Concurrency

Large Scale Programming, 1DL410, autumn 2009
Cons T Åhs
Multi core and threads

- Processors with multiple cores are standard today.
  - Only low end machines ship with single core processors.
  - Dual core is more or less standard.
  - Four cores is becoming more common.
  - 8 or 16 cores are present in high end machines with multiple processors and/or hyper threading.

- For a developer this means that you cannot ignore the problem of parallelising your program. For a computationally intensive program, users will complain if they see that the program is only using a single core of the processor.

- Ray tracing is an extremely computationally intensive problem.

- Can it be parallelised?
  - Yes!
  - It is actually quite simple and ray tracing is sometimes called an embarrassingly simple problem to parallelise.
  - Done right, making your ray tracer utilise multiple cores is deceptively simple.

  - All problems are not this simple to parallelise!
Parallel Ray Tracing

- Ray tracing an image of NxM pixels means that we generate NxM primary rays.
- The computation of the colour of one primary ray does not affect the colour of any other ray.
  - We have independent computations.
  - Given enough cores/processors we could do all NxM colour computations in parallel.
- Resources shared between ray colour computations:
  - The result of each computation, i.e., a colour, must be stored in the image.
    - Each primary ray will write in different pixels.
    - Only when all primary rays are computed will the image be read.
    - The resulting image is a shared write only resource during the computation.
  - Each computation will need access to the internal world representation, but (ideally) nothing will be written to it, i.e., there will be no modifications of the world or the objects in it.
    - The world is a shared read only resource.
- You can expect close to linear speed up.
Parallel Ray Tracing

- Assume we have $P$ cores available. For full utilisation we should be running $P$ threads.

- Trivial algorithm:
  - Divide the image in $P$ parts, e.g., by striping the image.
  - Create one thread for each part.
  - When all threads have finished, write the resulting image to disk.

- Very simple to implement.

- A drawback is that the computation might be spread unevenly between the threads as a consequence of different object density.
  - Some threads will finish fast and leave cores idling.
  - Creating more threads than cores, say $nP$, $n > 1$, will make the problem smaller but not go away. Also, it will lead to more unnecessary thread scheduling and overhead.
Parallel Ray Tracing

- Assume we have $P$ cores available. For full utilisation we should be running $P$ threads.

- Better algorithm:
  - Divide the image in $K$ parts. $K$ might be larger than $P$.
  - Create $P$ threads and let each thread work on one part.
  - When a thread is finished, let it check if there is more to do, either by grabbing a new unstarted part or taking a part from another thread.
  - When all parts are done, the image is written to disk.

- We divide the image in parts regardless of the number of cores available - we can call these parts jobs.

- Each thread, or worker, takes a job from a queue and then, when finished, takes a new job.

- Better utilisation of available cores.

- A bit more complicated to implement, but still rather simple.
Process vs. Thread

- What is the difference between multiple processes and multiple threads?
- Both processes and threads facilitate doing things at the “same time”, pending scheduling and the number of processors/cores available.
- A process has its own memory space, so communication between processes must be done by passing messages in some way.
- A thread shares memory with other threads, so communication takes place when memory changes.
- A process can contain multiple threads.
- Each thread has a private state, such as stack, program counter and local variables.
- All threads in a process share the same memory and thus all have access to the same object instances etc.
- This means that two threads can access and modify the same object at the same time.
- Compare with several persons working on the same physical project such as repairing a car. The actions must be communicated and coordinated.
General Problems

- In general, utilising concurrency is not this simple.
- The root of the problem comes from having a shared mutable resource.
  - shared - several threads may have access to the resource.
  - mutable - the resource can change state.
- There are no problems if the resource is not shared or is immutable.
  - For ray tracing, the world is immutable (at least it should be).
  - The result image is shared and mutable, but very simply.
- The problem is related to the number of threads, not the number of cores.
- Consider the classic example of a bank account
  - internal state is the amount of money, $m$.
  - operations:
    - withdraw($x$) : $\text{if } (m \geq x) \ m = m - x$;
    - deposit($x$) : $m = m + x$;
  - two threads changing the account at the same time might lead to inconsistent state of the account, i.e., holding the incorrect amount of money.
Solution

- To avoid the problems (there are several) associated with having a shared mutable state, one needs to assure that only one thread at a time can change the state.

- This is done by locking the object by using synchronisation.

- In Java
  - every object can be used as a lock
  - methods can be declared as `synchronized`, thereby locking the whole object when the method is being executed.
  - an unsynchronised method can be executed as the same time as a synchronised method - take care to synchronise all methods that need it.

- Note that locking everything certainly is safe, but might degrade your program to single threaded performance (depending on the problem). You need to think.

- There are many more problems associated with threads and they can be solved in many different ways as well, both in general and in Java. The exact details are beyond the scope of this course.
Practical Issues

- Use synchronized either on methods or on objects for locking and providing safe change and access.

- For using threads in Java, check out, at least, the following:
  - interface Runnable - needed to encapsulate a computation in a thread.
  - class Thread - to be able to create a thread
    - method start() - start a thread.
    - method join() - wait for a thread to finish.