Concurrent Programming

Session 6: Thread Management and Synchronization on Win32

Computer Engineering Department
Iran University of Science and Technology
Tehran, Iran

Instructor: Hadi Salimi
Distributed Systems Lab.
Computer Engineering Department,
Iran University of Science and Technology,
hsalimi@iust.ac.ir

Objects and Handles

- An (operating system) object is a data structure that represents a system resource, e.g., file, thread, bitmap.
- An application does not directly access object data or the resource that an object represents. Instead the application must acquire an object handle which it uses to examine or modify the state of the system resource.
- Each handle refers to an entry in an internal object table that contains the address of a resource and means to identify the resource type.

Handle and Objects

- The win32 API provides functions which:
  - Create, get, close, destroy an object
  - Set and get information about the object

- Objects fall into one of three categories:
  - ***kernel objects:*** used to manage memory, process and thread execution, and inter-process communication
  - ***user objects***: used to support window management
  - ***gdi objects:*** supporting graphics operations

Examples

- ***Windows kernel Objects:*** ***kernel32.dll***
  - Events, files and pipes
  - Memory-Mapped Files
  - Mutex and Semaphore objects
  - Processes and Threads

- ***GDI Objects:*** ***gdi32.dll***
  - pens, brushes, fonts and bitmaps

- ***User Objects:*** ***user32.dll***
  - Windows, hooks, menus, mouse cursors

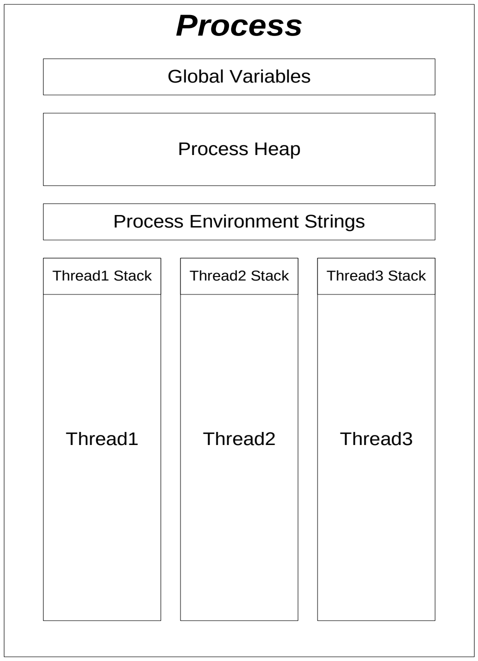
Threads

- A ***thread*** is a path of execution through a program's code, plus a set of resources (stack, register state, etc) assigned by the operating system.
- A thread lives in one and only one process. A process may have one or more threads.
- Each thread in the process has its own call stack, but shares process code and global data with other threads in the process.
- Pointers are process specific, so threads can share pointers.

Threads vs. Processes

- A Process is inert. A process never executes anything; it is simply a container for threads.
- Threads run in the context of a process. Each process has at least one thread.
- A thread represents a path of execution that has its own call stack and CPU state.

Threads vs. Processes



Process

Global Variables

Process Heap

Process Environment Strings

Thread1 Stack

Thread2 Stack

Thread3 Stack

Thread1

Thread2

Thread3

Thread Scheduling

- Windows XP, 2000 and NT are preemptive multi-tasking systems. Each task is scheduled to run for some brief time period before another task is given control of CPU.

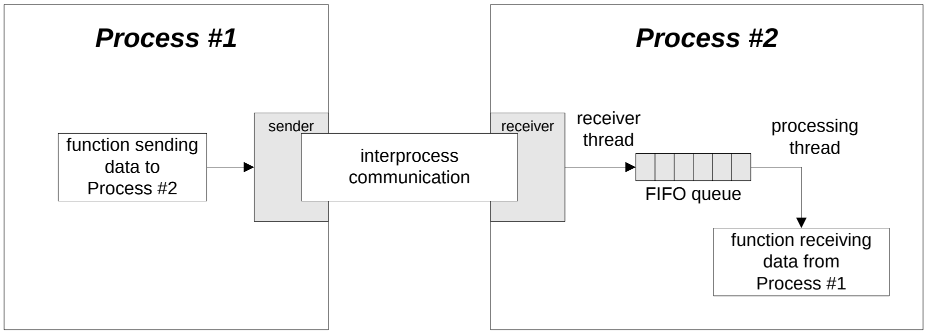
- Threads are the basic unit of scheduling on current Win32 platforms.

Thread Benefits

- Keeping user interfaces responsive even if required processing takes a long time to complete.
  - handle background tasks with one or more threads
  - service the user interface with a dedicated thread
- Take advantage of multiple processors available for a computation.
- Avoid low CPU activity when a thread is blocked waiting for response from a slow device or human by allowing other threads to continue.

Avoid Blocking

Non-Blocking Communication in Asynchronous System



Process #1

function sending data to Process #2

sender

interprocess communication

Process #2

receiver

receiver thread

FIFO queue

processing thread

function receiving data from Process #1

Potential Problems with Threads

- Conflicting access to shared memory
  - one thread begins an operation on shared memory, is suspended, and leaves that memory region incompletely transformed
- Race Conditions occur when:
  - correct operation depends on the order of completion of two or more independent activities
- Starvation
  - a high priority thread dominates CPU resources, preventing lower priority threads from running often enough or at all.
- Deadlock

MFC and Win32 API

- Developers can work with threads using either Windows API or supplementary libraries such as MFC.
- Using such libraries abstracts developers viewpoint form detailed and complex concepts of the operating system.

Creating Threads

```
CWinThread* AfxBeginThread(pfnThreadProc, pParam)
```

- pfnThredProc
  - Pointer to the thread function
- pParam
  - Parameter that is passed to the thread function.

```
void AfxEndThread(nExitCode , bDelete)
```

Thread Synchronization

- Synchronizing threads means that every access to data shared between threads is protected.
- The principle means:
  - Interlocked increments
  - Critical Sections
  - Mutexes
  - Events

Interlocked Operations

- InterlockedIncrement increments a 32 bit integer as an atomic operation. It is guaranteed to complete before the incrementing thread is suspended.

  ```
  long value = 5;
  InterlockedIncrement(&value);
  ```

- InterlockedDecrement decrements a 32 bit integer as an atomic operation:

  ```
  InterlockedDecrement(&value);
  ```

Win32 Critical Sections

- CCriticalSection is an MFC class used to protect a critical region against concurrent thread access.

```
CCriticalSection cs;
cs.lock();
… // desired critical region


cs.Unlock();
```

MFC Mutexes

- A mutex synchronizes access to a resource shared between two or more threads.
  - CMutex constructs a mutex object
  - Lock locks access for a single thread
  - Unlock releases the resource for acquisition by another thread

```
CMutex cm;
cm.Lock();
  // access a shared resource
cm.Unlock();
```

- CMutex objects are automatically released if the holding thread terminates.

CSingleLock & CMultiLock

- CSingleLock and CMultiLock classes can be used to wrap critical sections, mutexes and events,.

```
CCriticalSection cs;
CSingleLock slock(cs);
slock.Lock();
   // do some work on a shared resource
slock.Unlock();
```

This CSingleLock object will release its lock if an exception is thrown inside the synchronized area, because its destructor is called. That does not happen for the unadorned critical section.