

C++ Week 9 Threads

Create a new C++ console project.

The first step is to create a thread function that will contain the statements to be executed within a new thread. Add the function above your main function.

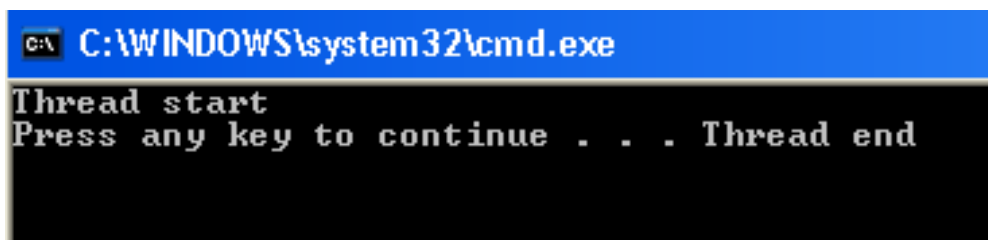
```
DWORD WINAPI BasicThread(LPVOID param){  
    cout << "Thread start \n";  
    Sleep(2000);  
    cout << "Thread end \n";  
    return 0;  
}
```

Note how the function signature conforms to that outlined within the presentation. The windows types DWORD, LPVOID are defined within the windows.h header file so be sure to include this within your project.

We shall now invoke the thread within the main function using the CreateThread function.

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

When you compile and run the application you output should resemble that below



```
C:\WINDOWS\system32\cmd.exe  
Thread start  
Press any key to continue . . . Thread end
```

The previous exercise demonstrates that the BasicThread function is being executed in parallel to the code within the main function. While main is waiting for the user to “Press any key” the cout statement within BasicThread outputs the message “Thread end”.

C++ Week 9 Threads

Exercise 1

Comment out the `system("pause");` statement and run the program once more using the Debug->Start Debugging menu option. What happened and why?

Now add the new thread function below to your program

```
DWORD WINAPI MessageThread(LPVOID param){
    for(int n=0;n<10;n++){
        cout << " Thread \n";
        Sleep(500);
    }
    return 0;
}
```

The function iterates through a loop 10 times displaying a message.

Now modify the main function in line with that below.

```
int main(){
    LPVOID param = NULL;
    DWORD threadId;
    HANDLE hdl[3];
    int label = 1;
    hdl[0] = CreateThread(NULL,0,MessageThread,(LPVOID)label,0,&threadId);
    label = 2;
    hdl[1] = CreateThread(NULL,0,MessageThread,(LPVOID)label,0,&threadId);
    label = 3;
    hdl[2] = CreateThread(NULL,0,MessageThread,(LPVOID)label,0,&threadId);
    system("pause");
    return 0;
}
```

Here we have created three threads.

Run the program and view the results. The output is somewhat confusing since we don't know which thread is outputting the "Thread" message. Let's make it a little clearer by ensuring that each thread outputs to a separate column within the console. Amend `MessageThread` in line with the code below.

C++ Week 9 Threads

```
DWORD WINAPI MessageThread(LPVOID param){
    int tab = (int)param;
    for(int n=0;n<10;n++){
        for(int i=0;i < 100000;i++){
            if (tab ==1)
                cout <<"\t\t 1:Thread\n";
            else if (tab == 2)
                cout <<"\t\t\t\t 2:Thread\n";
            else
                cout <<"\t\t\t\t\t\t 3:Thread\n";
        }
    }
    return 0;
}
```

When we create the thread we pass MessageThread a unique number 1,2 or 3 which we can use to determine how many tabs the message should be indented by.

When you run the program now you will notice that the pause message is displayed at the start (or possible middle) of the output not at the end as desired. This is because the main function thread has reached the system("pause") instruction before the three threads have completed their iterations. To stop the main thread from executing before it reaches the system call, we can include the two lines below just above system("pause").

```
HANDLE mainThread = GetCurrentThread();
SuspendThread(mainThread);
```

Here we are acquiring the handle of the main thread and suspending it.

Run the program once more.

There may be situations when you have a number of threads running concurrently and you want certain threads to have priority over others. The priority level of a thread can be set by invoking the SetThreadPriority method passing it the thread handle and a priority level.

Amend main in line with the code below and run the program once more.

C++ Week 9 Threads

```
int main(){
    LPVOID param = NULL;
    DWORD threadId;
    HANDLE hdl[3];
    int label = 1;
    hdl[0] = CreateThread(NULL,0,MessageThread,(LPVOID)label,0,&threadId);
    SetThreadPriority(hdl[0],THREAD_PRIORITY_IDLE);
    label = 2;
    hdl[1] = CreateThread(NULL,0,MessageThread,(LPVOID)label,0,&threadId);
    SetThreadPriority(hdl[1],THREAD_PRIORITY_TIME_CRITICAL);
    label = 3;
    hdl[2] = CreateThread(NULL,0,MessageThread,(LPVOID)label,0,&threadId);
    SetThreadPriority(hdl[2],THREAD_PRIORITY_LOWEST);
    HANDLE mainThread = GetCurrentThread();
    SuspendThread(mainThread);
    system("pause");
    return 0;
}
```

The output demonstrates that the thread with the highest priority completes first (see the separate thread handout for a full list of the priorities).

Thread Synchronization

Comment out the code within main and add that below. Here we have defined a simple game object which has been instantiated within main and passed as an argument to two thread functions.

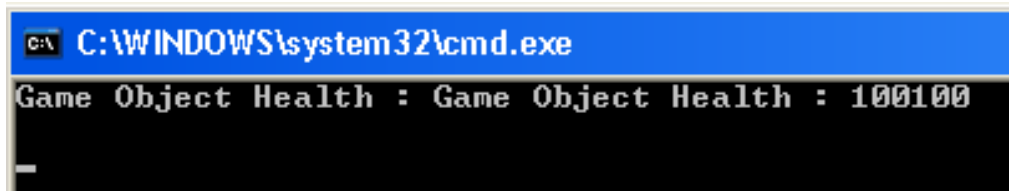
C++ Week 9 Threads

```
class GameObject{
public:
    int health;
    GameObject(){
        health = 100;
    }
};

DWORD WINAPI SyncThread(LPVOID param){
    GameObject *syncObj = (GameObject*)param;
    cout << "Game Object Health : " << syncObj->health << "\n";
    return 0;
}

int main(){
    LPVOID param = NULL;
    DWORD threadId;
    HANDLE hdl[2];
    GameObject *obj = new GameObject();
    hdl[0] = CreateThread(NULL,0,SyncThread,(LPVOID)obj,0,&threadId);
    hdl[1] = CreateThread(NULL,0,SyncThread,(LPVOID)obj,0,&threadId);
    HANDLE mainThread = GetCurrentThread();
    SuspendThread(mainThread);
    system("pause");
    delete obj;
    return 0;
}
```

The output from the program is illustrated below

A screenshot of a Windows command prompt window. The title bar is blue and contains the text "C:\WINDOWS\system32\cmd.exe". The command prompt area has a black background with white text. The output of the program is displayed as "Game Object Health : Game Object Health : 100100". The first "100" is on the same line as the first "Game Object Health :", and the second "100" is on the same line as the second "Game Object Health :". There is a small white cursor at the bottom left of the command prompt area.

```
C:\WINDOWS\system32\cmd.exe
Game Object Health : Game Object Health : 100100
```

C++ Week 9 Threads

Exercise 2

Why is it outputting 100100?

To avoid this problem we can synchronize the execution of separate threads by creating a mutex semaphore. A thread must acquire the mutex before it can output to the console, if it can't acquire the mutex because it has been acquired by another thread, it must wait until the mutex has been released by the other thread and then acquire it, so that it may output to the console.

Mutex semaphores are identified by their name which must be available to all threads, so let's define this as a global variable outside of any functions

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

Now modify SyncThread in line with the code below

```
DWORD WINAPI SyncThread(LPVOID param){  
    GameObject *syncObj = (GameObject*)param;  
    WaitForSingleObject(hMutex,INFINITE);  
    cout << "Game Object Health : " << syncObj->health << "\n";  
    ReleaseMutex(hMutex);  
    return 0;  
}
```

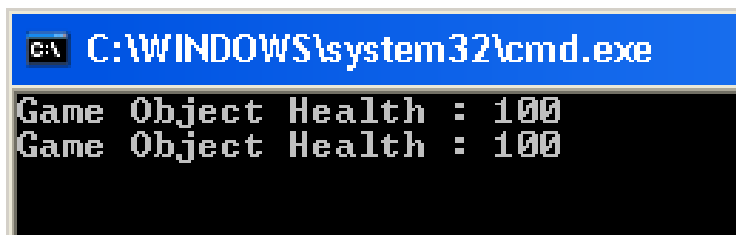
and the mutex is released.

Finally create the mutex within main before the threads are created and set it's acquire state **false**.

```
hMutex = CreateMutex(NULL,false,LPCWSTR(mutexName));
```

The output from the modified program is illustrated below.

CO650 Advanced Pi



```
C:\WINDOWS\system32\cmd.exe  
Game Object Health : 100  
Game Object Health : 100
```

C++ Week 9 Threads

Exercise 3

The code below implements the logic for a simple battleship game , where the players are represented by two thread. Familiarize yourself with the code and the logic behind the game. As it stands the program does not implement turn based game play. Correct this by including additional code that implements mutex semaphores.

Add the ocean class to your project.

```
#include <iostream>
#include <ctime>

using namespace std;

class Ocean{
public:
    // Values that can be assigned to the grid cells
    enum Cells {unknown_cell, visited_cell, ship_cell};
    // Players ids
    enum Players {unassigned = -1, redPlayer = 0, bluePlayer =1};
private:
    // Two 10 * 10 grids. One for each player
    Cells grids[2][10][10];
    Players players[2];
    int currentPlayer;
    // Flag indicating end of game
    bool finished;
public:
    Ocean();
    bool IsFinished();
    void SetFinished();
    bool setCurrentPlayer(Players player);
    char * GetPlayerDescription(Players player);
    Players AssignPlayer();
    Cells GetValue(Players player, int x, int y);
    bool Play(Players player, int x, int y);
};
```

C++ Week 9 Threads

```
Ocean::Ocean(){
    finished = false;
    int posX,posY;
    // Initialise players and grid cells
    players[0] = Players::unassigned;
    players[1] = Players::unassigned;
    for (int x=0;x <10;x++){
        for (int y=0;y <10;y++){
            grids[0][x][y] = Cells::unknown_cell;
            grids[1][x][y] = Cells::unknown_cell;
        }
        // Assign ship to a random cell within the first grid
        posX = rand() % 10;
        posY = rand() % 10;
        grids[0][posX][posY] = Cells::ship_cell;
        cout << "Player 0 ship x:" << posX << " y:" << posY << "\n";
        // Assign ship to a random cell within the second grid
        posX = rand() % 10;
        posY = rand() % 10;
        grids[1][posX][posY] = Cells::ship_cell;
        cout << "Player 1 ship x:" << posX << " y:" << posY << "\n";
    }

    bool Ocean::IsFinished(){
        return finished;
    }

    void Ocean::SetFinished(){
        finished = true;
    }
    // Turn based logic
    bool Ocean::setCurrentPlayer(Players player){
        if (currentPlayer != player){
            currentPlayer = player;
            return true;
        }
        else{
            return false;
        }
    }
    char * Ocean::GetPlayerDescription(Players player){
        if (player == Ocean::Players::redPlayer)
            return "Red";
        else
            return "Blue";
    }
}
```


C++ Week 9 Threads

```
// Players are assigned an ID of 0 or 1 at the start of play
Ocean::Players Ocean::AssignPlayer(){
    if (players[0] == Players::unassigned){
        players[0] = Players::redPlayer;
        cout << "Red player assigned \n";
        return Players::redPlayer;
    }
    else {
        players[1] = Players::bluePlayer;
        cout << "Blue player assigned \n";
        return Players::bluePlayer;
    }
}

// Return the value within a cell
Ocean::Cells Ocean::GetValue(Players player, int x, int y){
    return grids[!player][x][y];
}

// Returns true if a ship is located within the
// player's grid at position x,y.
// If not flags the cell as being visited
bool Ocean::Play(Players player, int x, int y){
    if (player == Players::redPlayer){
        if (grids[Players::bluePlayer][x][y] == ship_cell){
            finished = true;
        }
        else {
            grids[Players::bluePlayer][x][y] = visited_cell;
        }
    }
    else{
        if (grids[Players::redPlayer][x][y] == ship_cell){
            finished = true;
        }
        else {
            grids[Players::redPlayer][x][y] = visited_cell;
        }
    }
    return finished;
}
```

C++ Week 9 Threads

Now include the thread function below (just above main). Comment out the previous version of this function.

```
DWORD WINAPI SyncThread2(LPVOID param){
    Ocean *ocean = (Ocean*)param;
    Ocean::Cells target;
    int x,y;
    char playerDescription[5];
    // Get a unique player for this thread - either red or blue
    Ocean::Players player = ocean->AssignPlayer();
    if (player == Ocean::Players::redPlayer)
        strcpy_s(playerDescription,"Red");
    else
        strcpy_s(playerDescription,"Blue");
    srand (player);
    do{
        if (!ocean->IsFinished() && ocean->setCurrentPlayer(player)){
            do{
                // Select a random cell
                x = rand() % 10;
                y = rand() % 10;
                // Get the cells value
                target = ocean->GetValue(player,x,y);
                // If the cell has not been visited before
                if (target == Ocean::Cells::unknown_cell){
                    ocean->Play(player,x,y);
                    cout << ocean->GetPlayerDescription(player)<< " x:" << x <<
" y: " << y << "\n";
                    break;
                } else {
                    // Opponent's ship found!
                    if (target == Ocean::Cells::ship_cell) {
                        ocean->SetFinished();
                        cout << ocean->GetPlayerDescription(player) << " x:"
<< x << " y: " << y << "\n";
                        cout << "Player " << ocean-
>GetPlayerDescription(player) << " Wins\n";
                        break;
                    }
                }
            }
            // If the random cell has been visited before then try another one
            while(target == Ocean::Cells::visited_cell);
        }
    }
    // Until a ship has been found
    while(!ocean->IsFinished());
    return 0;
}
```

C++ Week 9 Threads

```
srand ( time(0) );  
LPVOID param = NULL;  
DWORD threadId;  
HANDLE hdl[2];  
Ocean *ocean = new Ocean();  
hdl[0] = CreateThread(NULL,0,SyncThread2,(LPVOID)ocean,0,&threadId);  
hdl[1] = CreateThread(NULL,0,SyncThread2,(LPVOID)ocean,0,&threadId);  
HANDLE mainThread = GetCurrentThread();  
SuspendThread(mainThread);  
system("pause");  
delete ocean;
```

The next step is to make the mutexname variable that is defined within your main project source file available to the Ocean.cpp where it is accessed by the code. Unfortunately if we move the definition to the cpp file we will get a multiple definitions error as the cpp file might be included in more than one compiled unit. The solution is to include a declaration of the variable within the cpp file that informs the compiler that the actual definition is in a different file. Add the following line to the Ocean.cpp file to achieve this.

```
extern char mutexName[];
```

Finally implement the changes required to make the game function correctly.