C++
Threads

CO650 Advanced Programming

Multitasking

- Two types of multitasking
- **Process-based** - Allows two or more programs to run concurrently.
- **Thread-based** – Allows concurrent execution of blocks of code within a Program.
- All processes have at least one thread
- A program with more than one thread can perform tasks concurrently.
- True concurrency is only achieved on multiple-CPU / core systems

Advantages

- Utilize CPU idle time
- Idle time typically caused by CPU waiting for I/O devices (network ports, disk drives, keyboard).
- **Example**

  While you Network Server is waiting for new players to connect, it could be processing the data from connected players and managing the games cycle.

Multithreading with C++

- Pre C++ 11 didn’t not directly support multithreading unlike C# & Java
- Operating systems provide thread support through APIs
- The support provided can vary. Windows provides functions for fine grain management of threads.
- C++ can access these APIs (<windows.h>)
- C++ threaded applications can take advantage of unique concurrency features of O.S. Leading to higher performance.

Creating a Thread

- The CreateThread function creates a new thread

```
HANDLE CreateThread( LPSECURITY_ATTRIBUTES secAttr,
                     SIZE_T stackSize,
                     LPTHREAD_START_ROUTINE threadFunc,
                     LPVOID param,
                     DWORD flags,
                     LPDWORD threadID);
```

- threadFunc is the name of the function that will be invoked within the new thread.

```
DWORD threadId;
HANDLE hdl;
hdl = CreateThread(NULL ,0, MyThreadFunction, NULL ,0, &threadId);
```

CreatingThread Arguments

Argument Type	Description
LPSECURITY_ATTRIBUTES	A pointer to a set of thread security attributes that control the access to the thread. Default attributes are applied if this value is set to NULL
SIZE_T	Size in bytes of the thread’s stack. If set to zero the size will match that of the creating threads stack.
LPTHREAD_START_ROUTINE	Pointer to a callback function that runs the new thread
LPVOID	Parameters
DWORD	Flag indicating execution state of thread. If set to zero the thread executes immediately
LPDWORD	The address of a int variable that holds the thread ID

The Thread Function

- An argument passed to CreateThread
- The thread’s equivalent of main.
- Must conform to the signature below

```
DWORD WINAPI threadfunc(LPVOID param);
```

- Execution continues until the function returns.
- Accepts arguments via the param.

```
DWORD WINAPI BasicThread(LPVOID param) {
		cout << "Thread start \n";
		Sleep(2000);
		cout << "Thread end \n";
		return 0;
}
```

Thread Example

```
#include "stdafx.h"
#include <iostream>

using namespace std;

DWORD WINAPI BasicThread(LPVOID param){
		cout << "Thread start \n";
		Sleep(2000);
		cout << "Thread end \n";
		return 0;
}

int main(){
		DWORD threadId;
		HANDLE hdl;
		hdl = CreateThread(NULL,0,BasicThread,NULL,0,&threadId);
		system("pause");
		return 0;
}
```

Passing Arguments

- Type cast the argument being sent to the thread callback to (LPVOID).
- Within the callback type cast the parameter back to its original type

```
DWORD WINAPI BasicThread( LPVOID param ) {
		GameObject *obj = (GameObject*)param;
		// Invoke member functions of obj here
}

int main(){
		DWORD threadId;
		HANDLE hdl;
		GameObject *obj = new GameObject();
		hdl = CreateThread(NULL,0,SyncThread, (LPVOID)obj ,0, &threadId);
}
```

Thread Functions

- Threads can be suspended

```
DWORD SuspendThread(HANDLE hThread);
```

- Suspended threads can be resume execution

```
DWORD ResumeThread(HANDLE hThread);
```

Prioritising Threads

- You may find situations where a multi thread application needs to prioritise the execution of the threads.

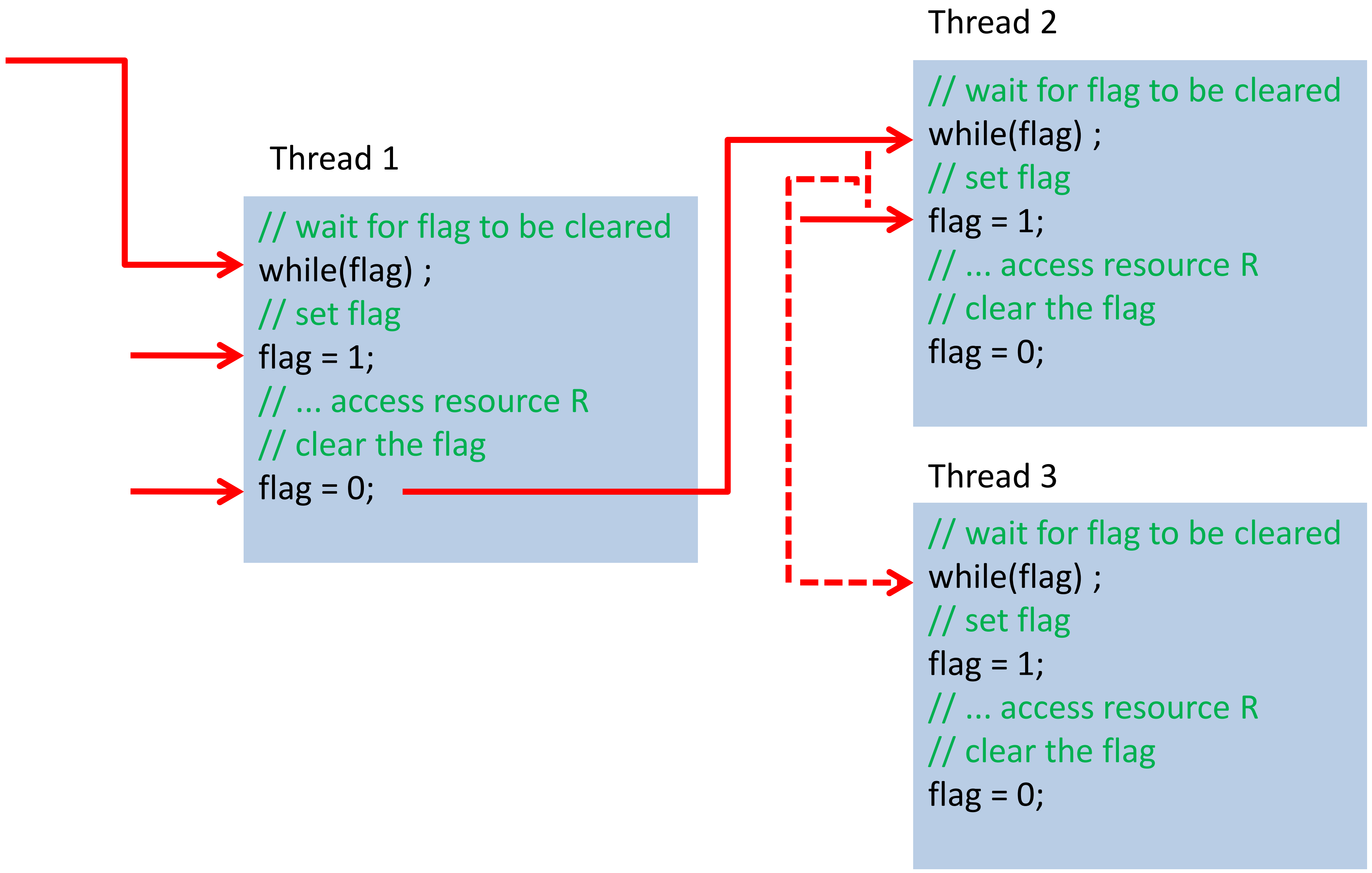
```
BOOL SetPriorityClass(HANDLE hApp, DWORD priority);
```

Thread Priority	Value
THREAD_PRIORITY_TIME_CRITICAL	15
THREAD_PRIORITY_HIGHEST	2
THREAD_PRIORITY_ABOVE_NORMAL	1
THREAD_PRIORITY_NORMAL	0
THREAD_PRIORITY_BELOW_NORMAL	-1
THREAD_PRIORITY LOWEST	-2
THREAD_PRIORITY_IDLE	-15

Synchronization

- Coordinate activity of two or more threads
- Manage shared resources
- Example : Ensure a file is only written to by one thread at a time.
- It is necessary to pause one thread until the correct state exists for it to resume execution.
- A blocked thread is one that is waiting for a resource (event) to become available before it can continue executing.

Synchronization Problem



Thread 1

```
// wait for flag to be cleared
while(flag) ;
// set flag
flag = 1;
// ... access resource R
// clear the flag
flag = 0;
```

Thread 2

```
// wait for flag to be cleared
while(flag) ;
// set flag
flag = 1;
// ... access resource R
// clear the flag
flag = 0;
```

Thread 3

```
// wait for flag to be cleared
while(flag) ;
// set flag
flag = 1;
// ... access resource R
// clear the flag
flag = 0;
```

Solution

- Windows provides functions that will in one uninterrupted operation test and if possible set the value of a flag.
- Synchronisation flags are known a semaphores.
- Four types
  - Semaphore : Restricts the number of threads that can access a resource concurrently.
  - Mutex Semaphore : Only one thread can access the resource concurrently.
  - Event Object : signals when an event has occurred
  - Waitable timer : Blocks a thread until a specific time is reached.

Mutex Functions

- Windows provide a function to create and set a mutex

```
HANDLE CreateMutex(LPSECURITY_ATTRIBUTES secAttr,
                   BOOL acquire,
                   LPCSTR name);
```

- The global mutex object is assigned a name. Threads use this name to refer to the same mutex object.
- The handle returned by CreateMutex is used to manage the mutex within the thread.
- If the acquire argument is set to true the CreateMutex will try to gain control of the mutex.

Mutex Functions

- The WaitForSingleObject function blocks until the mutex becomes available or the time-out occurs.

```
DWORD WaitForSingleObject(HANDLE hObject,
                          DWORD   howLong);
```

- This function should be placed before the code that accesses the shared resource.
- The function returns WAIT_TIMEOUT if the resource was still unavailable after the time-out occurred

Mutex Functions

- A thread should release the mutex after it has finished with the resource.

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

- It returns zero if the release failed.

Mutex Example

```
char mutexName[] = "MUTEX1";
HANDLE hMutex;

DWORD WINAPI SyncThread(LPVOID param){
		WaitForSingleObject(hMutex,INFINITE);
		// Access resource
		ReleaseMutex(hMutex);
		return 0;
}

int main(){
		DWORD threadId;
		HANDLE hdl;
		hMutex = CreateMutex(NULL,false,LPCWSTR(mutexName));
		hdl = CreateThread(NULL,0,SyncThread, NULL,0, &threadId);
		…………..
}
```

The acquire argument is set to false when creating the Mutex to ensure the first call to WaitForSingleObject can gain control and lock the Mutex.

Issues

- A member function can only become a thread if it is static. Non static Member functions have the **this** hidden parameter passed to them thus not conforming to the thread prototype rules. Static functions belong to the class not the object and therefore are not passed **this**.
- For a tutorial on the new threading support in C++ 11 see http://www.bogotobogo.com/cplusplus/C11/1_C11_creating_thread.php