Universidade Federal de São Carlos
Departamento de Computação

Sistemas Operacionais

Sincronização de Threads no Windows

Padrões de Projeto com Thread

Vantagens do uso de padrões no projeto com *threads*:

– A maioria dos problemas da programação *multithread* pode ser resolvida facilmente;

– Muitos erros de programação podem ser evitados

– Um vocabulário comum entre os desenvolvedores provê:

- Melhor entendimento dos problemas envolvidos
- Uma melhor análise dos problemas envolvidos

– Os defeitos mais comuns podem ser descritos

Problemas com Threads

Acesso conflitante à memória compartilhada

– Um dos *threads* começa uma operação na memória compartilhada, é suspenso, e deixa a região de memória com uma transformação incompleta;

– O segundo *thread* é ativado e acessa a memória compartilhada em um estado corrompido, causando erros na sua operação e potencialmente no *thread* suspenso quando ele retomar sua execução.

Race Condition & Deadlock

Race Condition

A correta operação depende da ordem de término de duas ou mais atividades independentes

A ordem de término não é determinística

Deadlock

Dois ou mais threads estão esperando para o outro terminar uma certa tarefa.

Sincronização de Thread

Alguns dos métodos de sincronização já vistos são:

- WaitForSingleObject
- WaitForMultipleObject

Sincronização de Thread

Também podemos sincronizar threads por:

- Incremento *Interlocked*;
- Seção Crítica;
- Objetos do Kernel
  - Mutex
  - Semaphore

Armazenamento *volatile*

Qualificador de armazenamento do ANSI C.

Assegura que a variável irá ser armazenada na memória depois da modificação

Informa o compilador que a variável pode mudar de valor a qualquer tempo.

Incremento Interlocked

O incremento *Interlocked* incrementa um inteiro de 32 bits como uma operação atômica. É garantido que termine antes do thread de incremento ser suspenso.

LONG InterlockedIncrement (LONG volatile* lpAddend);

O decremento *Interlocked* decrementa um inteiro de 32 bits como uma operação atômica.

LONG InterlockedDecrement(LONG volatile* lpAddend);

(Ver 1_Exemplo_Interlocked)

Seção Crítica

É a região do código que pode ser acessada somente por um *thread* de cada vez.

Programa Principal

```
InitializeCriticalSection();
        ...
CreateThread();
CreateThread();
CreateThread();
        ...
DeleteCriticalSection();
```

Threads

```
EnterCriticalSection();
        ...
LeaveCriticalSection();
```

Seção Crítica

- Não é um Objeto do *Kernel*
  - Pode aumentar a performance;
  - Pode sincronizar somente *threads* do mesmo processo;
  - Objetos do *Kernel* devem ser usados para sincronizar processos.

Threads podem entrar uma seção crítica que eles já possuam
  - Um contador é mantido;
  - O *thread* tem que sair da seção crítica o mesmo número de vezes que entrou.

A seção crítica não será liberada automaticamente se o *thread* que a mantem é terminado antes de executar LeaveCriticalSection()

Seção Crítica

void InitializeCriticalSection(
LPCRITICAL_SECTION *lpCriticalSection*);

void DeleteCriticalSection(
LPCRITICAL_SECTION *lpCriticalSection*);

void EnterCriticalSection(
LPCRITICAL_SECTION *lpCriticalSection*);

void LeaveCriticalSection(
LPCRITICAL_SECTION *lpCriticalSection*);

lpCriticalSection – Um ponteiro para a Seção Crítica.

(Ver 2_Exemplo_critical_Section)

Objetos do Kernel

Objetos do Kernel

- Estruturas de dados mantidas internamente pelo Sistema Operacional;
- Pode ser usado para sincronização inter-processo;
- Mutexes, Semaphores.

Um contador é associado com:

- Mutex: valor máximo do contador = 1
  - No máximo um thread possui ele.
- Semaphore: valor máximo do contador = N
  - No máximo N threads podem possuir ele.

Objeto do Kernel & Estado do Objeto

O contador:

- Decrementa quando o *Thread* o possui;
- Incrementa quando o *Thread* libera ele;
- Sempre um valor não negativo e não maior que máximo.

Estado do Objeto:

- Um objeto do *kernel* está em um estado sinalizado quando seu contador é maior que zero, caso contrário ele está no estado não sinalizado;
- Um *thread* pode somente possuir um objeto do *kernel* que esteja sinalizado.

Mutex

Mutex pode ser nomeado e ter manipuladores, desta forma eles podem ser também usados para sincronização inter-processo (*threads* em processos diferentes).

Programa Principal

```
HANDLE hMutex;
hMutex = CreateMutex( );
    ...
CreateThread();
CreateThread();
CreateThread();
    ...
CloseHandle();
```

Threads

```
WaitForSingleObject();
    ...
    ...
ReleaseMutex();
```

CreateMutex

A função é usada para criar ou abrir um objeto mutex nomeado ou não nomeado.

HANDLE CreateMutex(
LPSECURITY_ATTRIBUTES *lpMutexAttribute*,
BOOL *bInitialOwner*,
LPCTSTR *lpName*);

Se o valor de *bInitialOwner* é TRUE, o *thread* que cria o mutex o obtém inicialmente, caso contrário não o obtém.

BOOL ReleaseMutex (HANDLE *hMutex*)

(Ver 3_Exemplo_Mutex)

Semaphore

O objeto semáforo é sinalizado quando seu contador for maior que 0 e não sinalizado quando seu contador for 0.

Programa Principal

```
HANDLE hSemaphore;
hSemaphore = CreateSemaphore( );
        ...
CreateThread();
CreateThread();
CreateThread();
        ...
CloseHandle();
```

Threads

```
WaitForSingleObject( );
        ...
        ...
ReleaseSemaphore( );
```

CreateSemaphore

A função é usada para criar ou abrir um objeto semáforo nomeado ou não nomeado.

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

ReleaseSemaphore

Permite que o contador seja incrementado

- O contador do *Release* precisa ser maior que zero;
- O contador do Release não pode exceder o máximo;
- O ponteiro *lpPreviousCount* pode ser NULL, se não necessário;
- Qualquer *thread pode "release"* um semáforo

BOOL ReleaseSemaphore (
HANDLE *hSemaphore*,
LONG *cReleaseCount*,
LPLONG *lpPreviousCount*);

(Ver 4_Exemplo_Semaphore)