G52OSC
OPERATING SYSTEMS AND CONCURRENCY

Sharing Data

Dr Jason Atkin

Last 2 lectures

- Creating threads and processes
- Windows programs
  - Remember that you do not need to memorise API functions and their parameters
  - Just understand the concepts
- Message/Event loops
  - These an important example / illustration of a concept of inter-process communication

Threads and processes

- You create one new thread at a time
- There is an overhead in creating threads
  - E.g. Function **`CreateThread()`** takes time
  - Operating system has to allocate the resources for the thread
- You can create other processes
  - Usually fork() in Unix/Linux and CreateProcess in MS Windows
  - Processes have a separate address space
- You have no idea what order the threads/processes will execute, where any interleaving happens, etc
  - You don't know which will finish first
  - You may need to wait for other threads or processes to finish
- On windows you will get a Handle back to the operating system object, which you can pass to other functions…

Reminder: Windows Handles

- A **`Handle`** gives you something to use to ‘grip’ a windows object that the system owns
- Processes, threads, windows, events, mutexes etc all have handles that you can use
- You can’t do anything with these apart from use them to refer to the object
- CreateThread returns a Handle for the thread that was created – we should really close it when we finish with it
- We store all of these thread handles in an array in case we need them later

```
arrdwThreadHandles[iTN] =
            CreateThread( … );
```

This Lecture

- Reminder of thread creation
  - And shared data
- Volatile
- Shared globals / address space
- Reminder of process creation
- Mapping shared memory
- struct pointers

- Atomic updates are next lecture

Coordination between threads

- Two coordination problems:
  - Synchronise operations
    - Don’t do <?> until the other thread finished <?>
    - E.g. wait for a thread to end before continuing
  - Share information
    - Coordinate access to data – avoiding issues
    - Mutual exclusion
    - Prevent two threads accessing same data at once
- What I say about threads applies to processes too
  - Easier to illustrate with threads

Program from previous lecture

The thread function

```
DWORD WINAPI thread_function( LPVOID lpParam )
{
   for ( int j = 0; j < 20 ; j++ )
   {
      printf( "Thread %d running %d... total = %d\n",
              (int)lpParam,       j,     dwTotal );
      for ( int i = 0; i < 1000000/20; i++ )
         dwTotal++;
   }
   printf( "Total when thread %d ended was %d\n",
        (int)lpParam,      dwTotal );
   return 0;
}
```

We use this as a thread number

Outer loop

Inner loop

Increment the variable

Purpose:
Thread function will increase the value of this variable by one million.
Does it in 20 parts so we can see what it does (via printf).

Create the threads

```
int main()
{
   HANDLE arrdwThreadHandles[NUM_THREADS];

   for ( int iTN = 0; iTN < NUM_THREADS - 1; ++iTN )
      arrdwThreadHandles[iTN] = CreateThread(
            NULL, 0, thread_function, (LPVOID)iTN, 0, NULL );

   thread_function( (LPVOID)(NUM_THREADS - 1) );

   WaitForMultipleObjects( NUM_THREADS - 1,
           arrdwThreadHandles, TRUE, 10000 );

   printf( "Press RETURN" );
   while ( getchar() != '\n' )    ;
   return 0;
}
```

Shared data

- We used a global variable
  - Changed it from both threads
  - It was shared
- When we created a new process, it had its own global variables
  - Not shared between processes
  - Even multiple copies of the same program
- Accessing the same variable caused problems – explained next lecture

Volatile

- In general, incrementing the variable does the following:
  1. Load current value from memory into a register
  2. Increment register value
  3. Store new value back into memory
  - If you do it in a loop (e.g. 1 million times), it would repeat this
- Optimising compilers can do better
  1. Load current value from memory into a register
  2. Increment register value by 1 million
  3. Store new value back into memory
  - Eliminated the need for a loop, and MUCH faster
- **Volatile tells the compiler that the value of the variable can be seen or altered elsewhere** so don't do optimisation which assums that it isn't visible
  - i.e. it turns some optimisation off (i.e. it may be slower!)

Location of globals and locals

```
#include <Windows.h>
#include <stdio.h>

int global = 1;

DWORD WINAPI
   func1( LPVOID param )
{
   int local = 0;

   printf( " Thread %d:\n
global is at address %p,\n
local is at address %p\n",
   (int)param,
   &global,
   &local );
   return 0;
}
```

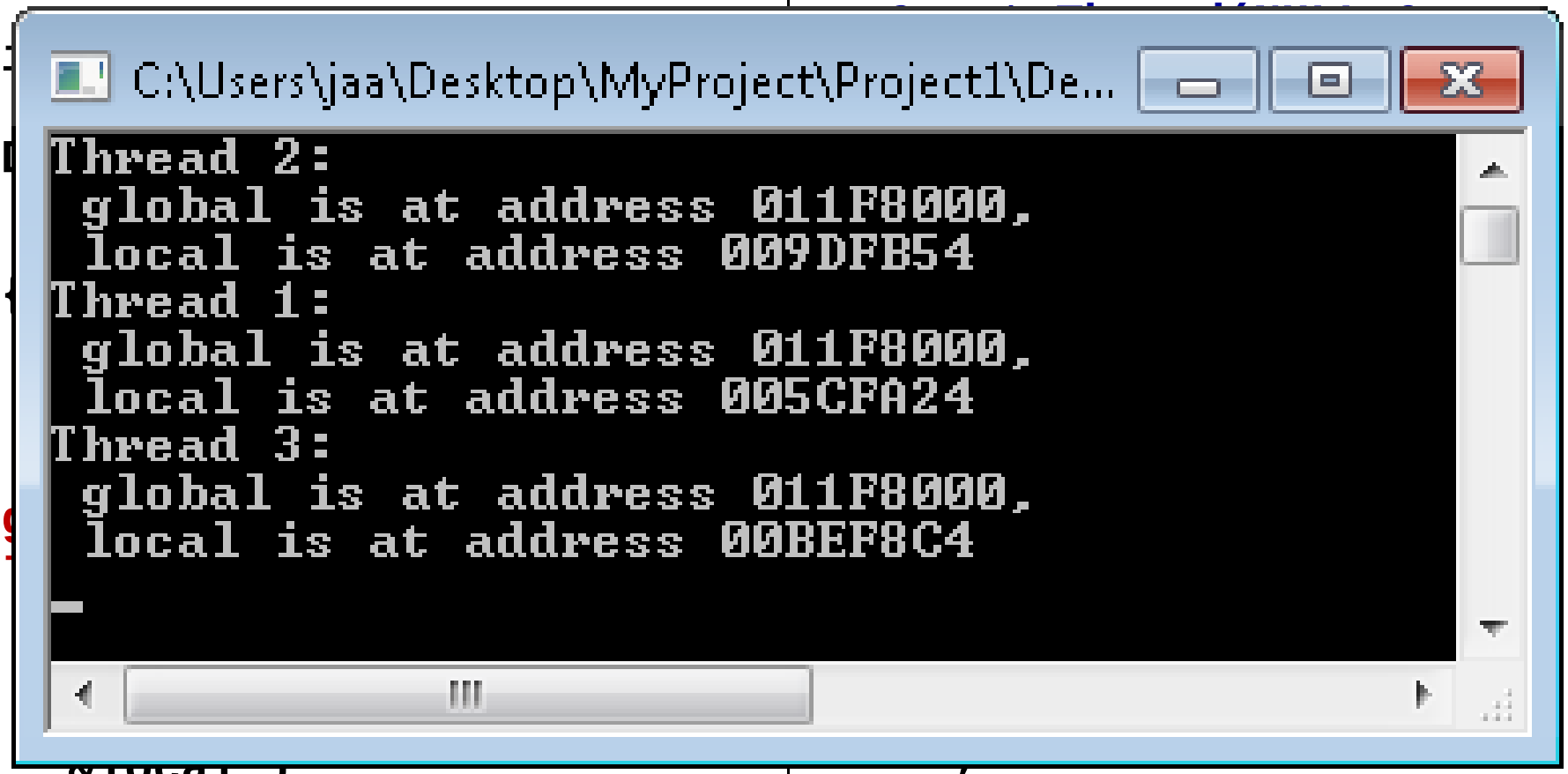
```
int main()
{
   CreateThread(NULL,0,
      func1, (LPVOID)1,
      0,NULL );
   CreateThread( NULL,0,
      func1, (LPVOID)2,
      0,NULL );
   CreateThread(NULL,0,
      func1, (LPVOID)3,
      0,NULL );

   while(getchar() != '\n')
      ;
   return 0;
}
```

Location of globals and locals

```
#include <Windows.h>
#include <stdio.h>
```

```
int main()
{
```




```
    return 0;
}
```

Proof? shared local variables

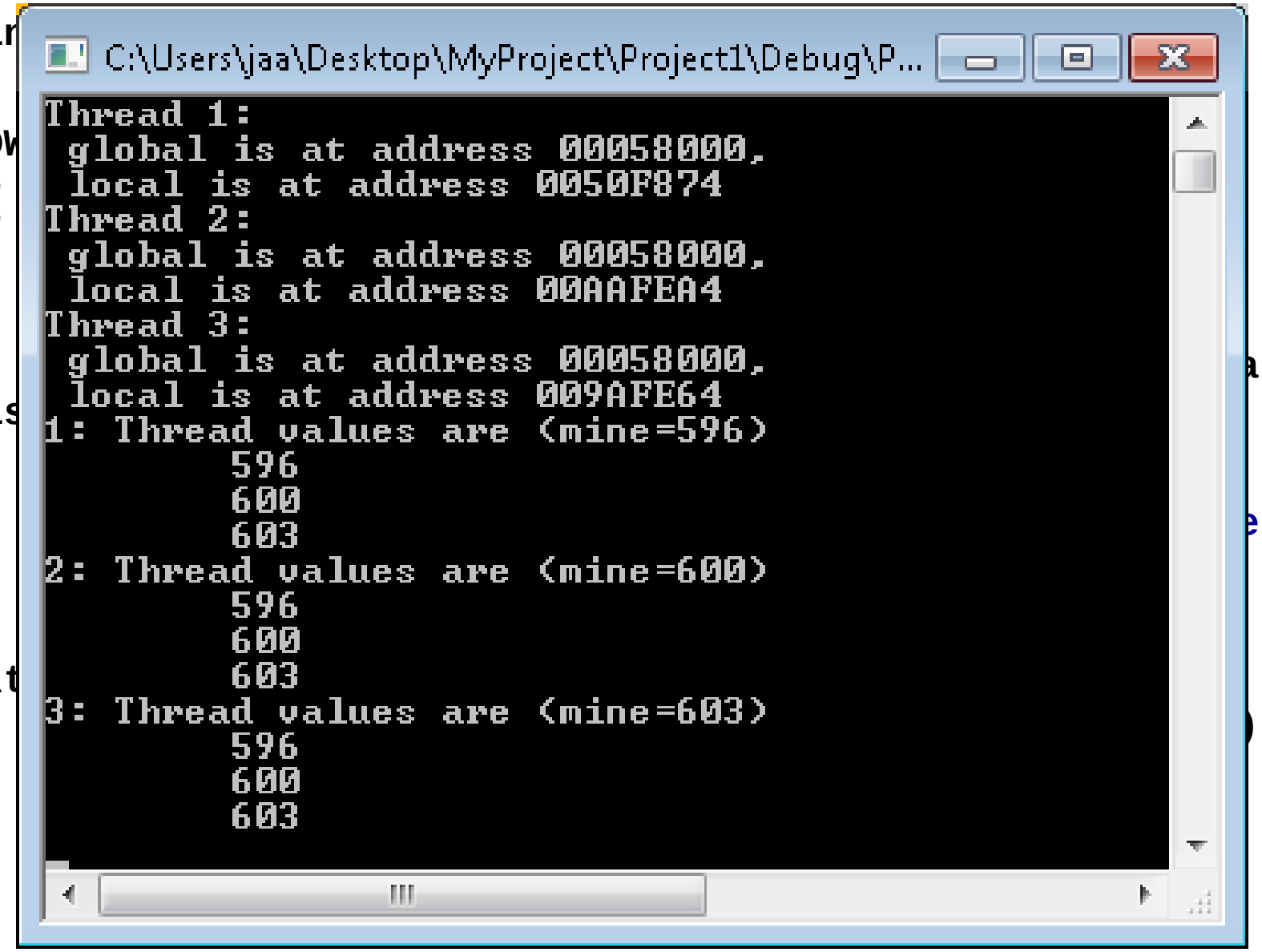
```
int* locals[4];
```

Change func1() to the following, which changes the value of the variable

```
DWORD WINAPI func1( LPVOID param )
{
   srand( GetTickCount() + (int)param );
   int local = (int)rand(); // changed to random number
   printf( "Thread %d:\n global is at address %p,\n local
is at address %p\n", (int)param,&global, &local );

   locals[(int)param] = &local; // Store pointer to mine
   Sleep( 2000 ); // Wait 2 seconds
   printf( "%d: Thread values are (mine=%d)\n
\t%d\n\t%d\n\t%d\n",
   (int)param, local, *locals[1],*locals[2],*locals[3] );
   Sleep( 2000 ); // Wait 2 seconds - hope it is enough
   return 0;
}
```

Proof? shared local variables



Local variables and parameters

- Calling a function will create new local variables and actual parameter values for the function call
  - Has to do this, or recursion would not work
  - E.g.

```
int factorial( int n )
{
   if (n<=1)
      return 1;
   else
      return n * factorial(n-1);
}
```

- So a thread function has its own local variables
  - Local variables are created on the stack for the current thread

Local vs global variables

- Global variables are created in a shared area of memory
  - Will be shared between all threads
  - The name refers to the same thing whichever thread you are in
- Local variables are created within the stack for the current thread
  - Local variables are not shared between functions
  - i.e. each function has its own copy
- But all are in the same address space
  - So if a thread can find them, any thread can access any item of data (e.g. give it a pointer to them)

Using a local to pass information

```
#include <Windows.h>
#include <stdio.h>

volatile int* pglobal = NULL;

int main()
{
      CreateThread( NULL,0,func1,0,0,NULL );
      func2(NULL);
      return 0;
}
```

Storage for pointer

Create a thread so two functions run at once

Function 1 : set a value

```
DWORD WINAPI func1( LPVOID param )
{
   int i;
   volatile int local = 0;
   Sleep( 100 );

   pglobal = &local;

   for (i = 0; i < 10; i++ )
   {
      local = i;
      Sleep( 1000 );
   }
   return 0;
}
```

Wait

Store pointer

Change the value

Wait

Function 2: looking at the value

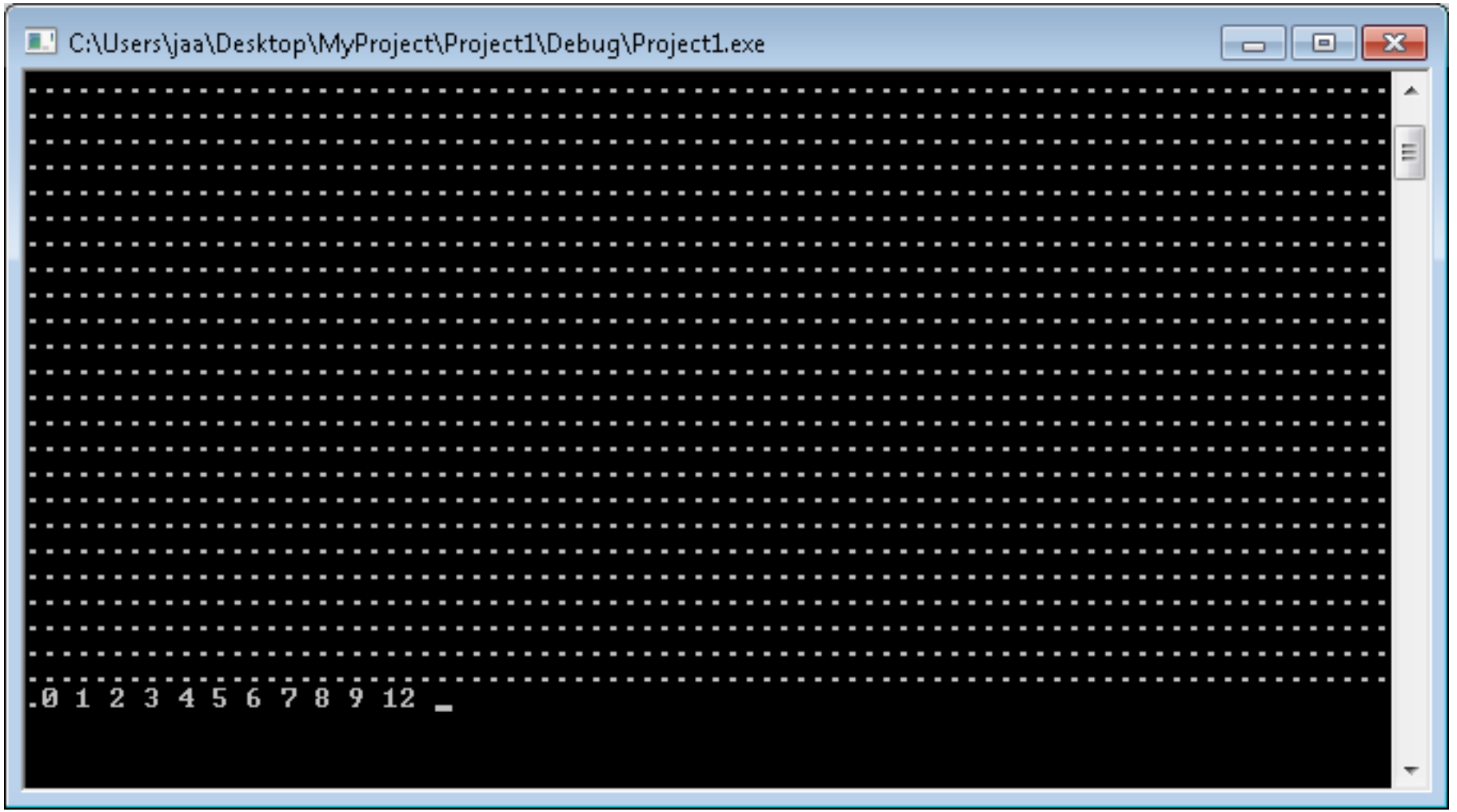
```
DWORD WINAPI func2( LPVOID param )
{
   int last = 0;
   while ( pglobal == NULL )
      printf( "." );
   last = *pglobal;
   printf( "%d ",last );
   while ( 1 )
   {
      while(*pglobal == last)  ;
      last = *pglobal;
      printf( "%d ",last );
   }
   return 0;
}
```

Loop until variable is set

Store last value

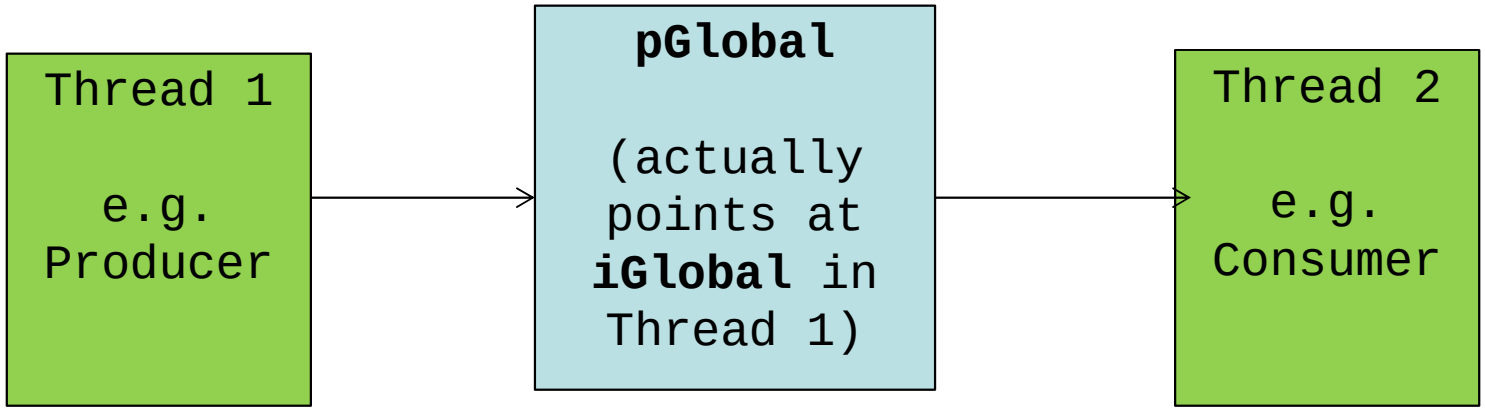
Loop until variable is changed

The Output


C:\Users\jaa\Desktop\MyProject\Project1\Debug\Project1.exe
.0 1 2 3 4 5 6 7 8 9 12

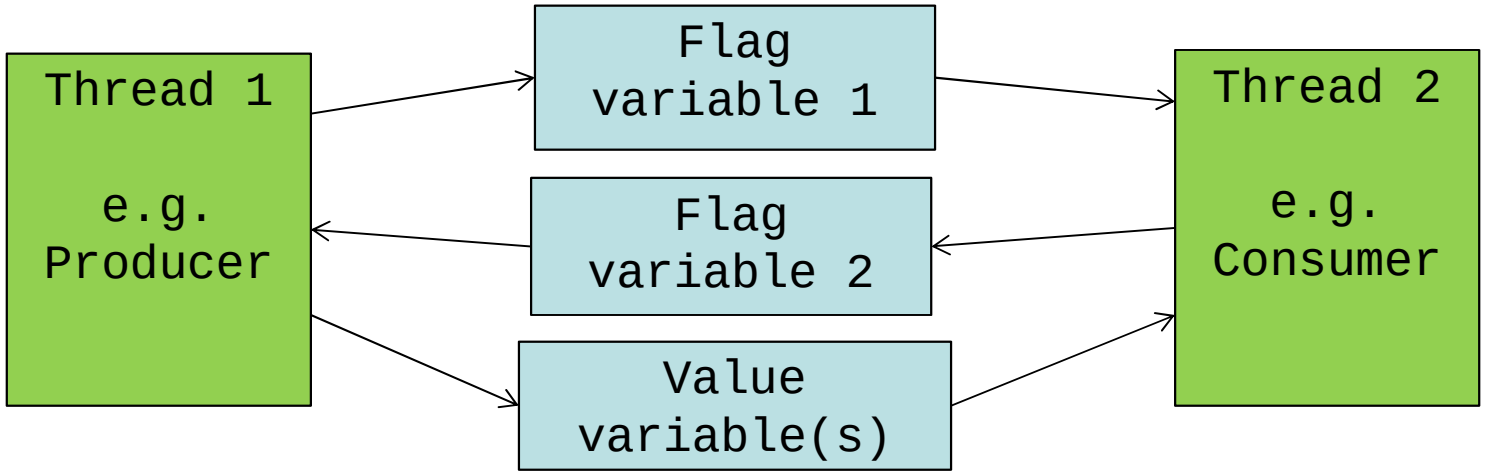
Aside: Coordination via variables

- In the previous example, one thread looked at a variable which the other changed (and did something when it did)
- Thread 1 changed the value
- Thread 2 read the value
- It seemed to work OK (initially) except:
  - No guarantee thread 2 would read it between updates
  - It was a race (called a race condition)
  - Also, it crashed when thread 1 ended – bad pointer



Aside: Coordination via variables

- Communicating by considering data values is a common way to synchronise threads or processes
  - Sending messages, events or signals is another
- We could use a separate variable to track the changes
- And the data could be multiple items
  - E.g. in a list or array : more about this later
- Can work well **if both cannot change it simultaneously**
  - If both can change it, there is a race condition



Address spaces for processes

- We will see spin-locks next lecture
  - They use this technique
  - Examine a variable and look for updates
- Not much use for separate processes?
  - Separate address spaces!
- Unless you can load the variables into both address spaces?
- Most modern operating systems actually allow you to do this
  - But it is done manually / explicitly

Simple way: use a struct for data

- Create a structure for the memory you want to share
  - Not necessary but will make the code easier for you, unless you are very good with pointer manipulation

```
struct MyDataStructure
{
      int iNumberProcesses;
      int iCount;
      char c;
      char str[128];
};
```

Create a mapping file

- Tell the system to create a (or open an existing) object for mapping a file structure into memory

```
HANDLE hMapFile = CreateFileMapping(
    INVALID_HANDLE_VALUE,           // use paging file
    NULL,                           // default security
    PAGE_READWRITE,                 // read/write access
    0, sizeof( MyDataStructure ),   // max size high,low
    "Local\\MyFileMappingObject" ); // name of object
```

- Linux: **`shm_open( SHARED_MEMORY_NAME, O_RDWR|O_CREAT|O_EXCL, S_IRUSR | S_IWUSR);`**

See: https://msdn.microsoft.com/en-us/library/windows/desktop/aa366537%28v=vs.85%29.aspx

Map memory into address space

- Load the *object* into the address space of each process
  - You are told the address, cast the struct pointer
- **Windows:**

```
volatile MyDataStructure * pMyData =
        (MyDataStructure *)MapViewOfFile(
        hMapFile,   // handle to map object
        FILE_MAP_ALL_ACCESS, // read/write permission
        0, 0,
        sizeof( MyDataStructure ) );
```

- **Linux:**

```
… = mmap( NULL, SIZE_OF_MEMORY, // Size needed
        PROT_READ | PROT_WRITE, // Read+write access
        MAP_SHARED, iFD, 0); // iFD was from shm_open
```

Use the memory

- We have a struct pointer to the memory

i.e. `volatile MyDataStructure * pMyData = (MyDataStructure *)MapViewOfFile`

- So we can access the members using ->
  - The -> operator replaces . when you have a pointer to the struct

```
printf( "I am process %d, there are %d processes and the total count is %d\n",
        iMyProcess,
        pMyData->iNumberProcesses,
        pMyData->iCount );
++pMyData->iCount;
```

Aside: there is a potential race condition since both could use the same number of iCount. However the update is not atomic anyway (next lecture)

Create a new copy of this process

```
// Work out the filename of the exe for process
char szMyFileName[1024];
GetModuleFileName( NULL,szMyFileName,1024 );
// We saw the create process before…
STARTUPINFO info = { sizeof( info ) }; // Input
PROCESS_INFORMATION processInfo; // Output
if ( CreateProcess( szMyFileName, // Program
      "", // Command line
      NULL,NULL,TRUE, CREATE_NEW_CONSOLE, NULL,NULL,
      &info, &processInfo ) )
      { // Created - close handles to it.
            CloseHandle( processInfo.hProcess );
            CloseHandle( processInfo.hThread );
      }
```

Tidy up at the end

- When you have finished, you need to free the resources that you mapped
  - Hopefully the OS is clever enough to do so when you end the process though

- Windows:

```
UnmapViewOfFile( (LPCVOID)pMyData );
CloseHandle( hMapFile );
```

- Linux:

```
munmap(map, SIZE_OF_MEMORY);
shm_unlink( SHARED_MEMORY_NAME );
```

Next lecture

- Atomic operations

- We will finally answer the question:
  - "Why, when I have three threads incrementing a value one million times each, is the final value not 3 million?"
  - And how to fix the problem