UNIVERSITY OF WATERLOO
Department of Electrical and Computer Engineering

Concurrency in C++ Using Expressions

Christopher Trudeau
ctrudeau@etude.uwaterloo.ca

Parallel and Distributed Systems Group

slides by *SlickSlides*

Outline

1. Concurrent Computing Review
2. Active Expressions Model
   (a) Objects and Data
   (b) Operators
   (c) Shared & Distributed Memory
   (d) Shared Regions & Synchronization
3. Implementation
   (a) Template Usage
   (b) Communication
   (c) Threads
   (d) Distributed Memory Concerns

Concurrency & Processes

- want to increase speed of computation
  - increase the number of processors
  - divide work between processors
  - *should* result in decreased computation time
- need to balance amount of communication & computation: **granularity**
- UNIX originally provided concurrency through "processes"
- a process is typically defined as a concurrent with its own address space
- operating systems like UNIX protect processes from each other
  - any attempt from a process to write into another process' address space results in a "core dump"
- any command can be started as a *background* process by placing a "&" at the end of the command line

Coding a UNIX Process

- executing a program implicitly creates the first process
- a process can create another process by calling the `fork()` library function
- `fork()` creates an *identical* copy of the current process and returns either the created process' id (to the parent) or 0 (to the child)
- `fork()` typically used with the `exec()` and *join()* commands

```
pid = fork();
if( pid == 0 )
{
    execl( "some-prog", "arg1", "arg2", NULL );
    printf( "Shouldn't get here\n" );
} // end if

// parent continues here with some parallel code
...
join();             // parent waits for child
```

Concurrency & Threads

- a process has a lot of creation overhead — not good for a lot of dynamic creation
- **threads** (a.k.a. light-weight processes) are concurrent entities within a process
- typically share process' address space
- operating system is now responsible for scheduling processes and threads within a process
- typically associated with a function — thread is created for the duration of the function call
- introduces a large number of concurrency issues due to shared address space
  - synchronization of threads
  - memory sharing between threads
  - communication between threads

Coding a UNIX Thread

- in UNIX you first create a thread attributes structure
- there is a library function for getting and setting each thread attribute (defaults are underlined):
  - scope: competes for resources at system or process level
  - detach state: whether a thread is "joinable" or not
  - stack size: a specific or system determined stack size
  - stack address: a specific or system determined starting stack address
  - schedule policy: FIFO, round-robin, other — Solaris only supports "other"
  - schedule parameter: a specified scheduling parameter — usually the thread's priority or NULL to indicate inheritance of priority
  - inherited schedule: whether a thread's scheduling policy must be explicitly defined or it is inherited
- create the thread itself using a pointer to a function for the thread to run
- changes to the attribute after the creation of the thread do not affect the previously created thread(s)

Coding a UNIX Thread — Example

```
void foo( void *arg );                  // function prototype
char string[] = "parameters";

// declare the thread structures
pthread_t thread;
pthread_attr_t thread_attr;

// initialize the thread attributes structure
pthread_attr_init( &thread_attr );

// change the stack size
pthread_attr_setstacksize( &thread_attr, 100000 );

// create the thread
pthread_create( &thread, &thread_attr, foo, (void *)string );
```

Introduction to Active Expressions

- C++ toolkit approach
- provides type-safe concurrent programming
- portable
- abstracts the concurrency model without restricting the developer
- does not require language extensions or pre-processors
- does not require a graphical tool
- originally created by Mauricio De Simone and Ajit Singh in 1996

Objects and Data

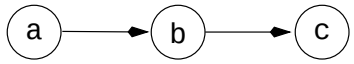
- an Active Expression is made up of Active Components
- Active Components are instances of user defined classes which inherit from toolkit abstract classes
- combinations of Active Components create the concurrency and communication pattern

```
class Producer : O_Ac<int> { ... };
class Worker   : IO_Ac<int,float> { ... };
class Consumer : I_Ac<float> { ... };
⋮
int i = 5;
C_Ae ae = producer | (i * worker) | consumer;
```

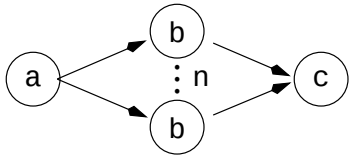
- type safety through templates
- communication through streams
- virtual functions *run()* and *clone()* for instantiation

Operators

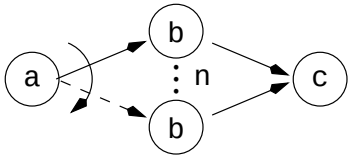
- various operators combine Active Components to create Active Expressions
- **Pipe "|":** competitive queue between components
- **Broadcast "||":** replicated-message queue
- **Replication "∗":** clones components
- **Object Operators:** user defined; e.g. *cycle()*
- operator precedence still valid – use "(. . .)"



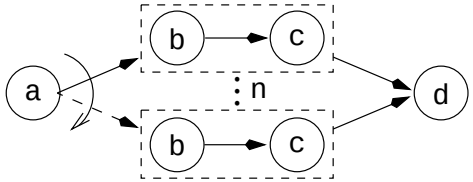
a | b | c



a | (n * b) | c



a || (n * b) | c



a || (n * (b | c) | d

Streams

- operators which instantiate communication create streams
- two stream types exist: `pin` and `pout`
- an Active Component inherits from one of the toolkit defined: closed, input, output, or input/output classes
- each class has a stream member
- input classes have a `pin` stream
- output classes have a `pout` stream
- when inherited from, Active Components require a template instance
- template instance determines what can be passed on the stream
- compiler generates an error if an incorrect object type is passed to the stream

Memory & Synchronization

Shared & Distributed Memory

- both memory models are supported
- coding is identical for both, compile flag indicates model
- placement in distributed memory defaults to round-robin but can be controlled via command line parameters

Shared Regions & Synchronization

- Shared Regions allow global shared variables through readers/writers protocol
- identical interface for shared and distributed memory architectures
- template based mechanism ensures type safety
- Mutex objects provide locking mechanism for synchronization

Swarm

- create a single queen and multiple worker bees
- worker bees try to follow queen around the screen
- typical task farm setup
  - queen is producer – produces position value as she moves
  - multiple workers fight over the consumption of the queen's position
- workers continue along their current path until they receive a change of position "message" from the queen
- can give a quantitative estimate of a platform's scheduling fairness
- Active Expression:

```
Queen queen( duration, velocity );
Worker worker( velocity );

C_AeHandle h = queen | (n * worker);
```

The "Worker" Class

```
class Taunt {
public:
    double x, y;
};

class Worker : public I_Ac<Taunt> {
private:
    Bee bee;     // used to store bee's position and velocity
public:
    Worker( double velocity ) : I_Ac<Taunt>(), bee( velocity ) {}
    virtual void execute ();
    virtual Ae* clone () { return new Worker( *this ); }
    virtual size_t size () { return sizeof( *this ); }
};

void Worker::execute () {
    Taunt taunt;
    while( pin ) {
        if( pin.attempt( taunt ) )
            bee.headTowards( taunt.x, taunt.y );

        bee.Draw();         // draw, then wait for some time interval
    }
} // end Worker::execute
```

The "Queen" Class

```
class Queen : public Bee, public O_we<Taunt>
{
private:
    const double duration;

public:
    Queen( double d, double velocity ) : O_we<Taunt>(), Bee( velocity ),
        duration( d ) {}

    virtual void execute ();
    virtual Ae* clone () { return new Queen( *this ); }
    virtual size_t size () { return sizeof( *this ); }
}; // end class Queen

void Queen::execute () {
    Taunt taunt;

    while( duration >= 0 ) {
        taunt = newPosition();

        pout << taunt;
        duration --;
    } // end while
} // end Queen::execute
```

Implementation

- recursive evaluation of expressions
- use of C++ "=" operator for constructor parameters
- binary tree of operands is created to represent Active Expression
- each Active Component is a thread using the components over-ridden *run()* method
- factories and engines are used to abstract the communication model
- switching from shared to distributed memory architectures is done through instantiation of the correct factory
- the Active Expression thread waits until all of its children Active Component threads terminate

Template Usage

- fundamental to the type-safety of the language
  - most other concurrent approaches use a run-time engine
  - time safety is expensive to check at run time
  - often ignored in most implementations
- type safe mechanism exists between Active Components and within locks for Shared Regions
- template based approach means type errors are caught at compile time
- trade-off between compile time and performance

Communication

- communication engine establishes whether simple queues or socket based communication is required at instantiation
- objects see input and output streams and are ignorant of where the data is going or how it is being used
- queue code is not UNIX IPC for portability reasons
- abstract class Streamable includes methods for sending class instantiations across a socket
- default Streamable functions are shallow but they can be over-ridden
- communication engine provides a mechanism for remote execution of an object's method
  - provides rudimentary RPC
  - used extensively in toolkit to perform start up, Shared Region, and synchronization functions

Threads

- in distributed memory architecture each machine runs the same executable with different command line parameters to distinguish between master and slave copies
- factory pattern establishes what thread type to use
- compile switches indicate platform and thus thread library
- thread code is abstracted so that addition of new thread libraries has no effect on existing code
- instantiation of remote threads is done using the communication engine's remote command feature
- thread handles are kept in a vector tracked by the master copy of the executable

Distributed Memory Concerns

- seamless implementation of Shared Regions for both shared and distributed memory architectures is difficult
- corresponding components created on different machines may not have the same physical addresses
- some tag is necessary to identify shared components
- could use identifier passed in by user – not type safe and must be done at run-time
- use automatically generated identifier based on order of instantiation
- order of instantiation is identical since the same piece of code is being run in the master and slave copies
- uses a static templated link list to store the identifiers