High-Level Concurrency for C++

Dr. Peter A. Buhr

http://plg.uwaterloo.ca/~usystem/uC++.html

Research Sponsors

C++

- C++ has the potential to become the next dominant programming language. (Java killer)
- Why?
  - based on C, which has a large programmer and code base,
  - as efficient as C in most situations,
  - direct access to memory, which is necessary for:
    * systems programming
    * memory management
    * real-time
  - high-level features, e.g., objects, polymorphism, exception handling, STL, etc., which programmers now demand.

Concurrency

- Concurrency is thrust upon us to attain higher performance.
- To maintain Moore's law, parallelism is added to the hardware [1].
- Concurrency is significantly more complex than sequential programming.
- Some success in *implicitly* discovering concurrency in sequential programs
- Fundamental limits to finding parallelism and only for certain kinds of problems.
- Alternatively, programmers *explicitly* think about and specify the concurrency.
- Implicit and explicit approaches are complementary, i.e., can appear together.
- However, the limitations of implicit approach mandate an explicit approach.

Concurrency in C++

- C++ has no concurrency!
- Many different concurrency approaches for C++ have been implemented with only varying degrees of adoption.
- As a result, no de facto approach dominating concurrent programming in C++.
  - C has two dominant but incompatible concurrency libraries: Win32 and pthreads.
- C++'s lack of concurrency is limiting its future.
- Therefore, it is imperative C++ be augmented with concurrency to maintain its programming base.

Outline

- Examine adding concurrency to C++:
  - concurrency has to be part of a programming language,
  - high-level concurrency is better than low-level,
  - language's design principles strongly direct (constrain) the possible concurrency models.
- Look at specific design/implementation called μC++.

Elementary Concurrency Properties

- All concurrent systems must have:

  1. **thread** : independent sequential execution that modifies state

  2. **execution context** : state needed to start and execute a thread (stack/PC)
     context switch is a thread switching between execution contexts.

  3. **mutual exclusion** / **synchronization** (ME/S) : exclusive access to a resource and timing relationships among threads.

- Properties 1 & 2 cannot be expressed in an architecture-independent way.

- Property 3 can be expressed by simpler concepts (Dekker/flags) but error-prone and inefficient (may need concurrent memory model).

Low-Level : Concurrency Library

- All current approaches to introduce concurrency into C++ use a *library approach*.
- However, a library approach is either unsound or inefficient [2, 3, 4, 5].
- The reasons are straightforward:
  - no concurrency means the compiler treats all programs as sequential
  - valid sequential optimizations may invalidate a concurrent program.

Relocating values	*Code reordering*	*Instruction substitution*

flag = 0; *// shared*
task$_1$: **while** (flag == 0); . . .
task$_2$: flag = 1; . . .
	acquire(lock)
	// critical section
	release(lock)
		written
		lw $sp,PrivateStack
		substituted
		lui $sp,4097
		lw $sp,–15288($sp)

- The solution is to turn off all optimizations affecting concurrency correctness.
- Impractical:
  - which optimizations cause problems?
  - sequential programs and the sequential portions of concurrent programs may run much slower.
- Therefore, achieving sound and efficient concurrency mandates it be part of a language's definition.
- *No alternative exists!*
- C++ standards committee is currently attempting to extend C++ with a concurrent memory-model.

Medium-Level : Basic Concurrency Primitives + Library

- If C++ must change to support concurrency, how much? (spectrum)
- Simple primitives with powerful abstractions is a compelling design principle.
- A small set of primitives does not imply a reduction in complexity (Turing tarpit).
  - E.g., OO programming can be simulated with a structure and routine pointer.
- Similarly for concurrency: providing only a mechanism to start a thread and a lock (e.g., pthreads) is too low level.
  - tedious and error-prone
  - does not take advantage of advanced programming features
- While some low-level mechanisms are important for extreme high-performance situations, most concurrent programs should be developed using medium or high level concurrency features to simplify the entire development cycle.

- C++ has a number of language mechanisms to bootstrap from low-level to medium-level concurrency (functor, closure, RAII) [6].
- These sequential mechanisms:
  - only marginally connect concurrency to the language,
  - may have non-orthogonal restrictions (declaration, inheritance),
  - require programmers to follow coding conventions to achieve correctness.
- To appreciate problems, examine:
  - thread/execution-context creation
  - mutual exclusion and synchronization (ME/S)
  - communication
- Many other problematic situations exist.

Thread/Execution-Context Creation

- Thread/execution-context creation needs "magic" as it is a primitive operation.
- Requires:
  - magic routine (pthread_create) or type (thread) to embed primitive operation
  - place for thread to start
  - place to create new execution context (stack)
- Can be created separately, but usually are combined in a single operation.
- Often a routine is used: named block and can be passed parameters.
- However, passing an arbitrary number and kind of initial arguments is problematic (single **void** * parameter)
- In OO language, extend the notion of a **class** using a closure object inheriting from magic thread type (Modula-3, Java, etc).

C++ CONNECTIONS

```
class thread {                              // magic type
  protected:
    virtual void main() = 0;                // thread starts in this routine
    . . .                                   // other thread specific members
  public:
    void start();                           // start the thread
    void join();                            // wait for thread termination
};
class mythread : public thread {            // inherit from magic type
    // closure data accessible by "main"
  protected:
    void main() {}                          // thread starts here
  public:
    mythread(. . .) { . . . }               // initialize closure data
};
```

- If thread “magic” is unknown to the compiler, sequential compilation problems still exist.
- If compiler knows about magic, either a distinguished type or a new kind of class can be used:

class mythread : **public** thread { or **task** mythread {

Without Magic	With Magic
mythread *t = **new** mythread(...);	mythread *t = **new** mythread(...);
t->start();	
...	...
t->join();	
delete t;	**delete** t;

- Problem: thread cannot start until after all constructor(s), otherwise thread's storage not initialized.
- Similarly, thread must terminate before destructor(s), otherwise deallocating a running thread's storage.
- Cannot be accomplished with C++ inheritance.
- Requires programmer to perform error-prone coding conventions.
- Use compiler magic after/before thread creation/destruction (if compiler knows).

Mutual Exclusion and Synchronization

- Providing ME/S is often done with mutex and condition locks (e.g., pthreads).
- Problematic: additional responsibilities for the programmer:
  - creating and initializing locks,
  - acquiring and releasing locks,
  - mutual exclusion when signalling a condition lock,
  - recheck an event after unblocking from a wait,
  - recursive acquiring of a lock,
  - not returning shared variables directly after releasing the monitor lock

```
lock.release();
return x + y;     // where x & y are shared
```

Without Magic	With Magic
class buffer {	**monitor** buffer {
mutex_t mutex;	
cond_t full, empty;	cond_t full, empty;
void insert(**int** elem) {	**void** insert(**int** elem) {
RAII lock(mutex);	
. . .	. . .
empty.wait(mutex);	empty.wait();

- Embedding locks in a class type simplifies declaration and initialization.
- Creating a special enter/exit class (RAII) gives automatic lock acquisition and release, and returned expressions are correctly returned [7].
- Alternatively, make ME/S implicit and contained with a new kind of class:
- The popularity of concurrent programming in Java comes largely from the ME/S simplification in its monitor type.

Communication

- Channels

```
channel x, y;
x.read(...);        // may block
y.read(...);        // may block
```

- Multiple queues require a selection mechanism:

```
select x.read(...) | y.read(...);
```

- Channels need to support different types of messages, resulting in complexity and possible type-unsafety (message passing).
- Alternatively, use member calls to **class**'s public interface for communication:

```
monitor {
  public:
    void m1(...);     // multiple parameter, type-safe calls, implicit ME/S
    void m2(...);
```

- Selection mechanism based on member routines:

```
select m1 | m2;
```

- Multiple forms of communication are confusing.

Summary

- Attempting to bootstrap from low-level concurrency mechanisms to even medium-level still results in:
  - unnecessary complexity for programmers,
  - lack of integration with other notions in the language,
  - compiler still lacks information about the intent of the program.
- Neither language's structure nor compiler's ability to deal with details are used.
- This approach may allow a larger set of concurrency models to be built.
- However, most developers would prefer one consistent powerful concurrency approach rather than a myriad of different but incompatible approaches.

High-Level : Advanced Concurrency Primitives

- Want a single consistent high-level powerful concurrency mechanism, but what should it look like?
- In theory, any high-level concurrency paradigm/model can be adapted into C++.
- C++ does not support all concurrency approaches equally well, e.g., tuple space, message passing, channels.
- C++ is fundamentally based on class model using routine call, and its other features leverage this model.
- Any concurrency approach matching the C++ model is better served because its concepts interact consistently with the language.
- Apply "Principle of Least Astonishment" whenever possible.
- Let C++ dictate which concurrency approaches fits best via its design principles.

- For OO language, thread/execution-context is best associated with class, and ME/S with member routines.

- Different properties produce different abstractions. Eureka!

object properties		member routine properties	
thread	stack	No ME/S	ME/S
No	No	**1** class	**2** monitor
No	Yes	**3** coroutine	**4** coroutine-monitor
Yes	No	**5** reject	**6** reject
Yes	Yes	**7** reject?	**8** task

- When thread or stack is missing it comes from calling object.

- Abstractions are not ad-hoc, rather derived from basic properties.

- Each of these abstractions has a particular set of problems it can solve, and therefore, each has a place in a programming language.

μC++ Design Principles

- integrate concurrency tightly into C++
  - leverage all class features
  - objects are light-weight: M:N thread model versus 1:1
- use mutex objects to contain mutual exclusion and synchronization
  - communication using routine call (versus messages/channels)
  - statically typed
- handle multi-processor environment
- integrate with underlying OS

μC++ : Concurrency in C++

- translator and run-time library
- provides high-level, integrated, object-oriented concurrency
- based on 3 elementary concurrency properties: thread, stack, and ME/S
- two new type constructors:
  - **_Coroutine** / **_Task**, extensions of **class**
  - provide stack and thread
- two new type qualifiers:
  - **_Mutex** / **_Nomutex**, qualify type constructors and member routines
  - provide mutual exclusion
- inherited members for context-switch/synchronization and one new statement:
  - suspend(), resume(), wait(), signal(), signalBlock(), **_Accept**

	No ME/S	ME/S

No Stack / No Thread	`class c {`
	
	`  public:`
	`    m() { }`
	`};`
		`_Mutex class M {  // _Monitor M`
		`    uCondition variables;`
		`  public:`
		`    m() { wait/signal/_Accept }`
		`};`

Stack / No Thread	`_Coroutine C {`
	
	`    main() { suspend }`
	`  public:`
	`    m() { resume }`
	`};`
		`_Mutex _Coroutine CM { // _Cormonitor CM`
		`    uCondition variables;`
		`    main() { suspend/wait/signal/_Accept }`
		`  public:`
		`    m() { resume/wait/signal/_Accept }`
		`};`

Stack / Thread		`_Task T {`
		`    uCondition variables;`
		`    main() { suspend/wait/signal/_Accept }`
		`  public:`
		`    m() { resume/wait/signal/_Accept }`
		`};`

- each coroutine implicitly inherits from uBaseCoroutine
- each task implicitly inherits from uBaseTask

Coroutine

- Execution context associated with *distinguished* member.

```
_Coroutine C {
    void main() {            // distinguished member
        ... suspend() ...    // restart last resume
        ... suspend() ...    // restart last resume
    }
  public:
    void m1( ... ) { ... resume(); ... }   // restart last suspend
    void m2( ... ) { ... resume(); ... }   // restart last suspend
};
```

- First resume starts main; subsequent resumes restart last suspend.
- suspend starts last resume
- Both statements cause a thread to context switch to a different execution context.
- Object becomes a coroutine on first resume; coroutine becomes an object when main ends.
- Multiple public member routines allow complex interface.
- Directly supports finite-state problems.

- Parse phone number: (555)123-4567

```
_Coroutine Phone {
    char ch;
    int stat;
  public:
    int next( char c ) {
        ch = c;
        resume();
        return stat;
    }
  private:
    void main() {
        int i;
        stat = 0;
        if ( ch == '(' ) {
            for ( i = 0; i < 3; i += 1 ) {
                suspend();
                if ( ! isdigit(ch) ) { stat = 2; return; }
            } // for
            suspend();
            if ( ch != ')' ) { stat = 2; return; }
            suspend();
        } // if

        for ( i = 0; i < 3; i += 1 ) {
            if ( ! isdigit(ch) ) { stat = 2; return; }
            suspend();
        } // for
        if ( ch != '-' ) { stat = 2; return; }

        for ( i = 0; i < 4; i += 1 ) {
            suspend();
            if ( ! isdigit(ch) ) { stat = 2; return; }
        } // for
        stat = 1;
    }
};
```

- Killer app: device drivers; cause 70%-85% of failures in Windows/Linux [8]

. . . STX . . . message . . . ESC ETX . . . message . . . ETX 2-byte crc . . .

Monitor

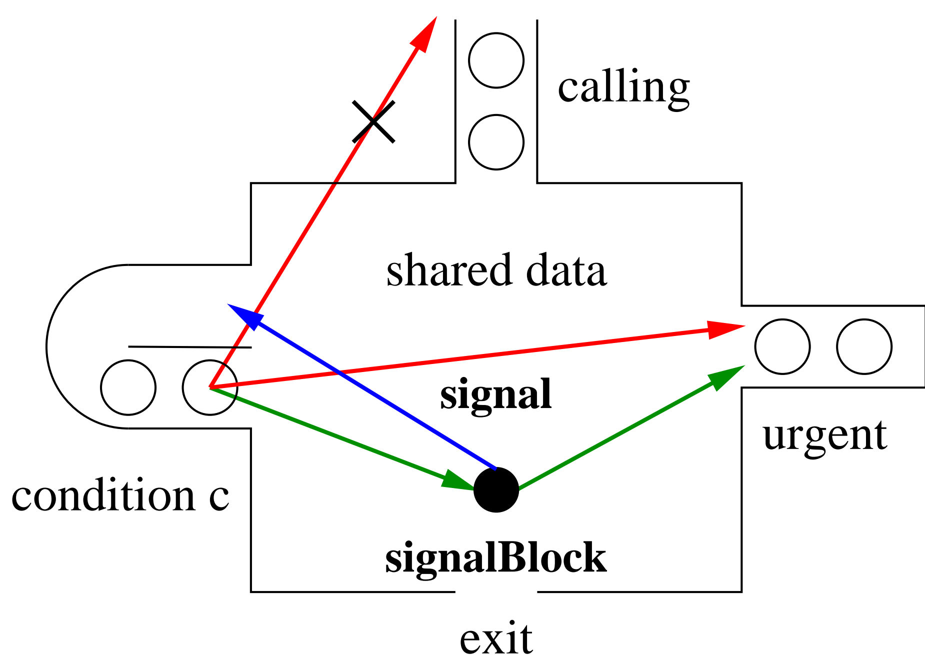
- Object with implicit mutual exclusion on calls to specified member routines:

```
_Mutex class M {   // default for public member routines
    uCondition c1, c2[10], *c3 = new uCondition;

    // private/protected default is no mutual exclusion
    void m1( . . . );                  // no mutual exclusion
    _Mutex void m2( . . . );           // mutual exclusion
  public:
    void m2( . . . );                  // mutual exclusion
    _Nomutex void m3( . . . );         // no mutual exclusion
    . . . // destructor is ALWAYS mutex
};
```

- Recursive entry is allowed (owner mutex lock).
- Because destructor is mutex, termination of a block containing a monitor or deleting a dynamically allocated monitor blocks if thread executing in monitor.
- Synchronization using condition variables (wait(), signal(), signalBlock()) or **_Accept** on mutex members.
- Service order of communication can be precisely controlled.

Condition Variable

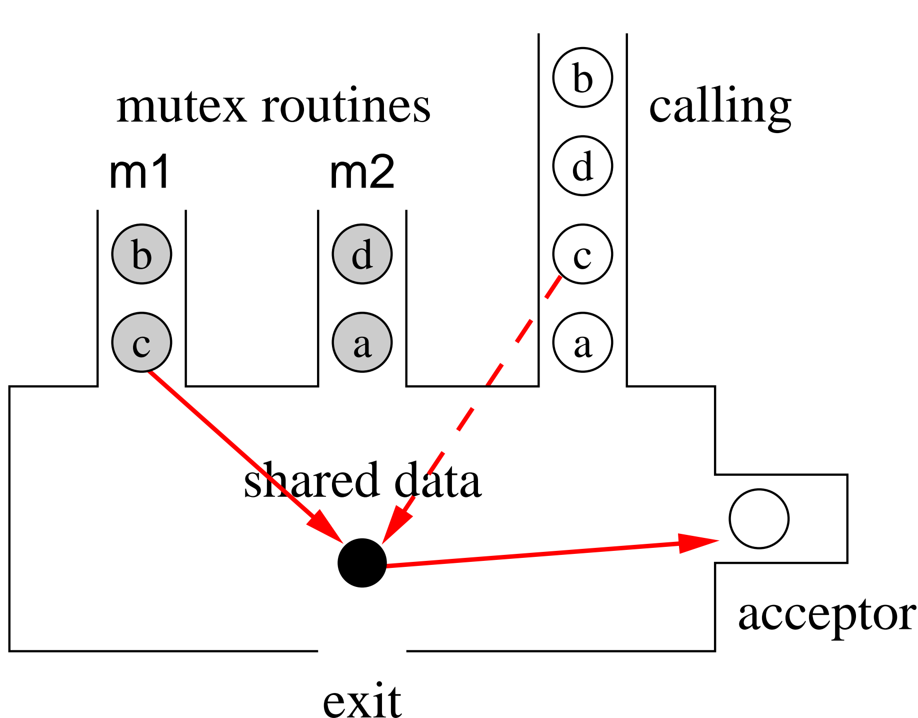


- c.wait() blocks the current thread, and restarts a signalled task or implicitly releases the monitor lock.
- c.signal() unblocks the thread on the front of the condition queue *after* the signaller thread blocks or exits.
- c.signalBlock() unblocks the thread on the front of the condition queue and blocks the signaller thread.

- NO BARGING TASKS! No polling for event occurrence.
- uCondition variable is branded by first mutex object that waits on it.

Accept Statement

- **_Accept** chooses the mutex-member call to execute next (Ada select).



- **_Accept**(m1) unblocks the thread on the front of the mutex queue *after* the accepter blocks.
- if no waiting task, wait until a call to the specified member occurs.
- the accepted call is executed like a conventional member call
- when the member ends, the caller and the acceptor continue at the call site and the **_Accept** statement.

- **_Accept** can appear at any level of routine nesting

- Extended **_Accept** statement chooses from a *group* of members.

```
_Accept( m1, m2, . . . );
```

- Service first non-empty mutex queue, or if empty, first call to any of the specified mutex members.

- Order of mutex names is important.

- Each selected member can be separated and supplied with a guard.

```
_When( conditional-expression ) // optional guard
    _Accept( m1 )
        statement                // optional statement
else _When( conditional-expression )
    _Accept( m2 )
        statement
. . .
else                             // default clause (does not block)
    statement
```

```
_Mutex class DatingService {
    uCondition Girls[20], Boys[20];        // array of condition locks
    int GirlPhoneNo, BoyPhoneNo;
  public:
    int Girl( int PhoneNo, int code ) {    // code => compatibility
        if ( Boys[code].empty() ) {        // boy waiting?
            Girls[code].wait();            // wait for boy
            GirlPhoneNo = PhoneNo;         // exchange
        } else {
            GirlPhoneNo = PhoneNo;         // exchange
            Boys[code].signalBlock();      // unblock boy
        }
        return BoyPhoneNo;
    }
    int Boy( int PhoneNo, int code ) {
        if ( Girls[code].empty() ) {
            Boys[code].wait();
            BoyPhoneNo = PhoneNo;
        } else {
            BoyPhoneNo = PhoneNo;
            Girls[code].signalBlock();
        }
        return GirlPhoneNo;
    }
};
```

```
_Mutex class ReadersWriter {    // multiple simultaneous readers, only one writer
        int rcnt, wcnt;
  public:
        void ReadersWriter() {
            rcnt = wcnt = 0;
        }
        void EndRead() {
            rcnt -= 1;
        }
        void EndWrite() {
            wcnt = 0;
        }
        void StartRead() {
            if ( wcnt == 1 ) _Accept( EndWrite );   // prevent all other calls
            rcnt += 1;
        }
        void StartWrite() {
            if ( wcnt == 1 ) _Accept( EndWrite );   // prevent all other calls
            else while ( rcnt > 0 )
                    _Accept( EndRead );             // prevent all other calls
            wcnt = 1;
        }
};
```

Coroutine-Monitor

- Coroutine with implicit mutual exclusion on calls to specified member routines:

```
_Mutex _Coroutine C { // default for public member routines
    void main() {
        . . . suspend() . . .
        . . . suspend() . . .
    }
  public:
    void m1( . . . ) { . . . resume(); . . . }   // mutual exclusion
    void m2( . . . ) { . . . resume(); . . . }   // mutual exclusion
    . . . // destructor is ALWAYS mutex
};
```

- Can use resume(), suspend(), condition variables (wait(), signal(), signalBlock()) or **_Accept** on mutex members.

- Coroutine can now be used by multiple threads, e.g., coroutine print-formatter accessed by multiple threads.

Task

- Coroutine-monitor with a thread.

```
_Task T {
    void main();      // distinguished member, thread starts here
    _Mutex void m1( . . . );        // mutual exclusion
  public:
    void m2( . . . );               // mutual exclusion
    _Nomutex void m3( . . . );      // no mutual exclusion
    . . . // destructor is ALWAYS mutex
};
```

- Can use resume(), suspend(), condition variables (wait(), signal(), signalBlock()) or **_Accept** on mutex members.

- Combination of condition variables and accepting mutex members is often used with tasks.

- Possibly accept destructor to know when to terminate task.

```
const int rows = 10, cols = 10;  // globals 8-(
int M[rows][cols], ST[rows];

_Task Adder {                     // add specific row
    static int row;               // sequential access
    int myrow, c;
    void main() {
        ST[myrow] = 0;            // location of subtotal
        for ( c = 0; c < cols; c += 1 )
            ST[myrow] += M[myrow][c];
    }
  public:
    Adder() { myrow = row++; }    // choose row
};
int Adder::row = 0;
void uMain::main() {
    // read matrix
    {
        Adder adders[rows];       // create N threads
    } // wait for threads to terminate
    int total = 0;                // sum subtotals
    for ( int r = 0; r < rows; r += 1 )
        total += ST[r];
    cout << total << endl;
}
```

```
_Task Server {             // combine condition variables and _Accept
    uCondition delay;
    void main() {
        for ( ;; ) {       // for each request from clients
            _Accept( ~Server ) {     // terminate when destructor called
                break;
              // each kind of client request
            } else _Accept( workReq1 ) {
                . . . delay.signalBlock(); . . .  // unblock client, runs first
            } else _Accept( workReq2 ) {
                . . .
            }
        }
        // shut down
    }
  public:
    void workReq1( Req1_t req ) {
        . . . delay.wait(); . . .        // block if client cannot be serviced
        // otherwise request is handled immediately
    }
    void workReq2( Req2_t req ) { . . . }
    . . .
};
```

Exception Handling

- Exception handling is based on traversing a call stack (execution context).
- With coroutines and tasks, there are now multiple call stacks.
- Therefore, exception handling is extended into this advanced environment.
- Provide a special exception type and exception hierarchy.
- Non-local exception handling : exceptions among coroutines and tasks

```
_Throw    [ throwable-event [ _At coroutine/task-id ] ] ; // termination
_Resume [ resumable-event [ _At coroutine/task-id ] ] ; // resumption
```

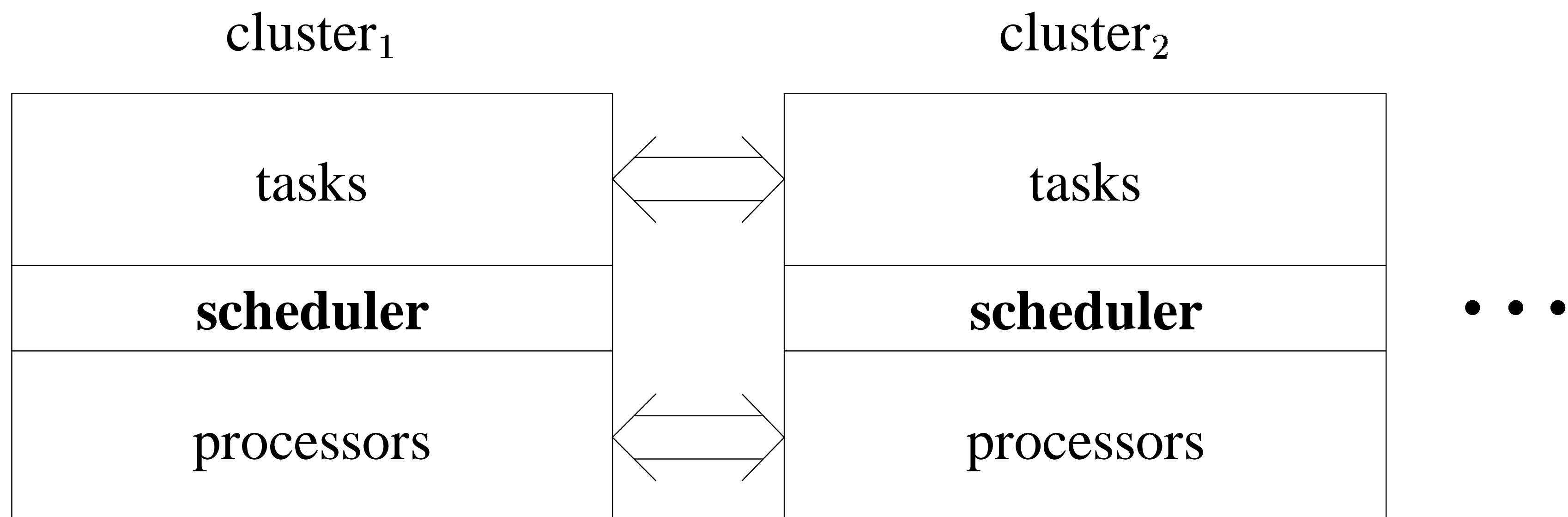
- When an exception is not handled by a coroutine, the exception is implicitly forwarded to the coroutine's resumer.
- Implies exception ultimately propagated to the task executing the coroutine(s).
- An asynchronous exception between tasks is delivered as soon as possible.
- Control delivery of non-local exceptions:

```
_Enable <E1><E2>... {                    _Disable <E1><E2>... {
    ... // exceptions delivered              ... // exceptions not delivered
}                                        }
```

- Specifying no exceptions enables/disables all exceptions.

Runtime Structure

- cluster: logical mechanism to organize M tasks and N processors
- each cluster has a scheduler to select tasks to run on processors



- each task/processor is created on a specific cluster
- task/processor can migrate among clusters

Input/Output

- C++ streams and UNIX files/sockets do not work with general threading.
- Might be "thread-safe" for pthreads (pthread pandemic).
- Need general thread-safe, object-oriented, nonblocking I/O across application.
- Stream
  - use RAII for locking
  - single line

```
isacquire( cin ) >> ...;
osacquire( cout ) << ... << endl;
```

  - multi-line

```
{
    osacquire acq( cout );
    cout << ...;
    cout << ...;
}
```

- File
  - use RAII for implicit open/close

```
void uMain::main() {
    uFile input( "abc" );          // connect to file
    uFileAccess in( input );       // open file
    char buf[100];

    for ( ;; ) {
        in.read( buf, 100 );       // locking
        . . .
    }
}
```

- Socket
  - Server and Acceptor (depending on kind of socket)
  - Client
  - connectionless and connected

Server

```
void uMain::main() {
    short unsigned int port;
    uSocketServer server( &port );
    try {
        uDuration t( 10, 0 );
        uSocketAccept apt( server, &t );
        char buf[100];

        for ( ;; ) {
            apt.read( buf, 100 );
            . . .
        } // for
    } catch( uSocketAccept::OpenTimeout ) {
        . . .
    } // try
}
```

Client

```
void uMain::main() {
    uSocketClient clt( port );
    char buf[100];

    for ( ;; ) {
        clt.write( buf, 100 );
        . . .
    } // for
}
```

Real-Time

- real-time programming is the most complex form of concurrency
- extensible schedulers (fixed, dynamic)
- transitive priority-inheritance protocol
- timeout facility

```
_Accept( m1, m2 ) { ... }
else _Accept ( m3 ) { ... }
else _Timeout( 1 ) { ... } // after 1 second
```

- real-time tasks:

```
_RealTimeTask A { ... };
_PeriodicTask  P { ... };
_SporadicTask  S { ... };
```

Miscellaneous

- optimized uniprocessor and multi-processor versions
- debug mode for testing (asserts and runtime checks)
- reasonable error messages
- profiler and debugger?
- compilers
  - gcc-3.2 or greater
  - Intel icc 8.1/9.0
- architectures
  - Linux IA-64 (HP/SGI)
  - Linux Opteron
  - Linux IA-32/AMD
  - Solaris 8/9/10 SPARC
  - IRIX 6.x MIPS
- http://plg.uwaterloo.ca/~usystem/uC++.html

Conclusions

- Concurrency is hard, but not impossible.
- High-level features can make concurrency a tractable programming paradigm.
- Three fundamental execution properties produce necessary constructs.
- New constructs based on **class** to leverage all existing capabilities.
- Mutual exclusion is implicit and synchronization contained in mutex objects.
- All communication is consistent (routine call) and statically type-safe.
- Choose the right defaults!
  - 99% of all tasks are started immediately after declaration
  - 99% of all tasks are destroyed immediately after joining
  - 99% of programmers don't understand barging tasks for mutex types
  - ...

References

[1] Sutter, H. “The Free Lunch Is Over: A Fundamental Turn Toward Concurrency in Software”. *Dr. Dobb’s: Software Tools for the Professional Programmer*, 30(3), Mar. 2005.

[2] Buhr, P. A., Ditchfield, G., and Zarnke, C. R. “Adding Concurrency to a Statically Type-Safe Object-Oriented Programming Language”. *SIGPLAN Notices*, 24(4):18–21, Apr. 1989.

[3] Buhr, P. A. and Ditchfield, G. “Adding Concurrency to a Programming Language”. In *USENIX C++ Technical Conference Proceedings*, pages 207–224, Portland, Oregon, U.S.A., Aug. 1992. USENIX Association.

[4] Buhr, P. A. “Are Safe Concurrency Libraries Possible?”. *Commun. ACM*, 38(2):117–120, Feb. 1995.

[5] Boehm, H.-J. “Threads Cannot be Implemented as a Library”. *SIGPLAN Notices*, 40(6):261–268, June 2005.

[6] Schmidt, D. C. “An OO Encapsulation of Lightweight OS Concurrency Mechanisms in the ACE Toolkit”. Technical Report 31, Washington University in St. Louis, 1995. http://www.cs.wustl.edu/~schmidt/PDF/IPC_SAP-92.pdf.

[7] Bershad, B. N., Lazowska, E. D., and Levy, H. M. “PRESTO: A System for Object-oriented Parallel Programming”. *Software—Practice and Experience*, 18(8):713–732, Aug. 1988.

[8] Swift, M. M., Bershad, B. N., and Levy, H. M. “Improving the Reliability of Commodity Operating Systems”. *ACM Trans. Comput. Syst.*, 23(1):77–110, Feb. 2005.