Reminder

Reminder

You should now have handed in part 2 of your logbook.

Concurrent Systems

January 15, 2019

- This semester's course
- Introduction to Concurrent Systems
- Concurrent processes in Java

This Semester

1: This semester's course

1.1: Concurrent systems

1.1.1: Properties of concurrent systems

- Critical sections
- Mutual exclusion
- Deadlock
- Starvation
- Liveness
- Loosely connectedness

1.1.2: Tools for concurrent systems

- Shared variables
- Semaphores
- Monitors

Quantum computing

1.2: Quantum computing

- Circuits as matrices and vectors
- Quantum systems
- Quantum circuits
- Quantum algorithms

Theoretical aspects

1.3: Theoretical aspects

- Correctness
- Complexity

Outcomes

1.4: Outcomes

1. Discuss the classification of algorithms accoroding to efficiency and complexity
2. Prove code correct
3. Demonstrate a knowledge of the characteristics of a range of concurrency paradigms
4. Explain the difference between classical and quantum computing
5. Use a standard notation to analyse the efficiency and complexity of algorithms

1.5: Lecture Plan

Week	Topic
13	Introduction
14	Dekker's Algorithm
15	Semaphores
16	Monitors
17	Modelling Circuits
— Guidance Week —	
19	Quantum Systems
20	Quantum Computing
21	Correctness
22	Complexity
23	Overspill/recap
24	

2: Introduction to Concurrent Systems

2.1: Why programme concurrent systems?

- Because they are efficient.
  Deterministic polynomial *vs.* nondeterministic polynomial
- Because they simplify programming.
  GUIs
- Because you have to.
  Operating systems.

Merge sort

2.1.1: Efficiency
2.1.1 A: Sequential merge sort
2.1.1 A(i): Algorithm

```
public void mergeSort() {
    int half; Sort left,right;
    if (size > 1) {
        half = size/2;
        left = new Sort(list,0,half-1);
        right = new Sort(list,half,size-1);
        left.mergeSort(); right.mergeSort();
        merge(left,right);
    }
}
```

Complexity

2.1.1 A(ii): Complexity

- Assume `merge` of **N** items takes **N** "time units" **t**.
- How many merges?

$$
\mathbf{n}\left\{\begin{array}{ll} \mathbf{1} \text{ merge} & \text{each } \mathbf{N} \\ \mathbf{2} \text{ merges} & \text{each } \frac{\mathbf{N}}{\mathbf{2}} \\ & \\ \mathbf{2^{n-1}} \text{ merges} & \text{each } \frac{\mathbf{N}}{\mathbf{2^{n-1}}} \\ \mathbf{2^{n}} \text{ merges} & \text{each } \frac{\mathbf{N}}{\mathbf{2^{n}}} \end{array}\right. \quad \vdots \quad \begin{array}{|cccc} \mathbf{1} & \times & \mathbf{N} & = \mathbf{Nt} \\ \mathbf{2} & \times & \frac{\mathbf{N}}{\mathbf{2}} & = \mathbf{Nt} \\ & & & \\ \mathbf{2^{n-1}} & \times & \frac{\mathbf{N}}{\mathbf{2^{n-1}}} & = \mathbf{Nt} \\ \mathbf{2^{n}} & \times & \frac{\mathbf{N}}{\mathbf{2^{n}}} & = \mathbf{Nt} \end{array}
$$

So $\mathbf{n} \times \mathbf{Nt}$. What is $\mathbf{n}$?

Complexity

2.1.1 A(iii): Complexity

- Assume `merge` of $\mathbf{N}$ items takes $\mathbf{N}$ "time units" $\mathbf{t}$.
- How many merges?

$$
\mathbf{n}\left\{
\begin{array}{cccc|cccc}
1 \text{ merge} & \text{each } \mathbf{N} & = 2^{n} & & 1 & \times & \mathbf{N} & = \mathbf{Nt} \\
2 \text{ merges} & \text{each } \frac{\mathbf{N}}{2} & = 2^{n-1} & & 2 & \times & \frac{\mathbf{N}}{2} & = \mathbf{Nt} \\
 & & \vdots & & & & & \\
2^{n-1} \text{ merges} & \text{each } \frac{\mathbf{N}}{2^{n-1}} & = 2 & & 2^{n-1} & \times & \frac{\mathbf{N}}{2^{n-1}} & = \mathbf{Nt} \\
2^{n} \text{ merges} & \text{each } \frac{\mathbf{N}}{2^{n}} & = 1 & & 2^{n} & \times & \frac{\mathbf{N}}{2^{n}} & = \mathbf{Nt}
\end{array}
\right.
$$

So $\mathbf{n} \times \mathbf{Nt}$. What is $\mathbf{n}$? $\mathbf{2^n = N \Rightarrow n = \log N}$.

- So (sequential) mergesort $\mathbf{tN \log N}$.

Parallel merge sort

2.1.1 B: Parallel merge sort
2.1.1 B(i): Algorithm

```
public void mergeSort() throws InterruptedException {
    int half; Sort left,right; // Note: Sort extends Thread
    if (size > 1) {
        half = size/2;
        left = new Sort(list,0,half-1);
        right = new Sort(list,half,size-1);
        left.start(); right.start();
        left.join();  right.join();
        merge(left,right);
    }
}
```

Complexity

2.1.1 B(ii): Complexity

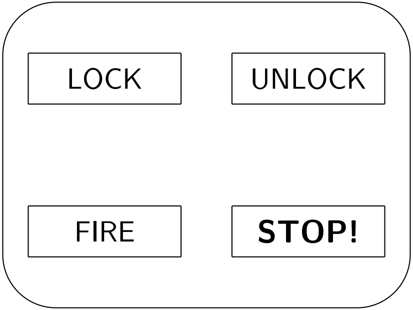
- Merges at each level can be executed in parallel

$$
\begin{array}{rcrcl|cccl}
1 \text{ merge} & \text{each} & \mathbf{N} & = & 2^n & 1 \times & \mathbf{N} & = & \mathbf{N}\ \mathbf{t} \\
2 \text{ merges} & \text{each} & \frac{\mathbf{N}}{2} & = & 2^{n-1} & 1 \times & \frac{\mathbf{N}}{2} & = & \frac{\mathbf{N}}{2}\ \mathbf{t} \\
 & & & \vdots & & & & & \\
2^{n-1} \text{ merges} & \text{each} & \frac{\mathbf{N}}{2^{n-1}} & = & 2 & 1 \times & \frac{\mathbf{N}}{2^{n-1}} & = & \frac{\mathbf{N}}{2^{n-1}}\mathbf{t} \\
2^n \text{ merges} & \text{each} & \frac{\mathbf{N}}{2^n} & = & 1 & 1 \times & \frac{\mathbf{N}}{2^n} & = & \frac{\mathbf{N}}{2^n}\ \mathbf{t}
\end{array}
$$

So $\mathbf{\Sigma}_{i=0}^{n} \frac{\mathbf{N}}{2^i} = \mathbf{N} + \frac{\mathbf{N}}{2} + \frac{\mathbf{N}}{4} + \cdots + 1 = 2\mathbf{N}$

- So (parallel) mergesort: **2Nt**.

2.1.2: Simplification



Sequential

```
while (true) {
    LOCK.listenTo();
    UNLOCK.listenTo();
    FIRE.listenTo();
    STOP.listenTo();
}
```

Parallel

```
while (true) {
    LOCK.listenTo() ||
    UNLOCK.listenTo() ||
    FIRE.listenTo() ||
    STOP.listenTo() ||
}
```

Necessity

2.1.3: Necessity
Operating Systems

- I/O devices
- Interrupts
- Multi-tasking
- Networks

Aspects of concurrent systems

2.2: Aspects of concurrent systems

Note: A concurrent system is not necessarily truly parallel — timeslicing, interleaving.

2.2.1: Necessary tools

- Communication
- Synchronisation

2.2.2: Properties

- Complexity
- Correctness
- Granularity

Java

3: Concurrent processes in Java

3.1: Defining process classes

A parallel process is an instance of a `Thread` — a `Thread` runs a `Runnable`.

- Either implement the `Runnable` class

  ```
  class Process implements Runnable {...}
  ```

- or extend the `Thread` class

  ```
  class Process extends Thread {...}
  ```

run

3.2: Defining process behaviour

```
public void run() {
    ...
}
```

Creating a Process

3.3: Creating a process

- From a subclass of `Thread`

```
Process process = new Process();
```

- From an implementation of `Runnable`

```
Thread thread = new Thread(new MyRunnable());
```

Note: this does *not* start the thread running.

Starting and Stopping Threads

3.4: Starting a thread

```
myThread.start();
```

Note: do *not* call `run()`.

3.5: Waiting for a thread to stop

```
try {
   myThread.join();
}  catch (InterruptedException e) {} ;
```

Sharing Data Betwwen Processes

3.6: Sharing data between processes

- a non-static variable is unique to the instance

```
int belongsToPooh;
```

- a **static** variable is shared by all instances of the class

```
static int botherItsPigletsToo;
```

Some Useful Methods

3.7: Some useful methods
3.7.1: Access

- `someThread.checkAccess()`
  Is the currently running thread allowed to modify `someThread`?
- `someThread.getId()` (returns a **long**)
- `someThread.getName()` (returns a `String`)

Some Useful Methods

3.7.2: Control

- `join`:
  - `someThread.join()`
    Wait for `someThread` to die
  - `someThread.join(millis)`
    Wait at most `millis` ms for `someThread` to die (millis is **long**)
- **static void** `sleep(millis)`
  Currently executing thread sleeps for `millis` ms.
- **static void** `yield()`
  Currently executing thread temporarily allows another thread to execute.

Priorities

3.7.3: Priorities

- **static void** `setPriority(`**int** `newPriority)`
- **int** `getPriority()`
- `MAX_PRIORITY`, `MIN_PRIORITY`, `NORM_PRIORITY`