ECE 3574: Applied Software Design

Actor Pattern

Today we are going to look at an abstraction of concurrent tasks called Actors.

- Actors, Mailboxes, and Controllers
- A simple C++ Actor system

Actor Pattern

The actor pattern uses concurrent entities, named *actors*, that do not share state and communicate only via message passing.

This is similar to producer consumer except that actors can create, or *spawn*, other actors.

This makes the communication requirements more complex, since messages must now have a address-like field so that messages can be sent to specific actors.

Actors have a few advantages and disadvantages

- scale to many cores with less effort
- actors can also work across machines, they can be *distributed*

The major drawback is in the time and code complexity of the message passing/delivery

There are actor-based languages, notably Smalltalk and Erlang.

Actors have an address/handle/id

This uniquely identifies the actor among all others.

You can send a message to an actor given its id.

Think “email address” for the object.

See `actor_id.h`

Actors exchange messages

These may be typed or untyped.

Messages can correspond to methods of the actor.

However, an actor may or may not respond to a given message.

See `message.h`

Since messages have receivers they have to be collected/dispatched centrally.

This is called the *mailbox* for the actor.

Sending a message to an actor's id queues the message in it's mailbox.

Lets use our thread-safe queue as the mailbox: see `message_queue.h` and `actor_base.h`

Defining an abstract actor

An Actor

- knows its own ID
- can do work
- can receive messages
- can exit

See `actor_base.h`

We need to guarantee a unique central controller for the actors.

Singleton Pattern: a *singleton* is an object of which there can only be one.

Similar to a global object, but somewhat safer.

It has limited uses, but a thread pool / actor controller is one.

See `singleton_ex.cpp`

Lets define a Controller as a singleton

Define the type of actors allowed

Add methods:

- spawn
- send
- exit
- wait_for_actors

We need to postpone the details of creating actors until they are defined, so we use the pointer-to-implementation idiom (PIMPL).

See `controller.h`/`controller.cpp`

Now we can define some example actors

These inherit from `ActorBase`.

Override `operator()` to specify the actors behavior.

See `actors.h`

Now we can implement the Controller

Use the factory design pattern to build actors.

Implement:

- spawn
- send
- exit
- wait_for_actors

See `controller_impl.h/controller_impl.cpp`

Finally we can implement main

Launch the first actor and wait for all actors to finish.

See `simple_actor_ex.cpp`.

Next Actions and Reminders

- Read about heap management