CO650 Advanced Programming

C++ Threads

Multitasking

There are 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 of Multitasking

- True multiple tasking allows to make use of CPU idle time by running tasks in parallel instead of waiting for one task to complete
- Be efficient in processing by making use of true concurrent processing and takes less time for getting things done.

Multithreading with C++

Previously C++ 11 did not directly support multithreading like C# or Java. In 2011 release thread class came about and makes it easier to create and manage threads. Before this, to create threads in C++ it required the use of APIs to connect with the resources of the operating system.

Creating a Thread

Following CreateThread function creates a new thread

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

Function Parameter Definitions

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. Generic type to provide data used by the thread.
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

Example of usage

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

threadFunc is the name of the function that will be invoked within the new thread. CreateThread function returns as a HANDLE data type which is a Windows data type that provides an abstraction of a resource. This can be used as a unique identifier for the thread.

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;

}
```

```
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 resume execution

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

Prioritizing Threads

You may find situations where a multithreaded 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	16

Synchronization

Synchronization is coordinating activity of two or more threads. There may be issues with synchronization when threads are trying to access and write to the same resource. Because of this issue, it may be necessary to restrict the access of a particular resource to one thread at one time.

This can be accomplished by implementing a basic **locking mechanism**. However, a time delay between the process of locking and unlocking may cause overlapping locks if they check resource status at the same time. So ideally, we need a way to lock and unlock without delay so that only one thread may be allowed to access at a time.

This is provided by Windows as functions that will in one uninterrupted operation test and if possible set the value of a flag. These synchronization flags are known as **semaphores**.

There are four types of semaphores:

- Semaphore: Restricts the number of threads that can access a resource concurrently.
- Mutext 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. Setting it to false will wait for a thread to attempt to access the resource first.

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

- The WaitForSingleObject function blocks until the mutex becomes available or the time-out occurs. DWORD type of howLong corresponds to the number of milliseconds in integer.

- The function returns WAIT_TIMEOUT if the resource was still unavailable after the time-out occurred.

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

- A thread should release the mutex after it has finished with the resource.
- It returns zero if the release fails.

Mutex Example

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

DWORD WINAPI SyncThread(LPVOID param) {

WaitForSingleObject(hMutex, INFINITE);
// Access resource
ReleaseMutex(hMutex);
return0;

}

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.