Synchronous Computations

Motivation

- Sometimes, we need threads to synchronize repeatedly with each other
- We can use a *barrier* to do so

C#

- C# provides a Barrier class
- Initialize:

```
1 Barrier B = new Barrier( count, (b) => { ... } );
```

  - The λ function will be executed when the final thread hits the barrier
  - It's passed the barrier itself as an argument
  - The threads don't resume until the λ function returns
- Signal:
  B.SignalAndWait();
  - You may not call this from the λ function! (Doesn't make sense anyway)

Pitfall

- All threads need to use the *same* barrier object. Bad:

```
new Thread( () => {
    Barrier B = new Barrier(3, (b)=>{} );
    ...stuff...
    B.SignalAndWait();
}).Start();

new Thread( () => {
    Barrier B = new Barrier(3, (b)=>{} );
    ...stuff...
    B.SignalAndWait();
}).Start();

new Thread( () => {
    Barrier B = new Barrier(3, (b)=>{} );
    ...stuff...
    B.SignalAndWait();
}).Start();
```

Better

```
Barrier B = new Barrier(3, (b)=>{} );
new Thread( () => {
    ...stuff...
    B.SignalAndWait();
}).Start();

new Thread( () => {
    ...stuff...
    B.SignalAndWait();
}).Start();

new Thread( () => {
    ...stuff...
    B.SignalAndWait();
}).Start();
```

Usage

- When is something like this useful?
- Consider the *merge sort* algorithm
  - Input: An array A of N elements
  - Create a temporary array B of N elements
  - Look at adjacent pairs of elements in A. Store them in the correct (sorted) order to corresponding slots in B
    - Ex: Compare A[0] and A[1]. If A[0] is smaller: Store A[0] to B[0].
    - Else, store A[1] to B[0]
    - Then store the other element to B[1]
- Do the same thing for A[2]/A[3], A[4]/A[5], etc.
- When we're done, B[0...1] is a sorted two-element array, as is B[2...3], B[4...5], etc.

Sorting

- Next, consider adjacent two-element arrays B[0...1], B[2...3]
- Sort these elements and store them back to A[0...3]
  - Look at B[0] and B[2]. Take smallest element, store to A[0]
  - ”Salami slicing” algorithm: Repeat until all 4 elements stored
- When we’re done, we have a bunch of 4 element sorted subarrays in A

Sorting

- Now, take adjacent pairs of 4-element arrays and salami-slice them again
- This will give a bunch of sorted 8-element arrays
- Repeat until we're all done

Example

- Suppose A is this:
  3, 1, 4, 1, 5, 9, 2, 6, 5, 3, 5, 8, 9, 7, 9, 3
- Pass 1: Sort pairs of 1-element subarrays to 2-element subarrays
  1,3, 1,4, 5,9, 2,6, 3,5, 5,8, 7,9, 3,9
- Pass 2: Sort adjacent 2-element subarrays to 4-element subarrays
  1,1,3,4, 2,5,6,9, 3,5,5,8, 3,7,9,9
- Pass 3: Sort adjacent 4-element subarrays to 8-element subarrays
  1,1,2,3,4,5,6,9, 3,3,5,5,7,8,9,9
- Pass 4: Merge these two 8-element subarrays
  1,1,2,3,3,3,4,5,5,5,6,7,8,9,9,9

Time

- How much time does this take?
  - Each time we do a round of merging, we're looking at all n elements of A
  - Each time we merge, we're doubling the number of elements in each subarray and thus halving the number of subarrays
  - n subarrays at first, then n/2, then n/4, then n/8, ...
  - Denominator is 2^0, then 2^1, then 2^2, ...
- When does denominator == n?
  - When 2^i == n
- What is i? Take lg of both sides: i = lg n
- So lg n iterations, each one of time n, so O(n lg n) time

Example

- Example implementation: mergeSort.cs

Parallelization

- We can do the merges in parallel
- But: We must stop after each ”turn of the crank” so that each round is fully done before we move on to another round
- Note that (for an array of n elements), we can have n/2 threads active (max) on the first iteration, but then n/4 and then n/8 and then n/16, ...
- Amount of available parallelism goes down as we progress!

Code

- Example implementation: mergeSortParallel.cs

Assignment

- Implement parallel blur
  - Your program should take a three command line arguments (via Main's args[] array): The name of an image file, the number of worker threads, and the number of rounds of blurring to perform
  - Load the image and perform the specified number of 5x5 box blurs of the image
  - Output the results to the file "out.png"
  - Do the blurring concurrently and correctly
  - Output the total time required to the console

Code

- To load an image:

```
using System;
using System.Drawing;
using System.Drawing.Imaging;
using System.Runtime.InteropServices;
...
Bitmap img = (Bitmap) Image.FromFile(filename);
```

Code

- To convert an image to a byte array so it can be examined:

```
1 var bdata = img.LockBits(new Rectangle(0,0,img.Width,img.Height),
      ImageLockMode.ReadWrite, PixelFormat.Format24bppRgb );
2 byte[] pix = new byte[bdata.Stride*bdata.Height];
3 Marshal.Copy(bdata.Scan0, pix, 0, pix.Length );
```

- The pix array will have the image data in BGR order, one row after another
- Each row is Stride bytes long

Code

- To save a modifed copy of the image:

```
1 Marshal.Copy(pix, 0, bdata.Scan0, pix.Length );
2 img.UnlockBits(bdata);
3 img.Save("out.png");
```

References

- http://www.vcskicks.com/image-to-byte.php

Created using LaTeX.

Main font: Gentium Book Basic, by Victor Gaultney. See http://software.sil.org/gentium/
Monospace font: Source Code Pro, by Paul D. Hunt. See https://fonts.google.com/specimen/Source+Code+Pro and http://sourceforge.net/adobe
Icons by Ulisse Perusin, Steven Garrity, Lapo Calamandrei, Ryan Collier, Rodney Dawes, Andreas Nilsson, Tuomas Kuosmanen, Garrett LeSage, and Jakub Steiner. See http://tango-project.org