Dacă e NULL – mutexul este anonim

Creează mutex cu drepturi de acces MUTEX\_ALL\_ACCESS

- Returnează handle-ul mutex-ului nou creat, sau deja existent cu numele dat, respectiv NULL în caz de eșec

## Deschiderea unui mutex existent

```
HANDLE WINAPI OpenMutex(  
    DWORD dwDesiredAccess,  
    BOOL bInheritHandle,  
    LPCTSTR lpName );
```

Drepturi de acces. Doar dreptul **SYNCHRONIZE** este necesar

TRUE pentru moștenibilitate

Numele obiectului mutex – creat cu CreateMutex

- Prin deschiderea mutexului cu nume dintr-un alt proces se poate sincroniza execuțiile threadurilor din procese diferite
  - Create-ul dintr-un proces trebuie să preceadă Open-ul din alt proces
  - Alternativa: folosirea Create în ambele procese – dacă nu contează ordinea

## Eliberarea mutexului

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

Handle-ul mutexului – returnat de CreateMutex sau OpenMutex

Eșuează dacă threadul apelant nu deține mutexul



# Mutex-uri

Un thread poate obține / bloca mutexul prin una din variantele următoare

- Prin creare cu CreateMutex cu parametrul **bInitialOwner** setat la TRUE
- Prin așteptarea cu funcțiile WaitForSingleObject sau WaitForMultipleObjects specificând handle-ul mutexului

Așteptarea după un singur obiect

```
DWORD WINAPI WaitForSingleObject(  
    HANDLE hHandle,  
    DWORD dwMilliseconds );
```

Handle-ul obiectului

Intervalul de time-out în milisecunde

- 0** (threadul nu va intra în așteptare, va reveni imediat)
- INFINITE** (se revine din așteptare doar când obiectul este semnalat)
- Valoare nenulă** (se revine din așteptare doar când timpul specificat se scurge sau obiectul este semnalat)

Așteptarea după mai multe obiecte

```
DWORD WINAPI WaitForMultipleObjects(  
    DWORD nCount,  
    const HANDLE *lpHandles,  
    BOOL bWaitAll,  
    DWORD dwMilliseconds );
```

Numărul de handle-uri la care pointează parametrul următor

Tabloul de handle-uri

Dacă TRUE – se revine din așteptare când toate obiectele sunt semnalate  
Dacă FALSE – se revine dacă oricare din obiecte este semnalat

Valoare returnată indică motivul revenirii

- WAIT\_ABANDONED** – obiectul nu a fost eliberat înainte ca threadul care l-a blocat să se termine
- WAIT\_OBJECT\_0** – obiectul este semnalizat
- WAIT\_TIMEOUT** – a expirat durata de timp și obiectul nu este semnalizat
- WAIT\_FAILED** – funcția a eșuat

```
DWORD dw = WaitForSingleObject(hProcess, 5000);  
switch (dw) {  
    case WAIT_OBJECT_0:  
        // The process terminated.  
        break;  
    case WAIT_TIMEOUT:  
        // The process did not terminate within 5000 milliseconds.  
        break;  
    case WAIT_FAILED:  
        // Bad call to function (invalid handle?)  
        break;  
}
```

# Mutex-uri

## Mutex-uri abandonate

- Mutex care nu este deblocat și threadul care l-a blocat se termină
- Handle-ul mutexului este semnalat
  - WaitForSingleObject și WaitForMultipleObjects returnează **WAIT\_ABANDONED\_0**
- Avantaj față de secțiunile critice

## Interblocare

- Două (sau mai multe) threaduri se blochează în timp ce fiecare așteaptă o resursă deținută de celălalt thread
  - Apare frecvent când două sau mai multe mutex-uri trebuie blocate în același timp
- Una din problemele cele mai frecvente în implementarea sincronizării

Accesul la două resurse logice – fiecare protejat de mutexul propriu → ambele trebuie blocate

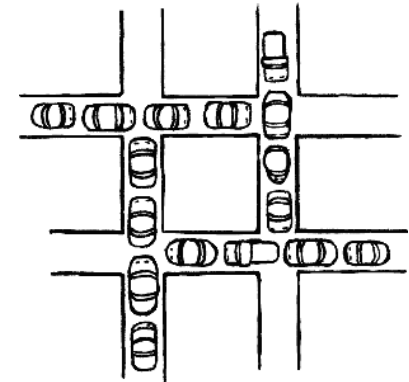
```
DWORD WINAPI ThreadFunc(PVOID pvParam) {  
    EnterCriticalSection(&g_csResource1);  
    EnterCriticalSection(&g_csResource2);  
    // Extract the item from Resource1  
    // Insert the item into Resource2  
    LeaveCriticalSection(&g_csResource2);  
    LeaveCriticalSection(&g_csResource1);  
    return(0);  
}
```

Schimbare  
de context

Deadlock

```
DWORD WINAPI OtherThreadFunc(PVOID pvParam) {  
    EnterCriticalSection(&g_csResource2);  
    EnterCriticalSection(&g_csResource1);  
    // Extract the item from Resource1  
    // Insert the item into Resource2  
    LeaveCriticalSection(&g_csResource2);  
    LeaveCriticalSection(&g_csResource1);  
    return(0);  
}
```

Ordinea de blocare  
este interschimbată



# Exemplu de lucru cu mutex-uri

```
#include <windows.h>
#include <stdio.h>

#define THREADCOUNT 2

HANDLE ghMutex;
DWORD WINAPI WriteToDatabase(LPVOID);

int main(void) {
    HANDLE aThread[THREADCOUNT];
    DWORD ThreadID;
    int i;

    // Create a mutex with no initial owner
    ghMutex = CreateMutex( NULL,           // default security attributes
                          FALSE,          // initially not owned
                          NULL);          // unnamed mutex

    if (ghMutex == NULL) {
        printf("CreateMutex error: %d\n", GetLastError());
        return 1;
    }

    // Create worker threads
    for (i = 0; i < THREADCOUNT; i++) {
        aThread[i] = CreateThread( NULL,    // default security attributes
                                  0,        // default stack size
                                  (LPTHREAD_START_ROUTINE)WriteToDatabase,
                                  NULL,     // no thread function arguments
                                  0,        // default creation flags
                                  &ThreadID); // receive thread identifier

        if (aThread[i] == NULL) {
            printf("CreateThread error: %d\n", GetLastError());
            return 1;
        }
    }

    // Wait for all threads to terminate
    WaitForMultipleObjects(THREADCOUNT, aThread, TRUE, INFINITE);

    // Close thread and mutex handles
    for (i = 0; i < THREADCOUNT; i++) CloseHandle(aThread[i]);

    CloseHandle(ghMutex);

    return 0;
}
```

```
DWORD WINAPI WriteToDatabase(LPVOID lpParam) {
    // lpParam not used in this example
    UNREFERENCED_PARAMETER(lpParam);

    DWORD dwCount = 0, dwWaitResult;

    // Request ownership of mutex.

    while (dwCount < 20) {
        dwWaitResult = WaitForSingleObject( ghMutex,    // handle to mutex
                                           INFINITE); // no time-out interval

        switch (dwWaitResult) {
            // The thread got ownership of the mutex
            case WAIT_OBJECT_0:
                __try {
                    // TODO: Write to the database
                    printf("Thread %d writing to database...\n",
                           GetCurrentThreadId());
                    dwCount++;
                }

                __finally {
                    // Release ownership of the mutex object
                    if (!ReleaseMutex(ghMutex)) {
                        // Handle error.
                    }
                }
                break;

            // The thread got ownership of an abandoned mutex
            // The database is in an indeterminate state
            case WAIT_ABANDONED:
                return FALSE;
        }
    }

    return TRUE;
}
```

# Recomandări

---

Să evităm blocarea definitivă a mutex-ului (în așteptarea obținerii lacătului)

- Prin **setarea duratei de time-out** în funcțiile de așteptare

Threadul **care blochează** un lacăt trebuie să **o și deblocheze**

- În cazul mutexurilor se semnalează starea de abandon – terminarea threadului fără deblocarea mutex-ului
- În cazul secțiunilor critice abandonul nu este semnalat și secțiunea rămâne într-o stare instabilă

Să **testăm valoarea** cu care revine funcția **WaitForSingleObject**

- Nu accesăm resursa dacă se revine datorită expirării duratei de timp

Să nu presupunem că un anumit thread va fi scos din așteptare la eliberarea mutex-ului

- Știm doar că unul din threadurile care așteaptă este trezit din așteptare
- Alegerea threadului depinde de prioritate și politica de planificare

O **resursă partajată** este protejată de un singur lacăt – care poate defini mai multe regiuni critice

- **Asocierea să fie directă** – cel mai bine într-o structură de date

Să nu blocăm lacătul pentru prea multă vreme (să existe **progres**)

- Regiunea critică să cuprindă doar strict codul necesar și blocarea să fie cât se poate de scurtă
- Sincronizarea are un impact semnificativ asupra performanței
  - Regiunile critice sunt executate în excludere mutuală → serializare a execuției (reducerea concurenței)

Să ne asigurăm că fiecare regiune critică are **o singură intrare** (unde se blochează lacătul) și **o singură ieșire** (unde se deblochează lacătul)

- Să evităm ieșirea prematură prin break, return sau exit

# Semafoare

---

Semaforul este un obiect kernel utilizat în sincronizarea accesului la resurse partajate care permit **accesul simultan a unui număr limitat de threaduri** (deci nu numai excludere mutuală)

- Are un contor asociat
  - Semaforul este semnalat dacă contorul  $> 0$

Când un thread **dorește să acceseze** resursa partajată (să intre în regiunea critică) apelează funcțiile de așteptare `WaitForSingleObject` sau `WaitForMultipleObjects`

- Efect: contorul este **decrementat**
  - Dacă contorul este 0 threadul intră într-o coadă de așteptare ca semaforul să fie semnalat

Când un thread **eliberează resursa** (părăsește regiunea critică) apelează funcția de eliberare a semaforului

- Efect: contorul este **incrementat** și semaforul este semnalat
  - Un thread care aștepta eliberarea semaforului este trezit
- Orice thread poate apela incrementarea contorului
  - Față de mutex-uri unde doar threadul care a blocat mutexul îl poate debloca

# Semafoare

## Crearea unui semafor

```
HANDLE WINAPI CreateSemaphore(  
    LPSECURITY_ATTRIBUTES lpSemaphoreAttributes,  
    LONG lInitialCount,  
    LONG lMaximumCount,  
    LPCTSTR lpName );
```

Atribute de securitate  
Dacă e NULL handle-ul nu poate fi moștenit

Contorul inițial – dacă este mai mare ca 0 semaforul este semnalat

Valoarea maximă a contorului

Numele obiectului semafor – dacă NULL atunci este anonim

- Returnează handle-ul semaforului nou creat, sau deja existent cu numele dat, respectiv NULL în caz de eșec

## Deschiderea unui semafor existent

```
HANDLE WINAPI OpenSemaphore(  
    DWORD dwDesiredAccess,  
    BOOL bInheritHandle,  
    LPCTSTR lpName );
```

Drepturi de acces. Doar dreptul **SYNCHRONIZE** este necesar

TRUE pentru moștenibilitate

Numele obiectului semafor – creat cu CreateSemaphore

- Prin deschiderea semaforului cu nume dintr-un alt proces se poate sincroniza execuțiile threadurilor din procese diferite
  - Create-ul dintr-un proces trebuie să preceadă Open-ul din alt proces
  - Alternativa: folosirea Create în ambele procese – dacă nu contează ordinea

## Incrementarea contorului

```
BOOL WINAPI ReleaseSemaphore(  
    HANDLE hSemaphore,  
    LONG lReleaseCount,  
    LPLONG lpPreviousCount );
```

Handle-ul semaforului – returnat de CreateSemaphore sau OpenSemaphore – cu dreptul **SEMAPHORE\_MODIFY\_STATE**

Incrementul

Adresa unde se pune valoarea anterioară a contorului

# Semafoare

Limitare: nu avem operație atomică de așteptare multiplă

- Decrementarea contorului se face doar cu câte o valoare
  - Dacă dorim decrementarea cu 2
    - două apeluri WaitForSingleObject, DAR atunci decrementarea cu 2 nu mai este o operație atomică !
    - Threadul poate fi preemptat între cele două apeluri de așteptare: Poate apare interblocarea

Considerăm codul următor:

Contor: 2

```
/* hSem is a semaphore handle.  
The maximum semaphore count is 2. */  
...  
/* Decrement the semaphore by 2. */  
WaitForSingleObject (hSem, INFINITE);  
WaitForSingleObject (hSem, INFINITE);  
...  
/* Release two semaphore counts. */  
ReleaseSemaphore (hSem, 2, &PrevCount);
```

Thread 1

Thread 2

Contor: 1

Contor: 0

Schimbare  
de context

Deadlock

- Nici unul dintre threaduri nu mai poate efectua al  
doilea Wait – contoul fiind 0 – ambele se blochează!

O soluție: protejarea decrementărilor cu un lacăt sau  
secțiune critică

```
/* Decrement the semaphore by 2. */  
EnterCriticalSection (&csSem);  
WaitForSingleObject (hSem, INFINITE);  
WaitForSingleObject (hSem, INFINITE);  
LeaveCriticalSection (&csSem);  
...  
ReleaseSemaphore (hSem, 2, &PrevCount);
```

Abordarea de așteptare cu WaitForMultipleObject folosind același  
handle de semafor de mai multe ori în tabel nu este validă

# Exemplu: lucrul cu semafoare

```
#include <windows.h>
#include <stdio.h>

#define MAX_SEM_COUNT 10
#define THREADCOUNT 12

HANDLE ghSemaphore;
DWORD WINAPI ThreadProc(LPVOID);

int main(void) {
    HANDLE aThread[THREADCOUNT];
    DWORD ThreadID;
    int i;

    // Create a semaphore with initial and max counts of MAX_SEM_COUNT
    ghSemaphore = CreateSemaphore( NULL,           // default security attributes
                                  MAX_SEM_COUNT,  // initial count
                                  MAX_SEM_COUNT,  // maximum count
                                  NULL);           // unnamed semaphore

    if (ghSemaphore == NULL) {
        printf("CreateSemaphore error: %d\n", GetLastError());
        return 1;
    }

    // Create worker threads
    for (i = 0; i < THREADCOUNT; i++) {
        aThread[i] = CreateThread( NULL,           // default security attributes
                                   0,              // default stack size
                                   (LPTHREAD_START_ROUTINE)ThreadProc,
                                   NULL,           // no thread function arguments
                                   0,              // default creation flags
                                   &ThreadID);    // receive thread identifier

        if (aThread[i] == NULL) {
            printf("CreateThread error: %d\n", GetLastError());
            return 1;
        }
    }

    // Wait for all threads to terminate
    WaitForMultipleObjects(THREADCOUNT, aThread, TRUE, INFINITE);

    // Close thread and semaphore handles
    for (i = 0; i < THREADCOUNT; i++)    CloseHandle(aThread[i]);

    CloseHandle(ghSemaphore);
    return 0;
}
```

```
DWORD WINAPI ThreadProc(LPVOID lpParam) {
    // lpParam not used in this example
    UNREFERENCED_PARAMETER(lpParam);

    DWORD dwWaitResult;
    BOOL bContinue = TRUE;

    while (bContinue) {
        // Try to enter the semaphore gate.

        dwWaitResult = WaitForSingleObject( ghSemaphore,    // handle to semaphore
                                           0L);             // zero-second time-out interval

        switch (dwWaitResult) {
            // The semaphore object was signaled.
            case WAIT_OBJECT_0:
                // TODO: Perform task
                printf("Thread %d: wait succeeded\n", GetCurrentThreadId());
                bContinue = FALSE;

                // Simulate thread spending time on task
                Sleep(5);

                // Release the semaphore when task is finished
                if (!ReleaseSemaphore( ghSemaphore,        // handle to semaphore
                                      1,                  // increase count by one
                                      NULL))               // not interested in previous count
                {
                    printf("ReleaseSemaphore error: %d\n", GetLastError());
                }
                break;

            // The semaphore was nonsignaled, so a time-out occurred.
            case WAIT_TIMEOUT:
                printf("Thread %d: wait timed out\n", GetCurrentThreadId());
                break;
        }
    }
    return TRUE;
}
```



# Evenimente

---

Un eveniment este un obiect kernel pentru sincronizare

- Pot semnala altui thread **îndeplinirea unei condiții**
- Pot trezi din așteptare mai multe threaduri care așteptau după același eveniment

## Categorii

- Cu resetare manuală
  - Poate semnaliza **mai multe threaduri** care așteaptă simultan pentru același eveniment, și **poate fi resetat**
- Cu resetare automată
  - Poate semnaliza un **singur thread** care așteaptă pentru un eveniment și este **resetat automat**

# Evenimente

## Crearea unui eveniment

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

Atribute de securitate  
Dacă e NULL handle-ul nu poate fi moștenit

Dacă este TRUE se creează un eveniment cu resetare manuală,  
Dacă este FALSE se creează un eveniment cu resetare automată

Valoarea inițială: TRUE înseamnă semnalat

Numele obiectului eveniment – dacă NULL atunci este anonim

- Returnează handle-ul evenimentului nou creat, sau deja existent cu numele dat, respectiv NULL în caz de eșec

## Deschiderea unui eveniment existent

```
HANDLE WINAPI OpenEvent(  
    DWORD dwDesiredAccess,  
    BOOL bInheritHandle,  
    LPCTSTR lpName );
```

Drepturi de acces.

TRUE pentru moștenibilitate

Numele obiectului eveniment – creat cu CreateEvent

- Prin deschiderea evenimentului cu nume dintr-un alt proces se poate sincroniza execuțiile threadurilor din procese diferite
  - Create-ul dintr-un proces trebuie să preceadă Open-ul din alt proces
    - Alternativa: folosirea Create în ambele procese – dacă nu contează ordinea

# Evenimente

## Setarea evenimentului – semnalarea evenimentului

```
BOOL WINAPI SetEvent(  
    HANDLE hEvent );
```

Handle-ul evenimentului – returnat de CreateEvent sau OpenEvent  
– cu dreptul **EVENT\_MODIFY\_STATE**

- În cazul evenimentelor cu resetare **manuală** starea evenimentului **rămâne semnalizată** până când se **resetează** manual prin funcția ResetEvent
- În cazul evenimentelor cu resetare **automată** starea evenimentului rămâne **semnalizată** până când **un thread este trezit** din așteptare și sistemul setează starea înapoi la nesemnalizat în mod automat
  - Dacă nici un thread nu așteaptă – starea rămâne semnalizată

## Resetarea evenimentului

```
BOOL WINAPI ResetEvent(  
    HANDLE hEvent );
```

Handle-ul evenimentului – returnat de CreateEvent sau OpenEvent  
– cu dreptul **EVENT\_MODIFY\_STATE**

- Schimbarea stării evenimentului în nesemnalizat – starea va putea fi modificată prin funcțiile SetEvent sau PulseEvent
  - Threadurile care așteaptă după eveniment vor fi blocate în așteptare

## Setarea pe moment a evenimentului

```
BOOL WINAPI PulseEvent(  
    HANDLE hEvent );
```

Handle-ul evenimentului – returnat de CreateEvent sau OpenEvent  
– cu dreptul **EVENT\_MODIFY\_STATE**

- Scoate din așteptare threadurile care așteaptă în acel moment și apoi se resetează (în cazul evenimentelor cu resetare automată scoate un singur thread din așteptare)

# Exemplu: lucrul cu obiecte eveniment

```
#include <windows.h>
#include <stdio.h>

#define THREADCOUNT 4

HANDLE ghWriteEvent;
HANDLE ghThreads[THREADCOUNT];
DWORD WINAPI ThreadProc(LPVOID);

void CreateEventsAndThreads(void) {
    int i;
    DWORD dwThreadId;

    // Create a manual-reset event object. The write thread sets this
    // object to the signaled state when it finishes writing to a
    // shared buffer.

    ghWriteEvent = CreateEvent( NULL,                // default security attributes
                               TRUE,                // manual-reset event
                               FALSE,               // initial state is nonsignaled
                               TEXT("WriteEvent")); // object name

    if (ghWriteEvent == NULL) {
        printf("CreateEvent failed (%d)\n", GetLastError());
        return;
    }

    // Create multiple threads to read from the buffer.
    for (i = 0; i < THREADCOUNT; i++) {
        // TODO: More complex scenarios may require use of a parameter to the thread
        // procedure, such as an event per thread to be used for synchronization.
        ghThreads[i] = CreateThread( NULL,           // default security
                                     0,              // default stack size
                                     ThreadProc,     // name of the thread function
                                     NULL,           // no thread parameters
                                     0,              // default startup flags
                                     &dwThreadId);
    }
}
```

```
        if (ghThreads[i] == NULL) {
            printf("CreateThread failed (%d)\n", GetLastError());
            return;
        }
    }
}
```

```
void WriteToBuffer(VOID) {
    // TODO: Write to the shared buffer.

    printf("Main thread writing to the shared buffer...\n");

    // Set ghWriteEvent to signaled

    if (!SetEvent(ghWriteEvent)) {
        printf("SetEvent failed (%d)\n", GetLastError());
        return;
    }
}

void CloseEvents() {
    // Close all event handles (currently, only one global handle).

    CloseHandle(ghWriteEvent);
}
```

# Exemplu: lucrul cu obiecte eveniment

```
int main(void) {
    DWORD dwWaitResult;

    // TODO: Create the shared buffer
    // Create events and THREADCOUNT threads to read from the buffer
    CreateEventsAndThreads();

    // At this point, the reader threads have started and are most likely waiting for
    // the global event to be signaled. However, it is safe to write to the buffer
    // because the event is a manual-reset event.

    WriteToBuffer();

    printf("Main thread waiting for threads to exit...\n");

    // The handle for each thread is signaled when the thread is terminated.
    dwWaitResult = WaitForMultipleObjects( THREADCOUNT,    // number of handles in array
                                          ghThreads,        // array of thread handles
                                          TRUE,              // wait until all are
                                          signaled            // INFINITE);

    switch (dwWaitResult) {
        // All thread objects were signaled
        case WAIT_OBJECT_0:
            printf("All threads ended, cleaning up for application exit...\n");
            break;

        // An error occurred
        default:
            printf("WaitForMultipleObjects failed (%d)\n", GetLastError());
            return 1;
    }

    // Close the events to clean up
    CloseEvents();

    return 0;
}
```

```
DWORD WINAPI ThreadProc(LPVOID lpParam) {
    // lpParam not used in this example.
    UNREFERENCED_PARAMETER(lpParam);

    DWORD dwWaitResult;

    printf("Thread %d waiting for write event...\n", GetCurrentThreadId());

    dwWaitResult = WaitForSingleObject( ghWriteEvent, // event handle
                                      INFINITE);    // indefinite wait

    switch (dwWaitResult) {
        // Event object was signaled
        case WAIT_OBJECT_0:

            // TODO: Read from the shared buffer

            printf("Thread %d reading from buffer\n", GetCurrentThreadId());
            break;

        // An error occurred
        default:
            printf("Wait error (%d)\n", GetLastError());
            return 0;
    }

    // Now that we are done reading the buffer, we could use another
    // event to signal that this thread is no longer reading. This
    // example simply uses the thread handle for synchronization (the
    // handle is signaled when the thread terminates.)

    printf("Thread %d exiting\n", GetCurrentThreadId());
    return 1;
}
```

# Evenimente

Patru moduri de utilizare ale evenimentelor

	Evenimente cu resetare manuală	Evenimente cu resetare automată
SetEvent	Toate threadurile care sunt în așteptare sunt eliberate <ul style="list-style-type: none"><li>până când se resetează evenimentul manual</li></ul>	Un singur thread este eliberat și apoi se resetează <ul style="list-style-type: none"><li>Unul care așteaptă acum sau primul care va aștepta</li></ul>
PulseEvent	Toate threadurile care așteaptă acum sunt eliberate și se resetează imediat evenimentul	Unul din threadurile care așteaptă acum este eliberat și se resetează imediat evenimentul

Utilizarea obiectelor eveniment poate duce la condiții de cursă, interblocare și alte probleme greu de tratat → se utilizează cu atenție

# Materiale de studiu

---

- Johnson HART, Windows System Programming, 4th edition, Addison Wesley, 2010
  - Capitolul 8
  - Exemplele incluse sunt din cartea de mai sus – vezi arhiva de pe moodle: WSP4\_Examples.zip