DynamiTE

A Framework for Concurrent Component-Based Design

Andrew John Hughes

Agenda

- Motivation
- Existing Methods
- Making Applications Task-Centric
- DynamiTE
- Demo
- Conclusion

The End of Moore's Law?

- **The Past**
- New computer, same software – (much) *faster*
- **Now...**
- New computer, same software – *may be faster*

Sequential Models

- Our models are still *inherently sequential*
- What happens to the Turing machine when there are multiple heads operating at the same time?

Resource Sharing

- Let's write a print spooler...

```
1.int fd =
  open(“/var/lib/print_jobs”);
2.seek(fd, END_OF_FILE);
3.write(fd, “my_print_job”);
4.close(fd);
```

Resource Sharing

```
1.int fd =
  open(“/var/lib/print_jobs”);
2.flock(fd, LOCK_EX);
3.seek(fd, END_OF_FILE);
4.write(fd, “my_print_job”);
5.flock(fd, LOCK_UN);
6.close(fd);
```

Existing Methods: Semaphores

- Has a *count* which is modified **atomically**
- Performing down decreases the count
- Performing up increases the count
- With a count of one, we have **mutual exclusion** (*a mutex*)

A Producer

```
1.item = produce_item();
2.down(mutex);
3.down(empty);
4.add_item_to_buffer(item);
5.up(mutex);
6.up(full);
```

A Producer

```
1.item = produce_item();
2.down(empty);
3.down(mutex);
4.add_item_to_buffer(item);
5.up(mutex);
6.up(full);
```

Problems

- See how easy it is to get this wrong?
- An up or down may be missed
- With multiple related semaphores, the order of statements becomes *crucial*
- It takes one miscreant to ruin everything
- Not always reproducible; it's all about *timing*
- And we're making everything **sequential** again

Existing Methods: Monitors

- A little better...
- We mark sections of code which must run in mutual exclusion
- Fit nicely with *objects*
- **But...** language dependent
- Still very prone to error

Producer – With Monitors

- Object item = produceItem();
- synchronized {
- if (used == BUFFER_SIZE)
- wait();
- buffer[used] = item;
- ++used;
- notifyAll(); }

Producer – With Monitors

- Object item = produceItem();
- synchronized {
- while (used == BUFFER_SIZE)
- wait();
- buffer[used] = item;
- ++used;
- notifyAll(); }

Back to the Drawing Board

- We are too concerned with protecting *resources*, especially *data*
- Our designs are *data-centric*
- Instead we need to:
  - Focus on minimal sequential *tasks*
  - Let the data flow along the pipes rather than being the centre of our universe
- Think pipelines e.g. `du -h | sort -n`

The Library System: Data-centric

- Focus is on *data objects*
  - **Borrower**
  - **Book**
- Tasks are *methods* of these objects
- But who runs them? What is the control flow?
- And how is concurrent access handled?
- **If at all?**

The Library System: Task-centric

- Focus on *tasks:*
  - Borrowing a book
  - Reserving a book
- Simple sequential tasks with no shared storage
- Long term storage can be managed by a *database guardian*
- Do it once and do it well

DynamiTE

- Design the application as self-contained sequential tasks which communicate with one another
- **You** write the tasks
- **DynamiTE** provides the plumbing
- Communication grounded in a *process calculus*

How It Works

- `Process` hierarchy provides *operational semantics*
- `Evolvers` provide *execution semantics*
- Transitions can have **side effects**
- Realised by *plugins*
- Plugins maintained by the `Context`

Demo

- The 'Hello World' of DynamiTE
- One task produces a message
- Another retrieves and prints it
- DynamiTE conveys the message

Conclusion

- Need to treat concurrency less as an *optional extra* and more as an *essential component* to fully utilise the performance of new machines
- Many existing concurrency techniques are just too **low-level**
- DynamiTE makes things easier...
- ... but still need to rethink our designs.

Get It Now!

- https://savannah.nongnu.org/projects/dynamite/
- Patches welcome!
- Post-viva drinks: 5pm, Monday 19th October in the Cavendish