Outline

An introduction to concurrent programming, with an emphasis on language constructs. Major topics include: exceptions, coroutines, atomic operations, critical sections, mutual exclusion, semaphores, high-level concurrency, deadlock, interprocess communication, process structuring on shared memory architectures. Students learn how to structure, implement and debug complex control-flow.

*Permission is granted to make copies for personal or educational use.*
## Contents

1 Advanced Control Flow (Review) .................................................. 1
   1.1 Static multi-level exit ....................................................... 2
   1.2 Dynamic Memory Allocation ............................................... 3

2 Dynamic Multi-Level Exit ............................................................ 5
   2.1 Implementation .............................................................. 8
   2.2 Traditional Approaches ..................................................... 9
   2.3 Exception Handling ........................................................ 12
   2.4 Execution Environment .................................................... 13
   2.5 Terminology ............................................................... 13
   2.6 Static/Dynamic Call/Return .............................................. 14
   2.7 Static Propagation ......................................................... 15
   2.8 Dynamic Propagation ........................................................ 16
      2.8.1 Termination ............................................................ 16
      2.8.2 Resumption ............................................................ 18
   2.9 Exceptional Example ....................................................... 19

3 Coroutine ................................................................................. 21
   3.1 Semi-Coroutine ............................................................... 22
      3.1.1 Fibonacci Sequence ................................................... 22
         3.1.1.1 Direct ................................................................ 22
         3.1.1.2 Routine ........................................................... 22
         3.1.1.3 Class ............................................................... 23
         3.1.1.4 Coroutine ......................................................... 23
      3.1.2 Format Output ............................................................. 25
         3.1.2.1 Direct ................................................................ 25
         3.1.2.2 Routine ........................................................... 26
         3.1.2.3 Class ............................................................... 26
         3.1.2.4 Coroutine ......................................................... 27
      3.1.3 Correct Coroutine Usage .............................................. 28
      3.1.4 Coroutine Construction ............................................... 29
   3.2 µC++ EHM ..................................................................... 29
   3.3 Exception Type ................................................................. 29
   3.4 Inherited Members ............................................................ 30
   3.5 Raising ............................................................................ 30
### 3.6 Handler
- 3.6.1 Termination ........................................ 31
- 3.6.2 Resumption .......................................... 31
- 3.6.3 Termination/Resumption ............................ 32

### 3.7 Nonlocal Exceptions .................................... 33

### 3.8 Memory Management ................................... 36

### 3.9 Semi-Coroutine Examples ............................... 36
- 3.9.1 Same Fringe ......................................... 36
- 3.9.2 Device Driver ........................................ 37
  - 3.9.2.1 Direct ........................................... 38
  - 3.9.2.2 Coroutine ....................................... 38
- 3.9.3 Producer-Consumer .................................. 40

### 3.10 Full Coroutines ........................................ 41
- 3.10.1 Ping/Pong .......................................... 43
- 3.10.2 Producer-Consumer ................................ 45

### 3.11 Coroutine Languages ................................. 47
- 3.11.1 Python 3.5 ......................................... 47
- 3.11.2 C++ Boost Library (V1.61) ......................... 48

### 4 More Exceptions .......................................... 51
- 4.1 Derived Exception-Type ............................... 51
- 4.2 Catch-Any ............................................. 51
- 4.3 Exception Parameters ................................. 52
- 4.4 Exception List ......................................... 53
- 4.5 Destructor ............................................. 54
- 4.6 Multiple Exceptions ................................. 54

### 5 Concurrency ............................................... 57
- 5.1 Why Write Concurrent Programs .................... 57
- 5.2 Why Concurrency is Difficult ....................... 57
- 5.3 Concurrent Hardware .................................. 58
- 5.4 Execution States ...................................... 59
- 5.5 Threading Model ...................................... 60
- 5.6 Concurrent Systems ................................... 61
- 5.7 Speedup ............................................... 62
- 5.8 Thread Creation ....................................... 64
  - 5.8.1 COBEGIN/COEND ................................. 64
  - 5.8.2 START/WAIT ..................................... 65
  - 5.8.3 Thread Object .................................... 65
  - 5.8.4 Actor ............................................ 67
- 5.9 Termination Synchronization ......................... 69
- 5.10 Divide-and-Conquer ................................. 69
- 5.11 Exceptions ........................................... 71
- 5.12 Synchronization and Communication During Execution ................................. 72
- 5.13 Communication .................................... 72
5.14 Critical Section .................................................. 73
5.15 Static Variables .................................................. 73
5.16 Mutual Exclusion Game ........................................... 74
5.17 Self-Testing Critical Section ................................. 75
5.18 Software Solutions ................................................ 75
  5.18.1 Lock ............................................................ 75
  5.18.2 Alternation .................................................... 76
  5.18.3 Declare Intent ............................................... 76
  5.18.4 Retract Intent ................................................ 77
  5.18.5 Prioritized Retract Intent ................................. 77
  5.18.6 Dekker (modified retract intent) ......................... 78
  5.18.7 Peterson (modified declare intent) ...................... 80
  5.18.8 N-Thread Prioritized Entry ............................... 81
  5.18.9 N-Thread Bakery (Tickets) ............................... 82
  5.18.10 Tournament ................................................ 82
  5.18.11 Arbiter ...................................................... 84
5.19 Hardware Solutions ............................................... 85
  5.19.1 Test/Set Instruction ................................. 86
  5.19.2 Swap Instruction ......................................... 86
  5.19.3 Fetch and Increment Instruction ......................... 86
6 Locks .................................................................. 89
  6.1 Lock Taxonomy ............................................... 89
  6.2 Spin Lock .......................................................... 89
    6.2.1 Implementation ........................................... 90
  6.3 Blocking Locks ...................................................... 91
    6.3.1 Mutex Lock ................................................ 91
      6.3.1.1 Implementation .................................... 92
      6.3.1.2 uOwnerLock ......................................... 95
      6.3.1.3 Mutex-Lock Release-Pattern ....................... 95
      6.3.1.4 Stream Locks ....................................... 96
    6.3.2 Synchronization Lock ...................................... 97
      6.3.2.1 Implementation .................................... 97
      6.3.2.2 uCondLock ........................................ 100
      6.3.2.3 Programming Pattern ............................ 100
    6.3.3 Barrier ...................................................... 101
      6.3.3.1 uBarrier ........................................... 103
    6.3.4 Binary Semaphore ........................................ 104
      6.3.4.1 Implementation .................................... 106
    6.3.5 Counting Semaphore ..................................... 106
      6.3.5.1 Implementation .................................... 107
  6.4 Lock Programming ............................................... 108
    6.4.1 Precedence Graph ...................................... 108
    6.4.2 Buffering .................................................... 109
      6.4.2.1 Unbounded Buffer ................................ 110
1 Advanced Control Flow (Review)

• **Within** a routine, basic and advanced control structures allow virtually any control flow.

• **Multi-exit loop** (or mid-test loop) has one or more exit locations occurring within the body of the loop, not just top (while) or bottom (do-while):

```cpp
for ( ;; ) {  // infinite loop, while ( true )
    ...
    if ( ... ) break;  // middle exit
    ...
}
```

condition reversed from while, outdent exit for readability

• Eliminates priming (duplicated) code necessary with while:

```cpp
cin >> d;  // priming
while ( ! cin.fail() ) {
    ...
    cin >> d;
    ...
}
```

• Eliminate else on loop exits:

<table>
<thead>
<tr>
<th>BAD</th>
<th>GOOD</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>for ( ;; ) {</code></td>
<td><code>for ( ;; ) {</code></td>
</tr>
<tr>
<td>S1</td>
<td>S1</td>
</tr>
<tr>
<td>if ( C1 ) {</td>
<td>if ( ! C1 ) break;</td>
</tr>
<tr>
<td>S2</td>
<td>S2</td>
</tr>
<tr>
<td>} else {</td>
<td>}</td>
</tr>
<tr>
<td></td>
<td>break;</td>
</tr>
<tr>
<td>S3</td>
<td>S3</td>
</tr>
</tbody>
</table>

S2 is logically part of loop body *not* part of an if.

• Allow multiple exit conditions:

```cpp
for ( ;; ) {
    S1
    if ( i >= 10 ) { E1; break; }
    S2
    if ( j >= 10 ) { E2; break; }
    S3
}
```

```cpp
bool flag1 = false, flag2 = false;
while ( ! flag1 & ! flag2 ) {
    S1
    if ( C1 ) flag1 = true;
    } else {
        S2
        if ( C2 ) flag2 = true;
    } else {
        S3
    }
}
```

```cpp
if ( flag1 ) E1;
else E2;
```
• Eliminate flag variables used solely to affect control flow, i.e., variable does not contain data associated with computation.

• Flag variables are the variable equivalent to a goto because they can be set/reset/tested at arbitrary locations in a program.

1.1 Static multi-level exit

• Static multi-level exit exits multiple control structures where exit point is known at compile time.

• Labelled exit (break/continue) provides this capability:

```
µC++ / Java                          C / C++
L1: { // good eye–candy
     ... declarations ...
L2: switch ( ... ) {
  L3: for ( ... ) {
    ... break L1; // exit block
    ... break L2; // exit switch
    ... break L3; // exit loop
  }
  ...
}

...}
```

• Why is it good practice to label all exits?

• Eliminate all flag variables with multi-level exit!

```
B1: for ( i = 0; i < 10; i += 1 ) {
  B2: for ( j = 0; j < 10; j += 1 ) {
    ...
    if ( ... ) break B2; // outdent
    ...
    // rest of loop
    if ( ... ) break B1; // outdent
    ...
    // rest of loop
  } // for
  ...
} // for

bool flag1 = false;
for ( i = 0; i < 10 && ! flag1; i += 1 ) {
  boolean flag2 = false;
  for ( j = 0; j < 10 && ! flag1 && ! flag2; j += 1 ) {
    ...
    if ( ... ) flag2 = true;
    else {
      ...
      // rest of loop
      if ( ... ) flag1 = true;
      else {
        ...
        // rest of loop
      } // if
      } // if
    } // if
  } // for
  ...
} // for
```
1.2 Dynamic Memory Allocation

- Occasionally a flag variable is necessary!

```cpp
int val; bool valDefault = false;
switch ( argv ) {
  
  case 3:
    if ( strcmp( argc[4], "d" ) != 0 ) valDefault = true; // default
    else val = stoi( argc[4] ); // value
  
  } // switch
for ( ;; ) {
  
    if ( valDefault ) // do something
      else // do another
  
  } // for
```

<table>
<thead>
<tr>
<th>duplication</th>
<th>no duplication</th>
</tr>
</thead>
<tbody>
<tr>
<td>if ( C1 ) { S1;</td>
<td>C: { if ( C1 ) { S1;</td>
</tr>
<tr>
<td>if ( C2 ) { S2;</td>
<td>if ( C2 ) { S2;</td>
</tr>
<tr>
<td>if ( C3 ) { S3;</td>
<td>if ( C3 ) { S3;</td>
</tr>
<tr>
<td>} else S4;</td>
<td>break C; }</td>
</tr>
<tr>
<td>} else S4;</td>
<td>} }</td>
</tr>
<tr>
<td>S4;</td>
<td>}</td>
</tr>
</tbody>
</table>

- Normal and labelled `break` are a `goto` with limitations:

  1. Cannot loop (only forward branch) ⇒ only loop constructs branch back.
  2. Cannot branch into a control structure.

- Only use `goto` to perform static multi-level exit, e.g., simulate labelled `break` and `continue`.

1.2 Dynamic Memory Allocation

- Stack allocation eliminates explicit storage-management and often more efficient than heap allocation — “Use the STACK, Luke Skywalker.”

```cpp
{ // GOOD, use stack
  cin >> size;
  int arr[size];
  ... // use arr[i]
}

{ // BAD, unnecessary dynamic allocation
  cin >> size;
  int * arr = new int[size];
  ... // use arr[i]
  delete [] arr; // why "[]"?
}
```
CHAPTER 1. ADVANCED CONTROL FLOW (REVIEW)

- These are the situations where dynamic allocation (heap) is necessary:

  1. When a variable’s storage must outlive the block in which it is allocated.


```
Type * rtn(...) {
    Type * tp = new Type; // MUST USE HEAP
    // initialize/compute using tp
    return tp; // storage outlives block
    // tp deleted later
}
```

  2. When the amount of data read is unknown.

```
vector<int> input;
int temp;
for (;;) {
    cin >> temp;
    if ( cin.fail() ) break;
    input.push_back( temp ); // implicit dynamic allocation
}
```

Does switching to emplace_back help?

  3. When an array of objects must be initialized via the object’s constructor.

```
struct Obj {
    int id; ... 
    Obj( int id ) : id( id ) { ... }
}
```

```
cin >> size;
Obj * objs[size];
for ( int id = 0; id < size; id += 1 ) {
    objs[id] = new Obj( id ); // each element has different value
    // ... 
    delete objs[id];
```

or

```
#include <memory>
{
    unique_ptr<Obj> objs[size];
    for ( int id = 0; id < size; id += 1 ) {
        // objs[id].reset( new Obj( id ) ); // C++11
        objs[id] = make_unique<Obj>( id ); // C++14
    }
    // ... 
} // automatically delete objs
```

  4. When large local variables are allocated on a small stack.

```
Coroutine C {
    void main() { // 64K stack
        int arr[100000]; // overflow
        int * arr = new int[100000];
        // ... 
    }
}
```

C{
2 Dynamic Multi-Level Exit

- Routine activation (call/invocation) introduces complex control flow.

- Among routines, control flow is controlled by call/return mechanism.

- Modularization: any contiguous code block can be factored into a (helper) routine and called from anywhere in the program (modulo scoping rules).

- Modularization fails when factoring exits, e.g., multi-level exits:

```
B1: for ( i = 0; i < 10; i += 1 ) {
    ...
    B2: for ( j = 0; j < 10; j += 1 ) {
        ...
        if ( ... ) break B1;
        ...
    }
    ...
}
```

Does this compile?

- Pattern: many routines have multiple outcomes.
Two alternate return parameters, denoted by * and implicitly named 1 and 2

```fortran
subroutine AltRet( c, *, * )
  integer c
  if ( c == 0 ) return ! normal return
  if ( c == 1 ) return 1 ! alternate return
  if ( c == 2 ) return 2 ! alternate return
end
```

Statements labelled 10 and 20 are alternate return points

```fortran
call AltRet( 0, *10, *20 )
print *, "normal return 1"
call AltRet( 1, *10, *20 )
print *, "normal return 2"
return
10 print *, "alternate return 1"
call AltRet( 2, *10, *20 )
print *, "normal return 3"
return
20 print *, "alternate return 2"
stop
end
```

$ gfortran AltRtn.for
$ a.out
  normal return 1
  alternate return 1
  alternate return 2

- Generalization of multi-exit loop and multi-level exit.
  - Control structures ends normally or with an exceptional transfer.

- Pattern acknowledges:
  - Pattern does not handle multiple levels of nested modularization.

- If AltRet is further modularized, new routine has an alternate return to AltRet, which retains its alternate return to its caller.
Two alternate return parameters, denoted by * and implicitly named 1 and 2

subroutine AltRet2( c, *, * )
    integer c
    if ( c == 0 ) return ! normal return
    if ( c == 1 ) return 1 ! alternate return
    return 2
end

Two alternate return parameters, denoted by * and implicitly named 1 and 2

subroutine AltRet( c, *, * )
    integer c
    call AltRet2( c, *30, *40 )
    return
    if ( c == 2 ) return 2 ! alternate return
end

• Rather than two-step operation, simpler for new modularized routine to bypass intermediate step and transfer directly to caller of AltRet.

• Why not call AltRet2( c, *10, *20 )?

• Dynamic multi-level exit extend call/return semantics to transfer in the reverse direction to normal routine calls, called nonlocal transfer.

label L;
void f( int i ) {
    // nonlocal return
    if ( i == ... ) goto L;
}
void g( int i ) {
    if ( i > 1 ) { g( i - 1 ); return; }
    f( i );
}
void h( int i ) {
    if ( i > 1 ) { h( i - 1 ); return; }
    L = L1; // set dynamic transfer–point
    f( 1 ); goto S1;
    L1: // handle L1 nonlocal return
    S1: // continue normal execution
    L = L2; // set dynamic transfer–point
    g( 1 ); goto S2;
    L2: // handle L2 nonlocal return
    S2: // continue normal execution
}

• Nonlocal transfer mechanism is a label variable containing the tuple:
  1.
  2.

• Nonlocal transfer, goto L, in f is a two-step operation:
1. Therefore, a label value is not statically/lexically determined.
   - recursion in \( g \) ⇒ unknown distance between \( f \) and \( h \) on stack.
   - what if \( L \) is set during the recursion of \( h \)?

2. This complexity is why label constants have local scope.

- Transfer between \texttt{goto} and label value causes termination of stack block.

- First, nonlocal transfer from \( f \) transfers to the label \( L1 \) in \( h \)’s routine activation, terminating \( f \)’s activation.

- Second, nonlocal transfer from \( f \) transfers to the static label \( L2 \) in the stack frame for \( h \), terminating the stack frame for \( f \) and \( g \).

- Termination is implicit for direct transferring to \( h \) or requires stack unwinding if activations contain objects with destructors or finalizers.

- Nonlocal transfer is possible in C using:
  - \texttt{jmp_buf} to declare a label variable,
  - \texttt{setjmp} to initialize a label variable,
  - \texttt{longjmp} to \texttt{goto} a label variable.

- Nonlocal transfer allows multiple forms of returns to any level.
  - Normal return transfers to statement after the call, often implying completion of routine’s algorithm.
  - Exceptional return transfers to statement \texttt{not} after the call, indicating an ancillary completion (but not necessarily an error).

2.1 Implementation

- Nonlocal transfer is \textit{limited} in most programming languages using exception handling.
2.2. TRADITIONAL APPROACHES

- To implement throw/catch, the throw must know the last guarded block with a handler for the raised exception type.

- One approach is to:
  -
  -
  -

- For C++, a direct transfer is often impossible because:

- However, setting/resetting label variable on try block entry/exit has a cost (small):
  -
  -
  - Instead, catch/destructor data is stored once externally for each block and handler found by linear search during a stack walk (no direct transfer).
  - Advantage, millions of try entry/exit, but only tens of exceptions raised.

- Hence, termination is often implemented using zero cost on guarded-block entry but an expensive approach on raise.

**2.2 Traditional Approaches**

- **return code**: returns value indicating normal or exceptional execution.

- **status flag**: set shared (global) variable indicating normal or exceptional execution; the value remains as long as it is not overwritten.
• **fix-up routine**: a global and/or local routine called for an exceptional event to fix-up and return a corrective result so a computation can continue.

```c
int fixup( int i, int j ) { ... } // local routine
rtn( a, b, fixup ); // fixup called for exceptional event
```

• Techniques are often combined, e.g.:

```c
if ( printf(...) < 0 ) {
    perror( "printf:" ); // errno describes specific error
    abort(); // terminate program
}
```

• Intermediate approach of **return union**, which mimics normal return type.

• Return union combines result/return-code, and checks code on result access.

• **ALL** routines must return an appropriate union.

```c
optional< int * > Malloc( size_t size ) {
    if ( random() % 2 ) return (int *)malloc( sizeof( int ) );
    return nullopt;
}
optional< int > rtn( ) {
    optional< int * > p = Malloc( sizeof( int ) );
    if ( ! p ) return nullopt; // malloc successful (true/false) ?
    *p = 7; // compute
    if ( random() % 2 ) return *p;
    return nullopt;
}
int main() {
    srand( getpid() );
    optional<int> ret = rtn();
    if ( ret ) cout << *ret << endl; // rtn successful?
    else cout << "no storage or bad computation" << endl;
}
```

```
$ repeat 5 a.out
no storage or bad computation
7
no storage or bad computation
7
7
7
```
2.2. TRADITIONAL APPROACHES

```cpp
enum Alloc { NoStorage };
variant< int *, Alloc > Malloc( size_t size ) {
    if ( random() % 2 ) return { (int *)malloc( sizeof( int ) ) };
    return { NoStorage };
}
enum Comp { BadComp };
variant< int, Alloc, Comp > rtn( ) {
    variant< int *, Alloc > p = Malloc( sizeof( int ) );
    if ( ! holds_alternative<int *>(p) ) return { NoStorage }; // malloc successful ?
    *get<int *>(p) = 7;
    if ( random() % 2 ) return { *get<int *>(p) };
    return { BadComp };
}
int main() {
    srandom( getpid() );
    variant< int, Alloc, Comp > ret = rtn();
    if ( holds_alternative<int>(ret) ) cout << get<int>(ret) << endl;
    else if ( holds_alternative<Comp>(ret) ) cout << "bad computation" << endl;
    else cout << "no storage" << endl;
}
$ repeat 5 a.out
no storage
bad computation
no storage
bad computation
7
```

• Like Fortran, only returns one level.

• Drawbacks of traditional techniques:
  
  o checking return code or status flag is optional
  
  o return code mixes exceptional and normal values

• Testing and handling of return code or status flag is often done locally (inline), otherwise information may be lost;

• Nonlocal testing from nested routine calls is difficult as multiple codes are returned for analysis, compounding the mixing problem.

• Status flag can be overwritten before examined, and cannot be used in a concurrent environment because of sharing issues (e.g., save errno)

• Local fix-up routines increases the number of parameters:

• Nonlocal (global) fix-up routines, implemented with global routine pointer, have identical problems with status flags (e.g., new_handler).

• Simulate dynamic multi-level exit with return codes.
CHAPTER 2. DYNAMIC MULTI-LEVEL EXIT

2.3 Exception Handling

- Dynamic multi-level exit allows complex forms of transfers among routines.
- Complex control-flow among routines is often called exception handling.
- Exception handling is more than error handling.
- An exceptional event is an event that is (usually) known to exist but which is ancillary to an algorithm.
- An exception handling mechanism (EHM) provides some or all of the alternate kinds of control-flow.
- Very difficult to simulate EHM with simpler control structures.
- Exceptions are supposed to make certain programming tasks easier, like robust programs.
• Robustness results because exceptions are active versus passive, forcing programs to react immediately when an exceptional event occurs.

• An EHM is not a panacea and only as good as the programmer using it.

2.4 Execution Environment

• The execution environment has a significant effect on an EHM.

```java
class T {
    int *i;
    T() { i = new int[10]; ... }
    ~T() { delete [] i; ... } // must free storage
};
L: {
    T t; // constructor must be executed
    ... if ( ... ) break L;
    ...
} // destructor must be executed
```

• Control structures with `finally` clauses must always be executed (e.g., Java/μC++).

<table>
<thead>
<tr>
<th></th>
<th>Java</th>
<th>μC++</th>
</tr>
</thead>
<tbody>
<tr>
<td>L: try {</td>
<td>infile = new Scanner( new File( &quot;abc&quot; ) );</td>
<td>L: try {</td>
</tr>
<tr>
<td></td>
<td>... if ( ... ) break L;</td>
<td>infile = new ifstream( &quot;abc&quot; );</td>
</tr>
<tr>
<td></td>
<td>... possibly continue;</td>
<td>... if ( ... ) break L; // alt 1</td>
</tr>
<tr>
<td></td>
<td>finally { // always executed</td>
<td>_Finally { // always executed</td>
</tr>
<tr>
<td></td>
<td>infile.close(); // must close file</td>
<td>infile.close(); // must close file</td>
</tr>
</tbody>
</table>

• Also, complex execution-environment involving continuation, coroutine, task, each with its own execution stack.

• Given multiple stacks, an EHM can be more sophisticated, resulting in more complexity.

2.5 Terminology

• `execution` is the language unit in which an exception can be raised, usually any entity with its own runtime stack.

• `exception type` is a type name representing an exceptional event.

• `exception` is an instance of an exception type, generated by executing an operation indicating an ancillary (exceptional) situation in execution.

• `raise` (throw) is the special operation that creates an exception.

• `source execution` is the execution raising an exception.
• **faulting execution** is the execution changing control flow due to a raised exception.

• **local exception** is when an exception is raised and handled by the same execution ⇒ source = faulting.

• **nonlocal exception** is when an exception is raised by a source execution but delivered to a different faulting execution ⇒ source ≠ faulting.

• **concurrent exception** is a nonlocal exception, where the source and faulting executions are executing concurrently.

• **propagation** directs control from a raise in the source execution to a handler in the faulting execution.

• **propagation mechanism** is the rules used to locate a handler.
  
  ◦ most common propagation-mechanisms give precedence to handlers higher in the lexical/call stack

• **handler** is inline (nested) routine responsible for handling raised exception.
  
  ◦ handler catches exception by **matching** with one or more exception types
  ◦ after catching, a handler executes like a normal subroutine
  ◦ handler can return, reraise the current exception, or raise a new exception
  ◦ **reraise** terminate current handling and continue propagation of caught exception.
  ◦ an exception is **handled** only if the handler returns rather than reraises

• **guarded block** is a language block with associated handlers, e.g., try-block in C++/Java.

• **unguarded block** is a block with no handlers.

• **termination** means control cannot return to the raise point.
  
  ◦ all blocks on the faulting stack from the raise block to the guarded block handling the exception are terminated, called **stack unwinding**

• **resumption** means control returns to the raise point ⇒ no stack unwinding.

• **EHM** = **Exception Type** + Raise (exception) + Propagation + Handlers

### 2.6 Static/Dynamic Call/Return

• All routine/exceptional control-flow can be characterized by two properties:

  1. **static/dynamic call**: routine/exception name at the call/raise is looked up statically (compile-time) or dynamically (runtime).
  2. **static/dynamic return**: after a routine/handler completes, it returns to its static (definition) or dynamic (call) context.
### 2.7 Static Propagation (Sequel)

- **Case 1)** is called a **sequel**, which is a routine with no return value, where:
  - the sequel name is looked up lexically at the call site, but
  - control returns to the end of the block in which the sequel is declared.

```
A: for ( ;; ) {

B: for ( ;; ) {

C: for ( ;; ) {
  ... 
  if ( ... ) { break A; }
  ... 
  if ( ... ) { break B; }
  ... 
  if ( ... ) { break C; }
  ... 
}
}

for ( ;; ) {
  sequel S1( ... ) { ... } // nested
  void M1( ... ) {
    ... if ( ... ) S1( ... ); ...
  }
  for ( ;; ) {
    sequel S2( ... ) { ... } // nested
    C: for ( ;; ) {
      M1( ... ); // modularize
      if ( ... ) S2( ... ); // modularize
      ... 
      if ( ... ) break C;
      ... 
    }
  } // S2 static return
} // S1 static return
```

- Without a sequel, it is impossible to modularize code with static exits.
- Sequel handles termination for a **non-recoverable** event (simple exception handling).

```
{ // new block
  sequel StackOverflow(... ) { ... } // handler

  class stack {
    void push( int i ) {
      if ( ... ) StackOverflow( ... ); // 2nd outcome
    } // 1st outcome
  } // S1 static return

stack s;
... s.push( 3 ); ... // overflow ?
```

- The advantage of the sequel is the handler is statically known (like static multi-level exit), and can be as efficient as a direct transfer.
• The disadvantage is that the sequel only works for monolithic programs because it must be
  statically nested at the point of use.
  ◦ Fails for modular (library) code as the static context of the module and user code are
disjoint.
  ◦ E.g., if stack is separately compiled, the sequel call in push no longer knows the static
blocks containing calls to it.

2.8 Dynamic Propagation
• Cases 3) and 4) are called termination and resumption, and both have dynamic raise with
  static/dynamic return, respectively.
• Dynamic propagation/static return (case 3) is also called dynamic multi-level exit (see Sec-
tion 2, p. 5).
• The advantage is that dynamic propagation
• The disadvantage (advantage) of dynamic propagation

2.8.1 Termination
• For termination:
  ◦ control transfers from the start of propagation to a handler ⇒ dynamic raise (call)
  ◦ when handler returns, it performs a static return ⇒ stack is unwound (like sequel)
• There are 3 basic termination forms for a non-recoverable operation: nonlocal, terminate,
and retry.
• Nonlocal transfer provides general mechanism for block transfer on call stack, but has goto
  problem (see Section 2, p. 5).
• terminate provides limited mechanism for block transfer on the call stack, like labelled
  break (see return union examples page 10):

```c
struct E {}; // label
void f(...) {
    ...
    throw E(); // raise
    // control never returns here
}
int main() {
    try {
        f(...);
    } catch( E ) {...} // handler 1
    try {
        f(...);
    } catch( E ) {...} // handler 2
    ...
}
```
2.8. DYNAMIC PROPAGATION

- No intermediate code to forward alternative outcome.

```c++
struct NoStorage {};
struct BadComp {};
int * Malloc( size_t size ) {
    if ( random() % 2 ) return (int*)malloc( sizeof( int ) );
    throw NoStorage{};
}
int rtn( ) {
    int * p = Malloc( sizeof( int ) );
    // DO NOT HAVE TO FORWARD NoStorage
    *p = 7; // compute
    if ( random() % 2 ) return *p;
    throw BadComp{};
}
int main() {
    srand( getpid() );
    try {
        cout << rtn() << endl;
    }
    catch( BadComp ) { cout << "bad computation" << endl; }
    catch( NoStorage ) { cout << "no storage" << endl; }
}
```

- C++ I/O can be toggled to raise exceptions versus return codes (like μC++).

<table>
<thead>
<tr>
<th>C++</th>
<th>μC++</th>
</tr>
</thead>
<tbody>
<tr>
<td>ifstream infile;</td>
<td>ifstream infile;</td>
</tr>
<tr>
<td>ofstream outfile;</td>
<td>ofstream outfile;</td>
</tr>
<tr>
<td>outfile.exceptions( ios_base::failbit );</td>
<td>uFile::Failure &amp;</td>
</tr>
<tr>
<td>infile.exceptions( ios_base::failbit );</td>
<td>\</td>
</tr>
<tr>
<td>switch ( argc ) {</td>
<td>switch ( argc ) {</td>
</tr>
<tr>
<td>case 3:</td>
<td>case 3:</td>
</tr>
<tr>
<td>try {</td>
<td>try {</td>
</tr>
<tr>
<td>outfile.open( argv[2] );</td>
<td>outfile.open( argv[2] );</td>
</tr>
<tr>
<td>} catch( ios_base::failure &amp; ) {...}</td>
<td>} catch( uFile::Failure &amp; ) {...}</td>
</tr>
<tr>
<td>// fall through to handle input file</td>
<td>// fall through to handle input file</td>
</tr>
<tr>
<td>case 2:</td>
<td>case 2:</td>
</tr>
<tr>
<td>try {</td>
<td>try {</td>
</tr>
<tr>
<td>infile.open( argv[1] );</td>
<td>infile.open( argv[1] );</td>
</tr>
<tr>
<td>} catch( ios_base::failure &amp; ) {...}</td>
<td>} catch( uFile::Failure &amp; ) {...}</td>
</tr>
<tr>
<td>break;</td>
<td>break;</td>
</tr>
<tr>
<td>default:</td>
<td>default:</td>
</tr>
<tr>
<td>...</td>
<td>...</td>
</tr>
<tr>
<td>} // switch</td>
<td>} // switch</td>
</tr>
<tr>
<td>string line;</td>
<td>string line;</td>
</tr>
<tr>
<td>try {</td>
<td>for ( ; ) {</td>
</tr>
<tr>
<td>for ( ; ; ) { // loop until end-of-file</td>
<td></td>
</tr>
<tr>
<td>getline( infile, line );</td>
<td></td>
</tr>
<tr>
<td>outfile &lt;&lt; line &lt;&lt; endl;</td>
<td></td>
</tr>
<tr>
<td>}</td>
<td></td>
</tr>
<tr>
<td>} catch( ios_base::failure &amp; ) {}</td>
<td>}</td>
</tr>
</tbody>
</table>

- μC++ provides exceptions for I/O errors, but no exception for eof.
• retry is a combination of termination with special handler semantics, i.e., restart the guarded block handling the exception (Eiffel). (Pretend end-of-file is an exception of type Eof.)

<table>
<thead>
<tr>
<th>Retry</th>
<th>Simulation</th>
</tr>
</thead>
<tbody>
<tr>
<td>char readfiles( char *files[], int N ) {</td>
<td>char readfiles( char *files[], int N ) {</td>
</tr>
<tr>
<td>int i = 0, value;</td>
<td>int i = 0, value;</td>
</tr>
<tr>
<td>ifstream infile;</td>
<td>ifstream infile;</td>
</tr>
<tr>
<td>infile.open( files[i] );</td>
<td>infile.open( files[i] );</td>
</tr>
<tr>
<td>try {</td>
<td>while ( true ) {</td>
</tr>
<tr>
<td>... infile &gt;&gt; value; ...</td>
<td>... infile &gt;&gt; value; ...</td>
</tr>
<tr>
<td>} retry( Eof ) {</td>
<td>} catch( eof ) {</td>
</tr>
<tr>
<td>i += 1;</td>
<td>i += 1;</td>
</tr>
<tr>
<td>infile.close();</td>
<td>infile.close();</td>
</tr>
<tr>
<td>if ( i == N ) goto Finished;</td>
<td>if ( i == N ) break;</td>
</tr>
<tr>
<td>infile.open( files[i] );</td>
<td>infile.open( files[i] );</td>
</tr>
<tr>
<td>} Finished: ;</td>
<td>}</td>
</tr>
</tbody>
</table>

2.8.2 Resumption

• resumption provides a limited mechanism to generate new blocks on the call stack:

  o control transfers from the start of propagation to a handler ⇒ dynamic raise (call)
  o when handler returns, it is dynamic return ⇒ stack is NOT unwound (like routine)

• A resumption handler is a corrective action so a computation can continue.

```c
void f() {  
    resume E(); // raise
    // control returns here
}
int main() {  
    try {  
        f(); // no parameters
    } catch( E ) {  
        // handler 1
    }  
    try {  
        f(); // no parameters
    } catch( E ) {  
        // handler 2
    }
}
```
2.9 Exceptional Example

```java
B1 {
  try {
    try {
      try {
        try {
          . . . throw E5(); . . .
        } catch( E7 ) { . . . }
        catch( E8 ) { . . . }
        catch( E9 ) { . . . }
      } catch( E4 ) { . . . }
      catch( E5 ) { . . . throw; . . . }
      catch( E6 ) { . . . }
      catch( E3 ) { . . . }
    } catch( E5 ) { . . . resume/retry/terminate }
    catch( E2 ) { . . . }
  }
}
```

The diagram illustrates the structure of the code with
- **guarded / unguarded blocks** indicated by the blocks with and without dash lines.
- **call** and **propagation** arrows connecting the blocks.
- **stack** indicated by the vertical arrow.
- **handlers** represented by the boxes labeled with `catch`.
- **throw** and **handled** labels on the blocks.
- **resumption**, **retry**, and **terminate** labels on the boxes.
3 Coroutine

- A coroutine is a routine that can also be suspended at some point and resumed from that point when control returns.

- The state of a coroutine consists of:
  - an execution location, starting at the beginning of the coroutine and remembered at each suspend.
  - an execution state holding the data created by the code the coroutine is executing.
  - an execution status—active or inactive or terminated—which changes as control resumes and suspends in a coroutine.

- Hence, a coroutine does not start from the beginning on each activation; it is activated at the point of last suspension.

- In contrast, a routine always starts execution at the beginning and its local variables only persist for a single activation.

- A coroutine handles the class of problems that need to retain state between calls (e.g. plugin, device driver, finite-state machine).

- Two different approaches are possible for activating another coroutine:
  1. A semi-coroutine acts asymmetrically, like non-recursive routines, by implicitly reactivating the coroutine that previously activated it.
  2. A full-coroutine acts symmetrically, like recursive routines, by explicitly activating a member of another coroutine, which directly or indirectly reactivates the original coroutine (activation cycle).

- These approaches accommodate two different styles of coroutine usage.
3.1 Semi-Coroutine

3.1.1 Fibonacci Sequence

\[ f(n) = \begin{cases} 
0 & n = 0 \\
1 & n = 1 \\
f(n-1) + f(n-2) & n \geq 2 
\end{cases} \]

- 3 states, producing unbounded sequence: 0, 1, 1, 2, 3, 5, 8, 13, 21, …

3.1.1.1 Direct

- Compute and print Fibonacci numbers.

```cpp
int main() {
    int fn, fn1, fn2;
    fn = 0; fn1 = fn; // 1st case
    cout << fn << endl;
    fn = 1; fn2 = fn1; fn1 = fn; // 2nd case
    cout << fn << endl;
    for ( ;; ) { // infinite loop
        fn = fn1 + fn2; fn2 = fn1; fn1 = fn; // general case
        cout << fn << endl;
    }
}
```

- Convert to routine that generates a sequence of Fibonacci numbers on each call (no output):

```cpp
int main() {
    for ( int i = 1; i <= 10; i += 1 ) { // first 10 Fibonacci numbers
        cout << fibonacci() << endl;
    }
}
```

3.1.1.2 Routine

```cpp
int fn1, fn2, state = 1; // global variables
int fibonacci() {
    int fn;
    switch ( state ) {
    case 1:
        fn = 0; fn1 = fn; state = 2;
        break;
    case 2:
        fn = 1; fn2 = fn1; fn1 = fn; state = 3;
        break;
    case 3:
        fn = fn1 + fn2; fn2 = fn1; fn1 = fn;
        break;
    }
    return fn;
}
```
3.1. SEMI-COROUTINE

```c
#define FIB_INIT { 0, 1 } /* first two Fibonacci numbers */
struct Fibonacci { int fn2, fn1; };
int fib( Fibonacci & f ) {
    int ret = f.fn2;
    int fn = f.fn1 + f.fn2; // only last state (3) in Fibonacci definition
    f.fn2 = f.fn1; f.fn1 = fn;
    return ret;
}

int main() {
    Fibonacci f1 = FIB_INIT, f2 = FIB_INIT; // multiple instances
    for ( int i = 1; i <= 10; i += 1 ) {
        cout << fib( f1 ) << " " << fib( f2 ) << endl;
    }
}
```

3.1.1.3 Class

```c
class Fibonacci {
    int fn, fn1, fn2, state = 1; // global class variables
public:
    int operator()() { // functor
        switch ( state ) {
            case 1:
                fn = 0; fn1 = fn; state = 2;
                break;
            case 2:
                fn = 1; fn2 = fn1; fn1 = fn; state = 3;
                break;
            case 3:
                fn = fn1 + fn2; fn2 = fn1; fn1 = fn;
                break;
        }
        return fn;
    }
}
```

```c
int main() {
    Fibonacci f1, f2; // multiple instances
    for ( int i = 1; i <= 10; i += 1 ) {
        cout << f1() << " " << f2() << endl;
    } // for
}
```

3.1.1.4 Coroutine

```c
_Coroutine Fibonacci { // : public uBaseCoroutine
    int fn; // used for communication
    void main() { // distinguished member
        int fn1, fn2; // retained between resumes
        fn = 0; fn1 = fn;
        suspend(); // return to last resume
        fn = 1; fn2 = fn1; fn1 = fn;
        suspend(); // return to last resume
        for ( ; ; ) {
            fn = fn1 + fn2; fn2 = fn1; fn1 = fn;
            suspend(); // return to last resume
        }
    }
}
```
public:
    int operator()() { // functor
        resume();      // transfer to last suspend
        return fn;
    }
};

int main() {
    Fibonacci f1, f2;  // multiple instances
    for ( int i = 1; i <= 10; i += 1 ) {
        cout << f1() << " " << f2() << endl;
    }
}

• no explicit execution state! (see direct solution)

• coroutine main can be called (even recursively), but normally a private/protected member. Why?

• compile with u++ command

• All coroutines inherit from base type uBaseCoroutine:

    _Coroutine uBaseCoroutine {
        protected:
            void resume(); // context switch to this
            void suspend(); // context switch to last resumer
        public:
            uBaseCoroutine();
            uBaseCoroutine( unsigned int stackSize ); // set stack size
            void verify();              // check stack
            const char * setName( const char * name ); // printed in error messages
            const char * getName() const;
            uBaseCoroutine & starter() const; // coroutine performing first resume
            uBaseCoroutine & resumer() const; // coroutine performing last resume
    }

• Program main called from hidden coroutine ⇒ has coroutine properties.

• resume/suspend cause a context switch between coroutine stacks

• first resume starts main on new stack (cocall); subsequent resumes reactivate last suspend.

• suspend reactivates last resume
• object becomes a coroutine on first resume; coroutine becomes an object when main ends
• routine frame at the top of the stack knows where to activate execution
• suspend/resume are protected members to prevent external calls. Why?
• Coroutine main does not have to return before a coroutine object is deleted.
• When deleted, a coroutine's stack is always unwound and any destructors executed. Why?
• Warning, do not use catch(...) in a coroutine, if it may be deleted before terminating, because a cleanup exception is raised to force stack unwinding (implementation issue).

3.1.2 Format Output

Unstructured input:

abcdefghijklmnopqrstuvwxyzabcdefghijklmnopqrstuvwxyz

Structured output:

abcd  efgi  jklm  nopq  rstuvwxyz

blocks of 4 letters, separated by 2 spaces, grouped into lines of 5 blocks.

3.1.2.1 Direct

• Read characters and print formatted output.

```cpp
int main() {
    int g, b;
    char ch;
    cin >> noskipws; // turn off white space skipping
    for ( ;; ) { // for as many characters
        for ( g = 0; g < 5; g += 1 ) { // groups of 5 blocks
            for ( b = 0; b < 4; b += 1 ) { // blocks of 4 chars
                for ( ;; ) { // for newline characters
                    cin >> ch; // read one character
                    if ( cin.fail() ) goto fini; // eof ? multi-level exit
                    if ( ch != '\n' ) break; // ignore newline
                }
                cout << ch; // print character
                cout << " "; // print block separator
            }
            cout << endl; // print group separator
        }
    }
    fini: ;
    if ( g != 0 || b != 0 ) cout << endl; // special case
}
```
3.1.2.2 Routine

```cpp
int g, b;  // global variables
void fmtLines( char ch ) {
    if ( ch != -1 ) {  // not EOF ?
        if ( ch == '\n' ) return;  // ignore newline
        cout << ch;  // print character
        b += 1;
        if ( b == 4 ) {  // block of 4 chars
            cout << " ";  // block separator
            b = 0;
            g += 1;
        }
        if ( g == 5 ) {  // group of 5 blocks
            cout << endl;  // group separator
            g = 0;
        }
    } else {
        if ( g != 0 || b != 0 ) cout << endl;  // special case
    }
}

int main() {
    char ch;
    cin >> noskipws;  // turn off white space skipping
    for ( ; ; ) {  // for as many characters
        cin >> ch;
        if ( cin.fail() ) break;  // eof ?
        fmtLines( ch );
    }
    fmtLines( -1 );  // indicate EOF
}
```

3.1.2.3 Class

```cpp
class Format {
    int g, b;  // global class variables

public:
    Format() : g( 0 ), b( 0 ) {}  // default constructor
    ~Format() { if ( g != 0 || b != 0 ) cout << endl; }
    void prt( char ch ) {
        if ( ch == '\n' ) return;  // ignore newline
        cout << ch;  // print character
        b += 1;
    }
}
```
3.1. SEMI-COROUTINE

```cpp
if ( b == 4 ) {
    cout << " ";  // block separator
    b = 0;
    g += 1;
}

if ( g == 5 ) {
    cout << endl;  // group separator
    g = 0;
}
```

```cpp
int main() {
    Format fmt;
    char ch;
    cin >> noskipws;  // turn off white space skipping
    for (;;) {
        cin >> ch;  // read one character
        if ( cin.fail() ) break;  // eof ?
        fmt.prt(ch);
    }
}
```

3.1.2.4 Coroutine

```cpp
Coroutine Format {
    char ch;  // used for communication
    int g, b;  // global because used in destructor
    void main() {
        for (;;) {  // for as many characters
            for ( g = 0; g < 5; g += 1 ) {  // groups of 5 blocks
                for ( b = 0; b < 4; b += 1 ) {  // blocks of 4 characters
                    suspend();
                    if ( ch != '\n' ) break;  // ignore newline
                }
                cout << ch;  // print character
            }
            cout << " ";  // print block separator
        }
        cout << endl;  // print group separator
    }

    public:
    Format() { resume(); }  // start coroutine
    ~Format() { if ( g != 0 || b != 0 ) cout << endl; }
    void prt( char ch ) { Format::ch = ch; resume(); }
};
```
3.1.3 Correct Coroutine Usage

- **Eliminate computation or flag variables retaining information about execution state.**
- E.g., sum even and odd digits of 10-digit number, where each digit is passed to coroutine:

<table>
<thead>
<tr>
<th>BAD: Explicit Execution State</th>
<th>GOOD: Implicit Execution State</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>for (int i = 0; i &lt; 10; i += 1) {</code></td>
<td><code>for (int i = 0; i &lt; 5; i += 1) {</code></td>
</tr>
<tr>
<td><code>if (i % 2 == 0) // even</code></td>
<td><code>even += digit;</code></td>
</tr>
<tr>
<td><code>even += digit;</code></td>
<td><code>else</code></td>
</tr>
<tr>
<td><code>odd += digit;</code></td>
<td><code>suspend();</code></td>
</tr>
<tr>
<td><code>}</code></td>
<td><code>odd += digit;</code></td>
</tr>
<tr>
<td><code>}</code></td>
<td><code>suspend();</code></td>
</tr>
</tbody>
</table>

- E.g., a BAD solution for the previous Fibonacci generator is:

```c
void main() {
    int fn1, fn2, state = 1;
    for ( ;; ) {
        switch (state) { // no Zen
            case 1:
                fn = 0; fn1 = fn;
                state = 2;
                break;
            case 2:
                fn = 1; fn2 = fn1; fn1 = fn;
                state = 3;
                break;
            case 3:
                fn = fn1 + fn2; fn2 = fn1; fn1 = fn;
                break;
        }
        suspend(); // no Zen
    }
}
```
• Coroutine’s capabilities not used:
  o explicit flag variable controls execution state
  o original program structure lost in \texttt{switch} statement
• Must do more than just \textit{activate} coroutine main to demonstrate understanding of retaining data and execution state within a coroutine.

3.1.4 Coroutine Construction
• Fibonacci and formatter coroutines express original algorithm structure (no restructuring).
• When possible, simplest coroutine construction is to write a direct (stand-alone) program.
• Convert to coroutine by:
  o putting processing code into coroutine main,
  o converting reads if program is consuming or writes if program is producing to \texttt{suspend},
    * Fibonacci consumes nothing and produces (generates) Fibonacci numbers ⇒ convert writes (\texttt{cout}) to suspends.
    * Formatter consumes characters and only indirectly produces output (as side-effect) ⇒ convert reads (\texttt{cin}) to suspends.
  o use interface members and communication variables to transfer data in/out of coroutine.

3.2 \textit{\textmu}C++ EHM
The following features characterize the \textit{\textmu}C++ EHM:
• exceptions must be generated from a specific kind of type.
• supports two forms of raising: throw and resuming.
• supports two kinds of handlers, termination and resumption, which match with the kind of raise.
• supports propagation of nonlocal and concurrent exceptions.
• all exception types (user, runtime, and I/O) are grouped into a hierarchy.

3.3 Exception Type
• C++ allows any type to be used as an exception type.
• \textit{\textmu}C++ restricts exception types to those types defined by \texttt{Event}.

\begin{verbatim}
_Event exception-type-name {
   ...
};
\end{verbatim}
• An exception type has all the properties of a **class**.

• Every exception type must have a public default and copy constructor.

• An exception is the same as a class-object with respect to creation and destruction:

  ```
  _Event D { . . . };  // local creation
  D d;  // local creation
  Resume d;  // dynamic creation
  D *dp = new D;  // dynamic creation
  _Resume *dp;
  delete dp;
  _Throw D();  // temporary local creation
  ```

### 3.4 Inherited Members

• Each exception type inherits the following members from `uBaseEvent`:

  ```
  class uBaseEvent {  // like std::exception
    uBaseEvent( const char *const msg = "" );
    const char *const message() const;
    const uBaseCoroutine &source() const;
    const char *sourceName() const;
    virtual void defaultTerminate();
    virtual void defaultResume();
  };
  ```

• `uBaseEvent( const char *const msg = "" )` – `msg` is printed if the exception is not caught.

• `message` returns the string message associated with an exception.

• `source` returns the coroutine/task that raised the exception.

• `sourceName` returns the name of the coroutine/task that raised the exception.

• `defaultTerminate` is implicitly called if an exception is thrown but not handled.

• `defaultResume` is implicitly called if an exception is resumed but not handled.

### 3.5 Raising

• There are two raising mechanisms: throwing and resuming.

  ```
  _Throw [ exception-type ] ;
  _Resume [ exception-type ] [ _At uBaseCoroutine-id ] ;
  ```

• Nonlocal/concurrent raise restricted to resumption as raising execution-state is often unaware of the handling execution-state.

• Resumption allows faulting execution greatest flexibility: it can process the exception as a resumption or rethrow the exception for termination.
• Exceptions in $\mu$C++ are propagated differently from C++.

<table>
<thead>
<tr>
<th>C++</th>
<th>$\mu$C++</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>class B {};</code></td>
<td><code>_Event B {};</code></td>
</tr>
<tr>
<td><code>class D : public B {};</code></td>
<td><code>_Event D : public B {};</code></td>
</tr>
<tr>
<td><code>void f(B &amp; t) {</code></td>
<td><code>void f(B &amp; t) {</code></td>
</tr>
<tr>
<td><code>    throw t;</code></td>
<td><code>    _Throw t;</code></td>
</tr>
<tr>
<td><code>} D m;</code></td>
<td><code>} D m;</code></td>
</tr>
<tr>
<td><code>f(m);</code></td>
<td><code>f(m);</code></td>
</tr>
</tbody>
</table>

○ In C++, routine $f$ is passed an object of derived type $D$ but throws an object of base type $B$.

○ In $\mu$C++, routine $f$ is passed an object of derived type $D$ and throws the original object of type $D$.

○ This change allows handlers to catch the specific (derived) rather than the general (base) exception-type.

### 3.6 Handler

• $\mu$C++ has two kinds of handlers, termination and resumption, which match with the kind of raise.

#### 3.6.1 Termination

• The $\mu$C++ termination handler is the `catch` clause of a `try` block, i.e., same as in C++.

#### 3.6.2 Resumption

• $\mu$C++ extends the `try` block to include resumption handlers.

• Resumption handler is denoted by a `_CatchResume` clause after `try` body:

```
try {
    ...  
    _CatchResume( E1 ) { ... } // must appear before catch clauses  
    // more _CatchResume clauses  
    _CatchResume( ... ) { ... } // must be last _CatchResume clause  
    catch( E2 ) { ... }        // must appear after _CatchResume clauses  
    // more catch clauses  
    catch( ... ) { ... }       // must be last catch clause  
}
```

• Any number of resumption handlers can be associated with a `try` block.

• All `_CatchResume` handlers must precede any `catch` handlers.

• Like `catch(...) (catch-any), `_CatchResume(...) must appear at the end of the list of the resumption handlers.
• Resumption handler can access types and variables visible in its local scope.

```c
typedef int Foo;
Foo i;
try {
    f(...) // f is recursive and raises Foo
} _CatchResume( Foo & e ) { // handler H
    Foo fix = i; // use type and variable in local scope
    ... e = fix ... // change _Resume block
}
```

1. call f
2. propagation from f to handler H
3. call handler
4. dereference lexical link to i

- **lexical link** is like **this** but to declaration block rather than object.

- **Resumption handler cannot perform a break, continue, goto, or return.**
  - Resumption handler is corrective action so computation can continue.
  - If correction impossible, handler should throw an exception not step into an enclosing block to cause the stack to unwind.

```c
B: try {
    f(); // recursive calls and _Resume E()
} _CatchResume( E e ) { // handler H
    ... break B; // force static return (disallowed)
    _Throw e; // force recover (allowed)
}
```

- Handler H above makes recursive calls to f, so **goto** must unwind stack to transfer into stack frame B (nonlocal transfer).
- Throw may find another recovery action closer to raise point than B that can deal with the problem.

### 3.6.3 Termination/Resumption

- The raise dictates set of handlers examined during propagation:
  - terminating propagation (**_Throw**) only examines termination handlers (**catch**),
  - resuming propagation (**_Resume**) only examines resumption handlers (**_CatchResume**).

- Exception types in each set can overlap:
3.7. NONLOCAL EXCEPTIONS

• Resumption handler \( H1 \) is invoked by the resume in the \texttt{try} block generating call stack:

\[
\text{rtn} \rightarrow \text{try[]}_\text{CatchResume}( E ), \ \text{catch}( E ) \rightarrow H1
\]

• Handler \( H1 \) throws \( E \) and the stack is unwound until the exception is caught by termination-handler \texttt{catch}( E ) and handler \( H2 \) is invoked.

\[
\text{rtn} \rightarrow H2
\]

• The termination handler is available as resuming does not unwind the stack.

• Note interaction between resuming, \texttt{defaultResume}, and throwing:

\[
\text{rtn} \rightarrow \text{try[]}_\text{catch}( R ) \rightarrow \text{defaultResume}
\]

• \textit{When defaultResume is called, the default action throws \( R \)} (see Section 3.4, p. 30).

\[
\text{rtn} \rightarrow H1
\]

• Terminating propagation unwinds the stack until there is a match with the \texttt{catch} clause in the \texttt{try} block.

3.7 Nonlocal Exceptions

• Nonlocal exceptions are exceptions raised by a source execution at a faulting execution.

• Nonlocal exceptions are possible because each coroutine (execution) has its own stack.

• Nonlocal exceptions are raised using \texttt{_Resume \ldots _At \ldots}
For nonlocal resumption, _Resume is a proxy for actual raise in the faulting coroutine ⇒ non-local resumption becomes local resumption.

While source delivers nonlocal exception immediately, propagation only occurs when faulting becomes active.

⇒ must suspend back to or call a member that does a resume of the faulting coroutine

Faulting coroutine performs local _Resume implicitly at detection points for nonlocal exceptions, e.g., in _Enable, suspend, resume.

Handler does not return to the proxy raise; control returns to the implicit local raise at exception delivery, e.g., back in _Enable, suspend, resume.

Nonlocal delivery is initially disabled for a coroutine, so handlers can be set up before any exception can be delivered (also see Section 5.11, p. 71).

Hence, nonlocal exceptions must be explicitly enabled before delivery can occur with _Enable.
• μC++ allows dynamic enabling and disabling of individual exception types versus all exception types.

    _Enable <E1><E2>... {
        // exceptions E1, E2 are enabled
    }

    _Disable <E1><E2>... {
        // exceptions E1, E2 are disabled
    }

• _Enable and _Disable blocks can be nested, turning delivery on/off on entry and reestablishing the delivery state to its prior value on exit.

• An unhandled exception raised by a coroutine raises a nonlocal exception of type uBaseCoroutine::UnhandledException at the coroutine’s last resumer and then terminates the coroutine.

    _Event E {};

    _Coroutine C {
        void main() { _Throw E(); } // unwind
        // defaultTerminate => _Resume UnhandledException() _At resumer()
        // => activate last resumer
        // coroutine terminated
    public:
        void mem() { resume(); } // inside resume is _Resume UnhandledException();
    }

    int main() {
        C c;
        try {
            c.mem();
            } _CatchResume( uBaseCoroutine::UnhandledException & ) { ... } // one of
            } catch( uBaseCoroutine::UnhandledException & ) { ... }
            // continue catch
    }

• Call to c.mem resumes coroutine c and then coroutine c throws exception E but does not handle it.

• When the base of c’s stack is reached, an exception of type uBaseCoroutine::UnhandledException is raised at ::main, since it last resumed c.

• _CatchResume continues from resume (dynamic return, fixup)
- **catch** continues after handler (static return, recover)

- The original exception’s \( E \) default terminate routine is not called because it has been caught and transformed.

- The coroutine terminates but control returns to its last resumer rather than its starter.

- Exception **UnhandledException** (and a few others) are always enabled.

### 3.8 Memory Management

<table>
<thead>
<tr>
<th>Normal Program Stack</th>
<th>Multiple Coroutine Stacks</th>
</tr>
</thead>
<tbody>
<tr>
<td>stack</td>
<td>free</td>
</tr>
<tr>
<td>stack1</td>
<td>free</td>
</tr>
</tbody>
</table>

- Normally program stack expands to heap; but coroutine stacks expand to next stack.

- In fact, coroutine stacks are normally allocated in the heap.

- Default \( \mu C++ \) coroutine stack size is 64K and it does not grow.

```c
Coroutine C {
    public:
    C() : uBaseCoroutine(8192) {};  // default 8K stack
    C(int size) : uBaseCoroutine(size) {};  // user specified stack size
    ...
};

C x, y(16384);    // x has an 8K stack, y has a 16K stack
```

- Check for stack overflow using coroutine member **verify**:

```c
void main() {
    ...
    verify();    // check for stack overflow
    ...
    // code
}
```

### 3.9 Semi-Coroutine Examples

#### 3.9.1 Same Fringe

- Two binary trees have same fringe if all leafs are equals from left to right.
• Requires iterator to traverse a tree, return the value of each leaf, and continue the traversal.

• No direct solution without using additional data-structure (e.g., stack) to manage the tree traversal.

• Coroutine uses recursive tree-traversal but suspends traversal to return value.

```cpp
template< typename T > class Btree {
    struct Node { . . . 

public:
    _Coroutine Iterator {
        Node * cursor;
        void walk( Node * node ) { // walk tree
            if ( node == nullptr ) return;
            if ( node->left == nullptr && node->right == nullptr ) { // leaf?
                cursor = node;
                suspend(); // multiple stack frames
            } else {
                walk( node->left ); // recursion
                walk( node->right); // recursion
            }
        }
        void main() { walk( cursor ); cursor = nullptr; } 
public:
    Iterator( Btree<T> & btree ) : cursor( &btree.root ) {} 
    T* next() {
        resume();
        return cursor; 
    }
};
};
```

```cpp
template<class T> bool sameFringe( BTtree<T> & tree1, BTtree<T> & tree2 ) {
    Btree<T>::Iterator iter1( btree1 ), iter2( btree2 ); // iterator for each tree
    T * t1, * t2;
    for ( ; ; ) {
        t1 = iter1.next(); t2 = iter2.next();
        if ( t1 == nullptr || t2 == nullptr ) break; // one traversal complete ?
        if ( *t1 != *t2 ) return false; // elements equal ?
    }
    return t1 == nullptr && t2 == nullptr; // and both traversals completed ?
}
```

3.9.2 Device Driver

• Parse transmission protocol and return message text, e.g.:

```plaintext
...STX...message...ESC ETX...message...ETX 2-byte CRC...
```
### 3.9.2.1 Direct

```c
int main() {
    enum { STX = '\002', ESC = '\033', ETX = '\003' ];
    enum { MaxMsgLnth = 64 };
    unsigned char msg[MaxMsgLnth];
    ... 
    try {
        msg: for ( ;; ) {       // parse messages
            int Inth = 0, checkval;
            do {
                byte = input( infile );   // read bytes, throw Eof on eof
            } while ( byte != STX );  // message start ?
            eom: for ( ;; ) {        // scan message data
                byte = input( infile );
            switch ( byte ) {
                case STX:
                    continue msg;   // uC++ labelled continue
                case ETX:
                    break eom;      // uC++ labelled break
                case ESC:
                    byte = input( infile );
                    break;
                } // switch
            if ( Inth >= MaxMsgLnth ) { // buffer full ?
                continue msg;          // uC++ labelled continue
            } // if
            msg[Inth] = byte;        // store message
            Inth += 1;
        } // for
        byte = input( infile );   // gather check value
        checkval = byte;
        byte = input( infile );
        checkval = (checkval << 8) | byte;
        if ( ! crc( msg, Inth, checkval ) ) ... // CRC error
    } catch( Eof ) {} 
    ... 
} // main
```

### 3.9.2.2 Coroutine

- Called by interrupt handler for each byte arriving at hardware serial port.
Coroutine DeviceDriver {
    enum { STX = '\002', ESC = '\033', ETX = '\003' }
    enum { MaxMsgLnth = 64 }
    unsigned char byte;
    unsigned char * msg;

    public:
        DeviceDriver( unsigned char * msg ) : msg( msg ) {
            resume();
        }
        void next( unsigned char b ) {
            // called by interrupt handler
            byte = b;
            resume();
        }

    private:
        void main() {
            msg: for ( ;; ) {
                int lnth = 0, checkval;
                do {
                    suspend();
                } while ( byte != STX );
                eom: for ( ;; ) {
                    // scan message data
                    suspend();
                }
            }
            switch ( byte ) {
                case STX:
                    ... // protocol error
                    continue msg; // uC++ labelled continue
                case ETX:
                    // end of message
                    break eom; // uC++ labelled break
                case ESC:
                    // escape next byte
                    suspend(); // get escaped character
                    break;
            } // switch

            if ( lnth >= MaxMsgLnth ) { // buffer full ?
                ... // length error
                continue msg; // uC++ labelled continue
            } // if
            msg[lnth] = byte; // store message
            lnth += 1;
        } // for

        suspend(); // gather check value
        checkval = byte;
        suspend();
        checkval = (checkval << 8) | byte;
        if ( ! crc( msg, lnth, checkval ) ) ... // CRC error
    } // for
}; // DeviceDriver
3.9.3 Producer-Consumer

Coroutine Cons {
    int p1, p2, status; bool done;
    void main() { // starter prod
        // 1st resume starts here
        int money = 1;
        for (; ! done; ) {
            cout << "cons " << p1 << " "
                << p2 << " pay $" << money
                << endl;
            status += 1;
            suspend(); // activate delivery or stop
            money += 1;
        }
        cout << "cons stops" << endl;
    }
    // suspend / resume(starter)
public:
    Cons() : status(0), done(false) {} 
    int delivery( int p1, int p2 ) {
        Cons::p1 = p1; Cons::p2 = p2;
        resume(); // activate main
        return status;
    }
    void stop() { done = true; resume(); } // activate main
};

Coroutine Prod {
    Cons & c;
    int N;
    void main() { // starter ::main 
        // 1st resume starts here
        for ( int i = 0; i < N; i += 1 ) {
            int p1 = rand() % 100; // products
            int p2 = rand() % 100;
            cout << "prod " << p1
                << " " << p2 << endl;
            int status = c.delivery( p1, p2 );
            cout << " stat " << status
                << endl;
        }
        c.stop();
        cout << "prod stops" << endl;
    } // suspend / resume(starter)
public:
    Prod( Cons & c ) : c(c) {} 
    void start( int N ) {
        Prod::N = N;
        resume(); // activate main
    }
};
int main() {
    Cons cons;              // create consumer
    Prod prod( cons );     // create producer
    prod.start( 5 );       // start producer
}

• Do both Prod and Cons need to be coroutines?

• The starter coroutine is the coroutine that does the first resume (cocall).

• For semi-coroutines, the starter is often the last (only) resumer, so it seems coroutine main implicitly suspends on termination.

○ dashed red ⇒ create stack and resume coroutine main
○ solid red ⇒ resume coroutine at last suspend
○ solid blue ⇒ resume last resumer
○ dashed blue ⇒ resume starter

3.10 Full Coroutines

• Semi-coroutine activates the member routine that activated it.

• Full coroutine has a resume cycle; semi-coroutine does not form a resume cycle.
• A full coroutine is allowed to perform semi-coroutine operations because it subsumes the notion of semi-coroutine.

```
_Coroutine Fc {
    void main() { // starter ::main
        mem();       // ?
        resume();    // ?
        suspend();   // ?
    }

    public:
    void mem() { resume(); }
}

int main() {
    Fc fc;
    fc.mem();
}
```

• Full coroutines can form an arbitrary topology with an arbitrary number of coroutines.

• There are 3 phases to any full coroutine program.
  1. starting the cycle
2. executing the cycle
3. stopping the cycle (return to the program main)

- Starting the cycle requires each coroutine to know at least one other coroutine.

- The problem is mutually recursive references.

  \[ Fc \ x(y), \ y(x); \quad // \text{does not compile, why?} \]

- One solution is to make closing the cycle a special case.

  \[
  Fc \ x, \ y(x);
  x.\text{partner}(y);
  \]

- Once the cycle is created, execution around the cycle can begin.

- Stopping can be as complex as starting, \textit{because a coroutine goes back to its starter}.

- For full-coroutines, the starter is often \textit{not} the last resumer, so coroutine main does not appear to implicitly suspend on termination.

- But it is necessary to activate the program main to finish (unless \texttt{exit} is used).

- The starter stack always gets back to the program main.

### 3.10.1 Ping/Pong

- Full-coroutine control-flow: 2 identical coroutines:
Why is PingPong::part a pointer rather than reference?

Each coroutine cycles N times, becoming inactive in the other’s cycle member.

Assume ping’s declaration is changed to ping( "ping", N + 1 ).
3.10. FULL COROUTINES

3.10.2 Producer-Consumer

- Full-coroutine control-flow and bidirectional communication: 2 non-identical coroutines:

```cpp
_Coroutine Prod {
    Cons * c;
    int N, money, receipt;
    void main() { // starter ::main
        // 1st resume starts here
        for ( int i = 0; i < N; i += 1 ) {
            int p1 = rand() % 100; // products
            int p2 = rand() % 100;
            cout << "prod " << p1 << " prod " << p2 << endl;
            int status = c->delivery(p1, p2);
            cout << "prod rec $" << money << " stat " << status << endl;
            receipt += 1;
        }
        c->stop();
        cout << "prod stops" << endl;
    }
}

_Coroutine Cons {
    Prod & p;
    int p1, p2, status;
    bool done;
    void main() { // starter prod
        // 1st resume starts here
        int money = 1, receipt;
        for ( ; ! done; ) {
            cout << "cons " << p1 << " "
                 << p2 << " pay $"
                 << money << endl;
            status += 1;
            receipt = p.payment(money);
            cout << "cons #"
                 << receipt << endl;
            money += 1;
        }
        cout << "cons stops" << endl;
    }
}

public:
    int payment( int money ) {
        Prod::money = money;
        resume(); // Prod::main 1st time, then
        return receipt; // prod in Cons::delivery
    }

    void start( int N, Cons & c ) {
        Prod::N = N; Prod::c = &c;
        receipt = 0;
        resume(); // activate main
    }
}
```

- Cheat using forward reference for Cons at c->delivery and c->stop. Fix by?

```cpp
table
<table>
<thead>
<tr>
<th>start</th>
<th>N, c</th>
</tr>
</thead>
<tbody>
<tr>
<td>main</td>
<td>cons(p,p1,p2, status, done)</td>
</tr>
<tr>
<td></td>
<td>prod(c, N, money, receipt)</td>
</tr>
<tr>
<td>delivery</td>
<td>p1, p2</td>
</tr>
<tr>
<td>resume</td>
<td>money</td>
</tr>
</tbody>
</table>

diagram
```
• Black dashed-line same control flow as ping/pong.

• Remove flag variable from full-coroutine producer-consumer.

```
(Event Stop {});
(Coroutine Prod {
  Cons * c;
  int N, money, receipt;
  void main() {
    for ( int i = 0; i < N; i += 1 ) {
      int p1 = rand() % 100;
      int p2 = rand() % 100;
      cout << "prod " << . . .
      int status = c->delivery(p1, p2);
      cout << "prod rec $" << . . .
      receipt += 1;
    }
  }
}
(Resume Stop()) _At resumer();
suspend(); // restart cons
} catch( Stop & ) {}
cout << "cons stops" << endl;
}

Coroutine Cons {
  Prod & p;
  int p1, p2, status = 0;
  void main() {
    int money = 1, receipt;
    try {
      for ( ;; ) {
        cout << "cons " << p1 << . . .
        status += 1;
        receipt = p.payment( money );
        cout << "cons #" << . . .
        money += 1;
        _Enable {} // trigger exception
      }
    } catch( Stop & ) {}
cout << "cons stops" << endl;
}

public:
  int payment( int money ) {
    Prod::.money = money;
    resume();
    return receipt;
  }
  void start( int N, Cons & c ) {
    Prod::.N = N; Prod::.c = &c;
    receipt = 0;
    resume();
  }
};
```
3.11 Coroutine Languages

- Coroutine implementations have two forms:

  1. stackless: use the caller’s stack and a fixed-sized local-state
  2. stackful: separate stack and a fixed-sized local-state

- Stackless coroutines cannot call other routines and then suspend, i.e., only suspend in the coroutine main.

- Generators/iterators are often simple enough to be stackless using `yield`.

- Simula, CLU, C#, Ruby, Python, JavaScript, Lua, F#, Boost all support `yield` constructs.

3.11.1 Python 3.5

- Stackless, semi coroutines, routine versus class, no calls, single interface

- Fibonacci (see Section 3.1.1.4, p. 23)

  ```python
  def Fibonacci( n ): # coroutine main
      fn = 0; fn1 = fn
      yield fn # suspend
      fn = 1; fn2 = fn1; fn1 = fn
      yield fn # suspend
      # while True:
      # for infinite generator
      for i in range( n - 2 ): # for infinite generator
          fn = fn1 + fn2; fn2 = fn1; fn1 = fn
          yield fn # suspend
  f1 = Fibonacci( 10 ) # objects
  f2 = Fibonacci( 10 )
  for i in range( 10 ):
      print( next( f1 ), next( f2 ) ) # resume
  for fib in Fibonacci( 15 ): # use generator as iterator
      print( fib )
  ```

- Format (see Section 3.1.2.4, p. 27)
def Format():
    try:
        while True:
            for g in range(5):
                # groups of 5 blocks
                for b in range(4):
                    # blocks of 4 characters
                    print((yield), end='')  # receive from send
                    print(' ', end='')  # block separator
            print()  # group separator
    except GeneratorExit:
        # destructor
        if g != 0 | b != 0:
            # special case
            print()
    fmt = Format()
    next(fmt)  # prime generator
    for i in range(41):
        fmt.send('a')  # send to yield

• send takes only one argument, and no cycles ⇒ no full coroutine

3.11.2 C++ Boost Library (V1.61)

• stackfull, semi/full coroutines, routine versus class, single interface

• return activates creator

• full coroutine uses resume( partner, . . . ), specifying target coroutine to be activated

• coroutine state passed as explicit parameter

• Fibonacci (semi-coroutine, asymmetric)

int fibonacci(coroutine<int>::push_type & suspend) {
    int fn = 0, fn1 = fn, fn2;  // 1st case
    suspend(fn);
    fn = 1; fn2 = fn1; fn1 = fn;  // 2nd case
    suspend(fn);
    for (; ; ) {
        fn = fn1 + fn2; fn2 = fn1; fn1 = fn;  // general case
        suspend(fn);
    }
}

int main() {
    coroutine<int>::pull_type fib(fibonacci);
    // declaration does first resume
    for (int i = 0; i < 15; i += 1) {
        cout << fib.get() << " ";  // get output
        fib();  // resume
    }
    cout << endl;
}

• Formatter (semi-coroutine, asymmetric)
void format( coroutine< char >::pull_type & suspend ) {
    int g, b; char ch;
    for ( ;; ) {
        for ( g = 0; g < 5; g += 1 ) {
            for ( b = 0; b < 4; b += 1 ) {
                for ( ;; ) {
                    ch = suspend.get();
                    if ( ch == '\377' ) goto fini;
                    suspend();
                    if ( ch != '\n' ) break;
                }
                cout << ch;
                cout << " ";
            }
            cout << endl;
        }
    }
    fini:
    if ( g != 0 || b != 0 ) cout << endl; // special case
}

int main() {
    coroutine< char >::push_type fmt( format );
    char ch;
    cin >> noskipws; // no white space skipping
    for ( ;; ) {
        cin >> ch;
        if ( cin.fail() ) break;
        fmt( ch );
    }
    fmt( '\377' ); // resume sentential
}

• Producer/Consumer (full-coroutine, symmetric)

typedef tuple< int, int > ConsParms;
typedef tuple< int, int > ProdParms;
typedef symmetric_coroutine< ConsParms > Cons;
typedef symmetric_coroutine< ProdParms > Prod;

void prod_main( Prod::yield_type & resume, Cons::call_type & cons ) {
    int money, status, receipt = 0;
    for ( int i = 0; i < 5; i += 1 ) {
        int p1 = rand() % 100, p2 = rand() % 100;
        cout << "prod delivers " << p1 << ", " << p2 << endl;
        tie( money, status ) = resume( cons, { p1, p2, receipt } ).get();
        cout << "prod gets money $" << money << " status " << status << endl;
        receipt += 1;
    }
    resume( cons, { -1, -1, -1 } );
    cout << "prod stops" << endl;
}
```cpp
void cons_main( Cons::yield_type & resume, Prod::call_type & prod ) {
    int p1, p2, receipt, money = 1, status = 0;
    for ( int i = 0; i < 5; i += 1 ) {
        tie( p1, p2, receipt ) = resume.get();
        cout << "cons receives " << p1 << "", " << p2 << " and pays $" << money << " gets receipt #" << receipt << endl;
        status += 1;
        resume( prod, { money, status } );
        money += 1;
    }
    cout << "cons stops" << endl;
    resume( prod, { -1, -1 } );
}

int main() {
    Prod::call_type prod;
    Cons::call_type cons;
    // lambda contains partner in closure
    prod = Prod::call_type(
        [&]( Prod::yield_type & resume ) { prod_main( resume, cons ); } );
    cons = Cons::call_type(
        [&]( Cons::yield_type & resume ) { cons_main( resume, prod ); } );
    prod( { -1, -1 } );
    cons( { -1, -1 } );
}
```
4 More Exceptions

4.1 Derived Exception-Type

- **derived exception-types** is a mechanism for inheritance of exception types, like inheritance of classes.

- Provides a kind of polymorphism among exception types:

  ![Exception Hierarchy](diagram)

  - Exception-type inheritance allows a handler to match multiple exceptions, e.g., a base handler can catch both base and derived exception-type.

  - To handle this case, most propagation mechanisms perform a linear search of the handlers for a guarded block and select the first matching handler.

    ```
    try {
        throw D(); // _Throw in uC++
    } catch( B e ) { // truncation
        // cannot down-cast
    } catch( B & e ) { // no truncation
        ... dynamic_cast<D>(e) ...
    }
    ```

    - When subclassing, it is best to catch an exception by reference:

      ```
      struct B {}; struct D : public B {};
      try {
        throw D(); // _Throw in uC++
      } catch( Arithmetic & ) { ... }
      catch( Overflow ) { ... // never selected!!! }
      }
      ```

- Notice, catching truncation is different from raising truncation, which does not occur in µC++ with _Throw.

4.2 Catch-Any

- **catch-any** is a mechanism to match any exception propagating through a guarded block.

- With exception-type inheritance, catch-any can be provided by the root exception-type, e.g., `catch( Exception )` in Java.

- Otherwise, special syntax is needed, e.g., `catch( ... )` in C++.

- For termination, catch-any is used as a general cleanup when a non-specific exception occurs.
• For resumption, this capability allows a guarded block to gather or generate information about control flow (e.g., logging).

```c
try {
  ...
} _CatchResume(...) { // catch–any
  ... // logging
  _Resume; // reresume for fixup
} catch(...) { // catch–any
  ... // cleanup
  _Throw; // rethrow for recovery
}
```

• Java finalization:

```c
try { ... } catch( E ) { ... } ...
  // other catch clauses
} finally { // always executed
  ... // cleanup
  // possibly rethrow
}
```

provides catch-any capabilities and handles the non-exceptional case.

  ○ difficult to mimic in C++, even with RAII, because of local variables.

4.3 Exception Parameters

• Exception parameters allow passing information from the raise to a handler.

• Inform a handler about details of the exception, and to modify the raise site to fix an exceptional situation.

• Different EHMs provide different ways to pass parameters.

• In C++/Java, parameters are defined inside the exception:

```c
struct E {
  int i;
  E( int i ) : i(i) {}
};
void f(...) { ... throw E( 3 ); ... } // argument
int main() {
  try {
    f(...);
  } catch( E p ) { // parameter, value or reference
    ... p.i ...
  }
}
```

• For resumption, values at raise modified via reference/pointer in caught exception:
4.4 Exception List

- Missing exception handler for arithmetic overflow in control software caused Ariane 5 rocket to self-destruct ($370 million loss).

- **exception list** is part of a routine’s prototype specifying which exception types may propagate from the routine to its caller.

  ```cpp
g(int x) throw(E) {
    try {
      f();
    } _CatchResume( E & e ) {
      e.r = 3; ...
    }
  }
```

  This capability allows:

  - static detection of a raised exception not handled locally or by its caller
  - runtime detection where the exception may be converted into a special **failure exception** or the program terminated.

- 2 kinds of checking:

  - checked/unchecked exception-type (Java, inheritance based, static check)
  - checked/unchecked routines (C++, exception-list based, dynamic check) (deprecated C++11, replaced with **noexcept**)

- While checked exception-types are useful for software engineering, reuse is precluded.

- E.g., consider the simplified C++ template routine sort:

  ```cpp
  template<class T> void sort( T items[] ) throw( ?, ?, ... ) {
    // using bool operator<( const T &a, const T &b );
  }
```

  using the operator routine < in its definition.

- Impossible to know all exception types that propagated from routine < for every type.
• Since only a fixed set of exception types can appear in sort’s exception list, some sortable types are precluded.

• Exception lists can preclude reuse for arguments of routine pointers (functional style) and/or polymorphic methods/routines (OO style):

```c++
// throw NO exceptions
void f( void (*p)() throw() ) {
    p();
}
void g() throw(E) { throw E(); }
void h() {
    try { ... f( g ); ... }
    catch( E ) {}  
}
  // throw NO exceptions
struct B {  // throw NO exceptions
    virtual void g() throw() {} 
    void f() { g(); }
};
struct D : public B {
    void g() throw(E) { throw E(); }
    void h() {
        try { ... f(); ... }
        catch( E ) {} 
    }
};
```

• Finally, determining an exception list for a routine can become impossible for concurrent exceptions because they can propagate at any time.

4.5 Destructor

• Destructor is implicitly noexcept ⇒ cannot raise an exception.

• Destructor can raise an exception, if marked noexcept(false), or inherits from class with noexcept(false) destructor.

```c++
struct E {};
struct C {
    ~C() noexcept(false) { throw E(); } 
};
  // outer try  
try {  
    C x; // raise on deallocation  
    try {  
        C y; // raise on deallocation  
        } catch( E ) {...}  // inner handler  
    } catch( E ) {...}  // outer handler  
```

4.6 Multiple Exceptions

• An exception handler can generated an arbitrary number of nested exceptions.
4.6. MULTIPLE EXCEPTIONS

```cpp
struct E {};
int cnt = 3;
void f( int i ) {
    if ( i == 0 ) throw E();
    try {
        f( i - 1 );
    } catch( E ) { // handler h
        cnt -= 1;
        if ( cnt > 0 ) f( 2 );
    }
}

int main() { f( 2 ); }
```

- Exceptions are nested as handler can rethrow its matched exception when control returned.
- However, multiple exceptions cannot propagate simultaneously.
- Only destructor code can intervene during propagation.
- Hence, a destructor **cannot** raise an exception during propagation; it can only start propagation.

```cpp
try {
    C x; // raise on deallocation
    throw E();
} catch( E ) {...}
```

- Raise of E causes unwind of inner **try** block.
- x’s destructor called during unwind, it raises an exception E, which one should be used?
  - Cannot start second exception without handler to deal with first exception, i.e., cannot drop exception and start another.
  - Cannot postpone first exception because second exception may remove its handlers during stack unwinding.
- Check if exception is being propagated with **uncaught_exception()**.
5 Concurrency

- A **thread** is an independent sequential execution path through a program.
  - Each thread is scheduled for execution separately and independently from other threads.
- A **process** is a program component (like a routine) that **has its own thread** and has the same state information as a coroutine.
- A **task** is similar to a process except that it is
  - reduced along some particular dimension (like the difference between a boat and a ship, one is physically smaller than the other).
  - It is often the case that a process has its own memory, while tasks share a common memory.
  - A task is sometimes called a light-weight process (LWP).
- **Parallel execution** is when 2 or more operations occur simultaneously, which can only occur when multiple processors (CPUs) are present.
- **Concurrent execution** is any situation in which execution of multiple threads *appears* to be performed in parallel.
  - It is the threads of control associated with processes and tasks that results in concurrent execution, *not the processors*.

5.1 Why Write Concurrent Programs

- Dividing a problem into multiple executing threads is an important programming technique just like dividing a problem into multiple routines.

5.2 Why Concurrency is Difficult

- to understand:
  - While people can do several things concurrently, the number is small because of the difficulty in managing and coordinating them.
- to specify:
  - How can/should a problem be broken up so that parts of it can be solved at the same time as other parts?
  - How and when do these parts interact or are they independent?
  - If interaction is necessary, what information must be communicated during the interaction?
- to debug:
Concurrent operations proceed at varying speeds and in non-deterministic order, hence execution is not repeatable (Heisenbug).

Reasoning about multiple streams or threads of execution and their interactions is much more complex than for a single thread.

- E.g. Moving furniture out of a room; can’t do it alone, but how many helpers and how to do it quickly to minimize the cost?
- How many helpers?
- Where are the bottlenecks?
- What communication is necessary between the helpers?

5.3 Concurrent Hardware

- Concurrent execution of threads is possible with only one CPU (uniprocessor); multitasking for multiple tasks or multiprocessing for multiple processes.

- Parallelism is simulated by context switching the threads on the CPU.
- Most of the issues in concurrency can be illustrated without parallelism.
- Unlike coroutines, task switching may occur at non-deterministic program locations, i.e., between any two machine instructions.
- Introduces all the difficulties in concurrent programs.
  * programs must be written to work regardless of non-deterministic ordering of program execution.
- Switching happens explicitly but conditionally when calling routines.
- Switching can happen implicitly because of an external interrupt independent of program execution.
- If interrupts affect scheduling (execution order), it is called preemptive, otherwise the scheduling is non-preemptive.
- Programmer cannot predict execution order, unlike coroutines.
- Granularity of context-switch is instruction level for preemptive (harder to reason) and routine level for non-preemptive.

- In fact, every computer has multiple CPUs: main CPU(s), bus CPU, graphics CPU, disk CPU, network CPU, etc.
5.4. EXECUTION STATES

- Concurrent/parallel execution of threads is possible with multiple CPUs sharing memory (multiprocessor):

- Concurrent/parallel execution of threads is possible with single/multiple CPUs on different computers with separate memories (distributed system):

5.4 Execution States
- A thread may go through the following states during its execution.

- State transitions are initiated in response to events (interrupts):
  - timer alarm (running → ready)
  - completion of I/O operation (blocked → ready)
  - exceeding some limit (CPU time, etc.) (running → halted)
  - error (running → halted)

- Non-deterministic “ready ↔ running” transition ⇒ basic operations unsafe:

```c
int i = 0; // shared
task0
i += 1
```

- If increment implemented with single inc i instruction, transitions can only occur before or after instruction, not during.
• If increment is replaced by a load-store sequence, transitions can occur during sequence.

\[
\begin{align*}
\text{ld} & \quad r1,i \quad \text{// load into register 1 the value of } i \\
\ldots \quad \text{// PREEMPTION} \\
\text{add} & \quad r1,#1 \quad \text{// add 1 to register 1} \\
\ldots \quad \text{// PREEMPTION} \\
\text{st} & \quad r1,i \quad \text{// store register 1 into } i
\end{align*}
\]

• Remember, context switch saves and restores registers for each coroutine/task.

\[
\begin{array}{ll}
\text{task0} & \quad \text{task1} \\
\text{1st iteration} & \text{1st iteration} \\
\text{ld} & \quad r1,i \quad (r1 <- 0) \\
\text{add} & \quad r1,#1 \quad (r1 <- 1) \\
\text{st} & \quad r1,i \quad (i <- 1) \\
\text{2nd iteration} & \text{2nd iteration} \\
\text{ld} & \quad r1,i \quad (r1 <- 1) \\
\text{add} & \quad r1,#1 \quad (r1 <- 2) \\
\text{st} & \quad r1,i \quad (i <- 2) \\
\text{3rd iteration} & \text{3rd iteration} \\
\text{ld} & \quad r1,i \quad (r1 <- 2) \\
\text{add} & \quad r1,#1 \quad (r1 <- 3) \\
\text{st} & \quad r1,i \quad (i <- 3)
\end{array}
\]

• The 3 iterations of task1 are lost when overwritten by task0.

• Hence, sequential operations, however small (increment), are unsafe in a concurrent program.

5.5 Threading Model

• For multiprocessor systems, a threading model defines relationship between threads and CPUs.

• OS manages CPUs providing logical access via kernel threads (virtual processors) scheduled across the CPUs.
5.6. CONCURRENT SYSTEMS

- More kernel threads than CPUs to provide multiprocessing, i.e., run multiple programs simultaneously.
- A process may have multiple kernel threads to provide parallelism if multiple CPUs.
- A program may have user threads scheduled on its process’s kernel threads.
- User threads are a low-cost structuring mechanism, like routines, objects, coroutines (versus high-cost kernel thread).
- Relationship is denoted by user:kernel:CPU, where:
  - 1:1:C (kernel threading) – 1 user thread maps to 1 kernel thread
  - N:N:C (generalize kernel threading) – N × 1:1 kernel threads (Java/Pthreads)
  - M:1:C (user threading) – M user threads map to 1 kernel thread (no parallelism)
  - M:N:C (user threading) – M user threads map to N kernel threads (Go, \( \mu \)C++)

- Often the CPU number (C) is omitted.

- Can recursively add nano threads (stackless) on top of user threads (stackful), and virtual machine below OS.

5.6 Concurrent Systems

- Concurrent systems can be divided into 3 major types:
  1. those that attempt to discover implicit concurrency in an otherwise sequential program, e.g., parallelizing loops and access to data structures
  2. those that provide concurrency through implicit constructs, which a programmer uses to build a concurrent program
3. those that provide concurrency through *explicit* constructs, which a programmer uses to build a concurrent program

- **To solve all concurrency problems, threads need to be explicit.**
- Both implicit and explicit mechanisms are complementary, and hence, can appear together in a single programming language.
- Some concurrent systems provide a single technique or paradigm that must be used to solve all concurrent problems.
- Therefore, a good concurrent system must support a variety of different concurrent approaches, while at the same time not requiring the programmer to work at too low a level.
- In all cases, as concurrency increases, so does the complexity to express and manage it.

### 5.7 Speedup

- Program *speedup* is $S_C = T_1 / T_C$, where $C$ is number of CPUs and $T_1$ is sequential execution.
- E.g., 1 CPU takes 10 seconds, $T_1 = 10$ (*user time*), 4 CPUs takes 2.5 seconds, $T_4 = 2.5$ (*real time*) $\Rightarrow S_4 = 10 / 2.5 = 4$ times speedup (linear).

![Speedup Chart]

- Aspects affecting speedup (assume sufficient parallelism for concurrency):
  1. amount of concurrency
  2. critical path among concurrency
  3. scheduler efficiency
- An algorithm/program is composed of sequential and concurrent sections.
- **Amdahl’s law** (Gene Amdahl): concurrent section of program is $P$ making sequential section $1 - P$, then maximum speedup using $C$ CPUs is:

$$S_C = \frac{1}{(1-P) + \frac{P}{C}} \text{ where } T_1 = 1, T_C = \text{sequential} + \text{concurrent}$$
### 5.7. SPEEDUP

- Normalize: \( T_1 = \frac{10}{10} = 1, T_4 = \frac{2.5}{10} = .25 \).

\[
S_4 = \frac{1}{(1-1) + 1 \times .25} = 4 \text{ times, } P = 1 \Rightarrow (100\%) \text{ of } T_4 \text{ is concurrent}
\]

- Change \( P = .8(80\%) \) so \( T_4/C = .8 \times .25 = .2 \) is concurrent and \( 1 - .8 = .2(20\%) \) is sequential.

\[
S_4 = \frac{1}{(1-.8) + .8 \times .25} = \frac{1}{.2 + .2} = 2.5 \text{ times, because of sequential code}
\]

- As \( C \) goes to infinity, \( P/C \) goes to 0, so maximum speedup is \( 1/(1-P) \), i.e., time for sequential section.

- E.g., an algorithm/program has 4 stages: \( t1 = 10, t2 = 25, t3 = 15, t4 = 50 \) (time units)

- Concurrently speedup sections \( t2 \) by 5 times and \( t4 \) by 10 times.

\[
T_C = 10 + \frac{25}{5} + 15 + \frac{50}{10} = 35 \text{ (time units)}
\]

\[
\text{Speedup} = \frac{100}{35} = 2.86 \text{ times}
\]

- Large reductions for \( t2 \) and \( t4 \) have only minor effect on speedup.

- Formula does not consider any increasing costs for the concurrency, i.e., administrative costs, so results are optimistic.

- While sequential sections bound speedup, concurrent sections bound speedup by the critical path of computation.

- **independent execution**: all threads created together and do not interact.

- **dependent execution**: threads created at different times and interact.

- Longest path bounds speedup (even for independent execution).
Finally, speedup can be affected by scheduler efficiency/ordering (often no control), e.g.:

- Therefore, it is difficult to achieve significant speedup for many algorithms/programs.

5.8 Thread Creation

Concurrency requires 3 mechanisms in a programming language.

1. **creation** – cause another thread of control to come into existence.
2. **synchronization** – establish timing relationships among threads, e.g., same time, same rate, happens before/after.
3. **communication** – transmit data among threads.

Thread creation must be a primitive operation; cannot be built from other operations in a language.

5.8.1 COBEGIN/COEND

Compound statement with statements run by multiple threads.

```c
#include <uCobegin.h>
int i;
void p1(...); void p2(...); void p3(...);
// initial thread creates threads
COBEGIN  // threads execute statement in block
    BEGIN i = 1; . . . END
    BEGIN p1( 5 ); . . . END  // order and speed of internal
    BEGIN p2( 7 ); . . . END   // thread execution is unknown
    BEGIN p3( 9 ); . . . END
    // order and speed of internal
COEND   // initial thread waits for all internal threads to
        // finish (synchronize) before control continues
```

- implicit or explicit concurrency?
- A **thread graph** represents thread creations:

[Diagram of thread graph]

• A thread graph represents thread creations:
5.8. THREAD CREATION

- Use recursion to create dynamic number of threads.

```c
void loop( int N ) {
    if ( N != 0 ) {
        COBEGIN
            BEGIN p1( . . . ); END
            BEGIN loop( N - 1 ); END // recursive call
        COEND // wait for return of recursive call
    }
}
```

```c
cin >> N;
loop( N );
```

- What does the thread graph look like?

5.8.2 START/WAIT

- Start thread in routine and wait (join) at thread termination, allowing arbitrary thread graph:

```c
#include <uCobegin.h>
int i;
void p( int i ) {...}
int f( int i ) {...}
auto tp = START( p, 5 ); thread starts in p(5)
s1 continue execution, do not wait for p
auto tf = START( f, 8 ); thread starts in f(8)
s2 continue execution, do not wait for f
WAIT( tp ); wait for p to finish
s3
i = WAIT( tf ); wait for f to finish
s4
```

- implicit or explicit concurrency?

- COBEGIN/COEND can only approximate this thread graph:

```c
COBEGIN
    BEGIN p( 5 ); END
    BEGIN s1; COBEGIN BEGIN f( 8 ); s2; END COEND END // wait for f!
COEND
s3; s4;
```

- START/WAIT can simulate COBEGIN/COEND:

```c
COBEGIN
    BEGIN p1( . . . ) END
    BEGIN p2( . . . ) END
    auto t1 = START( p1, . . . )
    auto t2 = START( p2, . . . )
    WAIT t1
    WAIT t2
    COEND
```

5.8.3 Thread Object

- C++ is an object-oriented programming language, which suggests:
○ wrap the thread in an object to leverage all class features
○ use object allocation/deallocation to define thread lifetime rather than control structure

```cpp
_Task T { // thread type
    void main() {...} // thread starts here }

COBEGIN
{ // { int i, j, k; } ???
    T t; // create object on stack, start thread
    // wait for thread to finish
}
COEND

START T * t = new T; // create thread object on heap, start thread
WAIT delete t; // wait for thread to finish
```

• Block-terminate/delete must wait for each task’s thread to finish. Why?

• Unusual to:
  ○ create object in a block and not use it
  ○ allocate object and immediately delete it.

• Simulate COBEGIN/COEND with _Task object by creating type for each statement:

```cpp
int i;
_Task T1 {
    void main() { i = 1; }
};
_Task T2 {
    void main() { p1(5); }
};
_Task T3 {
    void main() { p2(7); }
};
_Task T4 {
    void main() { p3(9); }
};

int main() {
    { // COBEGIN
        T1 t1; T2 t2; T3 t3; T4 t4;
    } // COEND
};

void p1(...) {
    { // COBEGIN
        T5 t5; T6 t6; T7 t7; T8 t8;
    } // COEND
}
```

• Simulate START/WAIT with _Task object by creating type for each call:

```cpp
int i;
_Task T1 {
    void main() { p(5); }
};
_Task T2 {
    int temp;
    void main() { temp = f(8); }
    public:
        ~T2() { i = temp; }
};

int main() {
    T1 * tp = new T1; // start T1
    ... s1 ...
    T2 * tf = new T2; // start T2
    ... s2 ...
    delete tp; // wait for p
    ... s3 ...
    delete tf; // wait for f
    ... s4 ...
};
```
5.8.4 Actor

- An **actor** (Hewitt/Agha) is a unit of work without a thread.

- Two popular programming languages with actors are Erlang and Scala.

- Communication is via polymorphic queue of messages (mailbox) ⇒ dynamic type-checking.

![Diagram of actor communication]

- Usually no shared information among actors and no blocking is allowed.

```cpp
#include <uActor.h>
struct StrMsg : public uActor::Message { // derived message
  string val; // string message
  StrMsg( string val ) : Message( uActor::Delete ), // delete after use
          val( val ) {}
};

_Actor Hello { // like COBEGIN / COEND
  Allocation receive( Message & msg ) { // receive base type
    Case( StrMsg, msg ) { // discriminate derived message
      ... msg_d->val; ... // access derived message
    } else Case( StopMsg, msg ) return Delete; // delete actor
    return Nodelete; // reuse actor
  }
};

int main() { // like COBEGIN / COEND
  uActorStart(); // start actor system
  *new Hello() | *new StrMsg( "hello" ) | uActor::stopMsg;
  *new Hello() | *new StrMsg( "bonjour" ) | uActor::stopMsg;
  uActorStop(); // wait for all actors to terminate
}
```

- Actor must receive at least one message to start.

- (StartMsg) uActor::startMsg / (StopMsg) uActor::stopMsg persistent predefined messages.

- Most actor systems leverage garbage collection to manage actors and messages, and the actor system ends after all actors terminate.

- C++ does not have garbage collection so µC++ actors and messages involve explicit storage management returning an allocation status for each actor and message.
class uActor {
    public:
        enum Allocation { Nodelete, Delete, Destroy, Finished }; // allocation actions
    struct Message {
        Allocation allocation; // allocation action
        ...
    }
    private:
        Allocation allocation; // allocation action
};

Nodelete ⇒ actor or message persists after an actor returns from receive. Use for multi-
use actors or messages during their life time. (message default)

Delete ⇒ actor or message is deleted after an actor returns from receive. Use with
dynamically allocated actors or messages at completion.

Destroy ⇒ actor’s or message’s destructor is called after an actor returns from receive
but storage is not deallocated. Use with placement allocated actors or messages at
completion.

Finished ⇒ actor is marked finished after it returns from receive but neither the destruc-
tor is called nor storage deallocated. (No action for a message.) Use with stack
allocated actors or messages at completion.

#include <uActor.h>
struct StrMsg : public uActor::Message { // default Nodelete
    string val;
    StrMsg( string val ) : val( val ) {}
};
.Actor Hello {
    Allocation receive( Message & msg ) {
        Case( StrMsg, msg ) {
            ... msg_d->val ...;
        }
        return Finished; // no delete/destroy but remove from actor system
    }
};

int main() {
    uActorStart();
    Hello hellos[2]; // stack allocate actors and messages
    StrMsg hello( "hello" ), bonjour( "bonjour" );
    hellos[0] | hello;
    hellos[1] | bonjour;
    uActorStop();
} // DEALLOCATE ACTORS/MESSAGES

- One shot actor with single string message (no stopMsg).
5.9 Termination Synchronization

- A thread terminates when:
  - Children can continue to exist even after the parent terminates (although this is rare).
    - E.g. sign off and leave child process(es) running
- Synchronizing at termination is possible for independent threads.
- Termination synchronization may be used to perform a final communication.

5.10 Divide-and-Conquer

- Divide-and-conquer is characterized by ability to subdivide work across data ⇒ work can be performed independently on the data.
- Only termination synchronization is required to know when the work is done
- Partial results are then processed further if necessary.
- Sum rows of a matrix concurrently using concurrent statement:

```c
#include <uCobegin.h>
int main() {
    const int rows = 10, cols = 10;
    int matrix[rows][cols], subtotals[rows], total = 0;
    // read matrix
    COFOR( r, 0, rows,
        // for ( int r = 0; r < rows; r += 1 )
        subtotals[r] = 0; // r is loop number
        for ( int c = 0; c < cols; c += 1 )
            subtotals[r] += matrix[r][c];
    ); // wait for threads
    for ( int r = 0; r < rows; r += 1 ) {
        total += subtotals[r]; // total subtotals
    }
    cout << total << endl;
}
```

- COFOR logically creates end – start threads, indexed start..end – 1 one per loop body.
- implicit or explicit concurrency?
• Sum rows of a matrix concurrently using actors:

```cpp
_Actor Adder {
    int * row, cols, & subtotal; // communication
    Allocation receive( Message & ) { // only startMsg
        subtotal = 0;
        for ( int c = 0; c < cols; c += 1 ) subtotal += row[c];
        return Delete; // delete actor (match new)
    }
}
public:
    Adder( int row[], int cols, int & subtotal ) :
        row( row ), cols( cols ), subtotal( subtotal ) {} 
};
```

```cpp
int main() {
    // same
    uActorStart(); // start actor system
    for ( int r = 0; r < rows; r += 1 ) { // actor per row
        *new Adder( matrix[r], cols, subtotals[r] ) | uActor::startMsg;
    }
    uActorStop(); // wait for all actors to terminate
    ... // same
} // main
```

• implicit or explicit concurrency?

• Sum rows of a matrix concurrently using concurrent objects:

```cpp
_Task Adder {
    int * row, cols, & subtotal; // communication
    void main() {
        subtotal = 0;
        for ( int c = 0; c < cols; c += 1 ) subtotal += row[c];
    }
}
public:
    Adder( int row[], int cols, int & subtotal ) :
        row( row ), cols( cols ), subtotal( subtotal ) {} 
};
```

```cpp
int main() {
    // same
    Adder * adders[rows];
    for ( int r = 0; r < rows; r += 1 ) { // start threads to sum rows
        adders[r] = new Adder( matrix[r], cols, subtotals[r] );
    }
    for ( int r = 0; r < rows; r += 1 ) { // wait for threads to finish
        delete adders[r];
        total += subtotals[r]; // total subtotals
    }
    cout << total << endl;
}
5.11. EXCEPTIONS

```cpp
int main() {
    ... // same
    {
        unique_ptr<Adder> adders[rows];
        for (int r = 0; r < rows; r += 1) { // start threads to sum rows
            adders[r] = make_unique<Adder>( matrix[r], cols, subtotals[r] );
        } // wait for tasks to terminate
        for (int r = 0; r < rows; r += 1) {
            total += subtotals[r]; // total subtotals
        }
    }
}
```

- implicit or explicit concurrency?
- Why are the tasks created in the heap?
- Does it matter in what order adder tasks are created?
- Does it matter in what order adder tasks are deleted? (critical path)

### 5.11 Exceptions

- Exceptions can be handled locally within a task, or nonlocally among coroutines, or concurrently among tasks.

  - Local task exceptions are the same as for a class.
    - An unhandled exception raised by a task terminates the program.
  - Nonlocal exceptions are possible because each coroutine/task has its own stack (execution state)
  - Nonlocal exceptions between a task and a coroutine are the same as between coroutines (single thread).
  - Concurrent exceptions among tasks are more complex due to the multiple threads.
  - A concurrent exception provides an additional kind of communication among tasks.
  - For example, two tasks may begin searching for a key in different sets:
5.12 Synchronization and Communication During Execution

- Synchronization occurs when one thread waits until another thread has reached a certain execution point (state and code).

```plaintext
bool Insert = false, Remove = false;
int Data;

_Task Prod {
    int N;
    void main() {
        for ( int i = 1; i <= N; i += 1 ) {
            Data = i; // transfer data
            Insert = true;
            while ( ! Remove ) {} // busy wait
            Remove = false;
        }
    }
}

public:
    Prod( int N ) : N( N ) {}
};

_Task Cons {
    int N;
    void main() {
        for ( int i = 1; i <= N; i += 1 ) {
            while ( ! Insert ) {} // busy wait
            Insert = false;
            while ( ! Remove ) {} // busy wait
            data = Data; // remove data
            Remove = true;
        }
    }
}

public:
    Cons( int N ) : N( N ) {}
};

int main() {
    Prod prod( 5 ); Cons cons( 5 );
}
```

- 2 infinite loops!

- A loop waiting for an event among threads is called a busy wait.

- Are 2 synchronization flags necessary?

5.13 Communication

- Once threads are synchronized there are many ways that information can be transferred from one thread to the other.

- If the threads are in the same memory, then information can be transferred by value or address (e.g., reference parameter).
• If the threads are not in the same memory (distributed), then transferring information by value is straightforward but by address is difficult.

5.14 Critical Section

• Threads may access non-concurrent objects, like a file or linked-list.

• There is a potential problem if there are multiple threads attempting to operate on the same object simultaneously.

• Not a problem if the operation on the object is atomic (not divisible).

• Where an operation is composed of many instructions, it is often necessary to make the operation atomic.

• A group of instructions on an associated object (data) that must be performed atomically is called a critical section.

• Preventing simultaneous execution of a critical section by multiple threads is called mutual exclusion.

• Must determine when concurrent access is allowed and when it must be prevented.

• Improve by differentiating between reading and writing
  
  ○ allow multiple readers or a single writer; still wasteful as a writer may only write at the end of its usage.

• Need to minimize the amount of mutual exclusion (i.e., make critical sections as small as possible, Amdahl’s law) to maximize concurrency.

5.15 Static Variables

• Warning: static variables in a class are shared among all objects generated by that class.

• These shared variables may need mutual exclusion for correct usage.

• However, a few special cases where static variables can be used safely, e.g., task constructor.

• If task objects are generated serially, static variables can be used in the constructor.

• E.g., assigning each task its own name:
_Task T {
  static int tid;
  string name; // must supply storage

  public:
  T() {
    name = "T" + to_string( tid ); // shared read
    setName( name.c_str() ); // name task
    tid += 1; // shared write
  }

  ...}

int T::tid = 0; // initialize static variable in .C file
T t[10]; // 10 tasks with individual names

- Task constructor is executed by the creating thread, so array constructors executed sequentially.

- This approach only works if one task creates all the objects and initialization data is internal.

- Instead of static variables, pass a task identifier to the constructor:

    T::T( int tid ) { ... } // create name
    T * t[10]; // 10 pointers to tasks
    for ( int i = 0; i < 10; i += 1 ) {
      t[i] = new T(i); // with individual names
    }

- In general, it is best to avoid using shared static variables in a concurrent program.

5.16 Mutual Exclusion Game

- Is it possible to write code guaranteeing a statement (or group of statements) is always serially executed by 2 threads?

- Rules of the Game:

  1. Only one thread can be in a critical section at a time with respect to a particular object (safety).

  2. Threads may run at arbitrary speed and in arbitrary order, while the underlying system guarantees a thread makes progress (i.e., threads get some CPU time).

  3. If a thread is not in the entry or exit code controlling access to the critical section, it may not prevent other threads from entering the critical section.

  4. In selecting a thread for entry to a critical section, a selection cannot be postponed indefinitely (liveness). Not satisfying this rule is called indefinite postponement or livelock.

  5. After a thread starts entry to the critical section, it must eventually enter. Not satisfying this rule is called starvation.
• **Indefinite postponement and starvation are related by busy waiting.**

• Unlike synchronization, looping for an event in mutual exclusion *must* ensure eventual progress.

• Threads waiting to enter can be serviced in any order, as long as each thread eventually enters.

• If threads are *not* serviced in first-come first-serve (FCFS) order of arrival, there is a notion of **unfairness**

• Unfairness implies waiting threads are overtaken by arriving threads, called **barging**.

### 5.17 Self-Testing Critical Section

```c
void CriticalSection() {
    static uBaseTask * curr = &uThisTask(); // shared
    for ( int i = 1; i <= 100; i += 1 ) {
        if ( curr != &uThisTask() ) {
            // check
            abort( "interference" );
        }
        ... // work
    }
}
```

• What is the minimum number of interference tests and where?

• Why are multiple tests useful?

### 5.18 Software Solutions

#### 5.18.1 Lock

```c
enum Yale { CLOSED, OPEN } Lock = OPEN; // shared
__Task PermissionLock {
    void main() {
        for ( int i = 1; i <= 1000; i += 1 ) {
            while ( ::Lock == CLOSED ) {} // entry protocol
            ::Lock = CLOSED;
            CriticalSection(); // critical section
            ::Lock = OPEN; // exit protocol
        }
    }
    public:
    PermissionLock() {};
    int main() {
        PermissionLock t0, t1;
    }
```
5.18.2 Alternation

```c++
int Last = 0; // shared

_Task Alternation {
    int me;

    void main() {
        for ( int i = 1; i <= 1000; i += 1 ) {
            while ( ::Last == me ) {} // entry protocol
           CriticalSection(); // critical section
            ::Last = me; // exit protocol
        }
    }
}

public:
    Alternation(int me) : me(me) {};

int main() {
    Alternation t0( 0 ), t1( 1 );
}
```

5.18.3 Declare Intent

```c++
enum Intent { WantIn, DontWantIn };

_Task DeclIntent {
    Intent & me, & you;
    void main() {
        for ( int i = 1; i <= 1000; i += 1 ) {
            me = WantIn; // entry protocol
            while ( you == WantIn ) {} // critical section
            me = DontWantIn; // exit protocol
        }
    }
}

public:
    DeclIntent( Intent & me, Intent & you ) :
        me(me), you(you) {};

int main() {
    Intent me = DontWantIn, you = DontWantIn;
    DeclIntent t0( me, you ), t1( you, me );
}
```
5.18.4 Retract Intent

```cpp
enum Intent { WantIn, DontWantIn };

_Task RetractIntent {
    Intent & me, & you;
    void main() {
        for ( int i = 1; i <= 1000; i += 1 ) {
            for ( ;; ) {
                // entry protocol
                me = WantIn;
                if ( you == DontWantIn ) break;
                me = DontWantIn;
                while ( you == WantIn ) {}
            }
            CriticalSection();  // critical section
            me = DontWantIn;    // exit protocol
        }
    }
}

public:
    RetractIntent( Intent & me, Intent & you ) : me(me), you(you) {};

int main() {
    Intent me = DontWantIn, you = DontWantIn;
    RetractIntent t0( me, you ), t1( you, me );
}
```

5.18.5 Prioritized Retract Intent

```cpp
enum Intent { WantIn, DontWantIn };
enum Priority { HIGH, low };

_Task PriorityEntry {
    Intent & me, & you; Priority priority;
    void main() {
        for ( int i = 1; i <= 1000; i += 1 ) {
            for ( ;; ) {
                // entry protocol
                me = WantIn;
                if ( you == DontWantIn ) break;
                if ( priority == low ) {
                    me = DontWantIn;
                    while ( you == WantIn ) {} // busy wait
                }
            }
            CriticalSection();  // critical section
            me = DontWantIn;    // exit protocol
        }
    }
}

public:
    PriorityEntry( Priority p, Intent & me, Intent & you ) : priority(p), me(me), you(you) {};

int main() {
    Intent me = DontWantIn, you = DontWantIn;
    PriorityEntry t0( HIGH, me, you ), t1( low, you, me );
} // main
```
5.18.6 Dekker (modified retract intent)

```cpp
enum Intent { WantIn, DontWantIn };  
Intent * Last;

_Task Dekker {
    Intent & me, & you;
    void main() {
        for ( int i = 1; i <= 1000; i += 1 ) {
            for ( ;; ) {  // entry protocol, high priority
                me = WantIn;  // READ FLICKER
                if ( you == DontWantIn ) break;  // does not want in
                if ( ::Last == &me ) {  // low priority task?
                    me = DontWantIn;  // retract intent, READ FLICKER
                    while ( ::Last == &me // low priority busy wait
                            && you == WantIn ) {}
                }
            }
            CriticalSection();
            if ( ::Last != &me )  // exit protocol
                ::Last = &me;  // READ FLICKER
            me = DontWantIn;  // READ FLICKER
        }
    }
}

class Dekker( Intent & me, Intent & you ) : me(me), you(you) {
};

int main() {
    Intent me = DontWantIn, you = DontWantIn;
    ::Last = &me;  // arbitrary who starts as last
    Dekker t0( me, you ), t1( you, me );
}
```

- Dekker’s algorithm appears **RW-safe**.
  - On cheap multi-core computers, read/write is not atomic.
  - Hence, simultaneous writes scramble bits, and for simultaneous read/write, read sees flickering bits during write.
  - RW-safe means a mutual-exclusion algorithm works for non-atomic read/write.
  - Dekker has no simultaneous W/W because intent reset after alternation in exit protocol.
  - Dekker has simultaneous R/W but all are equality so works if final value never flickers.

- 2015 Hesselink found failure case if final value flickers:
• RW-safe version (Hesselink)

  ○ line 6: add conjunction \( \text{you} == \text{WantIn} \Rightarrow \text{stop spinning} \)

  ○ line 8: add conditional assignment to ::Last

    \[
    \begin{array}{l}
    \text{T}_0 \\
    7 \text{ Critical Section} \\
    9 \text{:Last} = \&\text{me} \quad \text{(flicker you } \text{T}_1) \\
    \text{(flicker me } \text{T}_0) \\
    10 \text{ me} = \text{DontWantIn} \quad \text{(repeat)} \\
    \end{array} \\
    \begin{array}{l}
    \text{T}_1 \\
    3 \text{ you} == \text{DontWantIn} \ (\text{true}) \\
    7 \text{ Critical Section} \\
    9 \text{:Last} = \&\text{me} \\
    3 \text{ you} == \text{DontWantIn} \ (\text{false}) \\
    4 \text{:Last} = \&\text{me} \ (\text{true}) \\
    6 \text{ low priority wait} \\
    6 \text{:Last} = \&\text{me} \ (\text{true}, \text{spin forever}) \\
    \end{array}
    \]

  ○ \( \text{T}_1 \) starvation (rule 5)

  ○ Not assigning at line 9 when ::Last != \&me prevents flicker so \( \text{T}_1 \) makes progress.

• Dekker has unbounded overtaking (not starvation) because \text{race loser retracts intent}.

• \( \Rightarrow \) thread exiting critical does not exclude itself for reentry.

  ○ \( \text{T}_0 \) exits critical section and attempts reentry
  ○ \( \text{T}_1 \) is now high priority (Last != me) but delays in low-priority busy-loop and resetting its intent.
  ○ \( \text{T}_0 \) can enter critical section unbounded times until \( \text{T}_1 \) resets its intent
  ○ \( \text{T}_1 \) sets intent \( \Rightarrow \) bound of 1 as \( \text{T}_1 \) can be entering or in critical section
5.18.7 Peterson (modified declare intent)

```cpp
eenum Intent { WantIn, DontWantIn }
Intent * Last;

_Task Peterson {
    Intent & me, & you;
    void main() {
        for ( int i = 1; i <= 1000; i += 1 ) {
            me = WantIn;  // entry protocol, order matters
            ::Last = &me;  // RACE!
            while ( you == WantIn && ::Last == &me ) {}
            CriticalSection();  // critical section
            me = DontWantIn;  // exit protocol
        }
    }
}
public:
    Peterson( Intent & me, Intent & you ) : me(me), you(you) {}
};
int main() {
    Intent me = DontWantIn, you = DontWantIn;
    Peterson t0(me, you), t1(you, me);
}
```

- Peterson’s algorithm is **RW-unsafe** requiring atomic read/write operations.
- Peterson has **bounded overtaking** because **race loser does not retracts intent**.
- ⇒ thread exiting critical excludes itself for reentry.
  - T0 exits critical section and attempts reentry
  - T0 runs race by itself and loses
  - T0 must wait (Last == me)
  - T1 eventually sees (Last != me)
- Bounded overtaking is allowed by rule 3 because the prevention is occurring in the entry protocol.
- Can line 2 be moved before 1?

```
1 2  ::Last = &me;  // RACE!
1 1  me = WantIn;  // entry protocol
3 3  while ( you == WantIn && ::Last == &me ) {}  
4 4  CriticalSection();  // critical section
5 5  me = DontWantIn;  // exit protocol
```
  - T0 executes Line 1 ⇒ ::Last = T0
  - T1 executes Line 1 ⇒ ::Last = T1
  - T1 executes Line 2 ⇒ T1 = WantIn
  - T1 enters CS, because T0 == DontWantIn
  - T0 executes Line 2 ⇒ T0 = WantIn
  - T0 enters CS, because ::Last == T1
5.18.8 N-Thread Prioritized Entry

```cpp
enum Intent { WantIn, DontWantIn };
_Task NTask { // Burns–Lynch/Lamport: B–L
    Intent * intents;
    int N, priority, i, j;
    void main() {
        // Burns–Lynch/Lamport: B–L
        intents[N, priority, i, j];
        void main() {
            for (i = 1; i <= 1000; i += 1) {
                // step 1, wait for tasks with higher priority
                do {
                    // entry protocol
                    intents[priority] = WantIn;
                    // check if task with higher priority wants in
                    for (j = priority - 1; j >= 0; j -= 1) {
                        if (intents[j] == WantIn) {
                            intents[priority] = DontWantIn;
                            while (intents[j] == WantIn) {}
                            break;
                        }
                    }
                } while (intents[priority] == DontWantIn);
                // step 2, wait for tasks with lower priority
                for (j = priority + 1; j < N; j += 1) {
                    while (intents[j] == WantIn) {}
                }
                CriticalSection();
                intents[priority] = DontWantIn; // exit protocol
            }
        }
    }
}

public:
    NTask( Intent i[], int N, int p ) : intents(i), N(N), priority(p) {}
};
```

- Only $N$ bits needed.
• No known solution for all 5 rules using only \( N \) bits.

• Other \( N \)-thread solutions use more memory.
  (best: 3-bit RW-unsafe, 4-bit RW-safe).

### 5.18.9 \( N \)-Thread Bakery (Tickets)

```c
_Task Bakery { // (Lamport) Hehner–Shyamasundar
    int * ticket, N, priority;
    void main() {
        for ( int i = 0; i < 1000; i += 1 ) {
            // step 1, select a ticket
            ticket[priority] = 0;  // highest priority
            int max = 0;           // O(N) search
            for ( int j = 0; j < N; j += 1 ) { // for largest ticket
                int v = ticket[j];        // can change so copy
                if ( v != INT_MAX && max < v ) max = v;
            }
            max += 1;               // advance ticket
            ticket[priority] = max;
            // step 2, wait for ticket to be selected
            for ( int j = 0; j < N; j += 1 ) { // check tickets
                while ( ticket[j] < max ||
                        (ticket[j] == max && j < priority) ) {} // other tasks proceed
            }
            CriticalSection();
            ticket[priority] = INT_MAX;  // exit protocol
        }
    }
}

public:
    Bakery( int t[], int N, int p ) : ticket(t), N(N), priority(p) {} // exit protocol
};
```

\[
\begin{array}{ccccccccccc}
\text{HIGH} & 0 & 1 & 2 & 3 & 4 & 5 & 6 & 7 & 8 & 9 \\
\text{priority} & \infty & \infty & 17 & \infty & 0 & 18 & 18 & 0 & 20 & 19
\end{array}
\]

• \( NM \) bits, where \( M \) is the ticket size (e.g., 32 bits)

• Lamport RW-safe

• Hehner/Shyamasundar RW-unsafe

  assignment \( \text{ticket[priority]} = \text{max} \) can flickers to \( \text{INT_MAX} \Rightarrow \) other tasks proceed

### 5.18.10 Tournament

• Binary (d-ary) tree with \( \lceil N/2 \rceil \) start nodes and \( \lceil \lg N \rceil \) levels.
• Thread assigned to start node, where it begins mutual exclusion process.

• Each node is like a Dekker or Peterson 2-thread algorithm.

• Tree structure tries to find compromise between fairness and performance.

• Exit protocol must retract intents in reverse order.

• Otherwise race between retracting/released threads along same tree path:
  
  - T₀ retracts its intent (left) at D₁,
  - T₁ (right) now moves from D₁ to D₄, sets its intent at D₄ (left), and with no competition at D₄ proceeds to D₆ (left),
  - T₀ (left) now retracts the intent at D₄ set by T₁,
  - T₂/₃ continue from D₂, sets its intent at D₄ (right), and with no competition at D₄ (left) proceeds to D₆, which ultimately violates mutual exclusion.

• No overall livelock because each node has no livelock.

• No starvation because each node guarantees progress, so each thread eventually reaches the root.

• Tournament algorithm RW-safety depends on MX algorithm; tree traversal is local to each thread.

• Tournament algorithms have unbounded overtaking as no synchronization among the nodes of the tree.

• For a minimal binary tree, the tournament approach uses \((N - 1)M\) bits, where \((N - 1)\) is the number of tree nodes and \(M\) is the node size (e.g., intent, turn).
CHAPTER 5. CONCURRENcy

_Task TournamentMax { // Taubenfeld – Buhr
    struct Token { int intents[2], turn; }; // intents/turn
    static Token * t; // triangular matrix
    int depth, id;

    void main() {
        unsigned int lid; // local id at each tree level
        for ( int i = 0; i < 1000; i += 1 ) {
            lid = id; // entry protocol
            for ( int lv = 0; lv < depth; lv += 1 ) {
                binary_prologue( lid & 1, &t[lv][lid >> 1] );
                lid >>= 1; // advance local id for next tree level
            }
           CriticalSection( id );
        for ( int lv = depth - 1; lv >= 0; lv -= 1 ) { // exit protocol
            lid = id >> lv; // retract reverse order
            binary_epilogue( lid & 1, &t[lv][lid >> 1] );
        }
        }
    }
}

public:
    TournamentMax( struct Token * t[], int depth, int id ) :
        t( t ), depth( depth ), id( id ) {};

• Can be optimized to 3 shifts and exclusive-or using Peterson 2-thread for binary.

• Path from leaf to root is fixed per thread ⇒ table lookup possible using max or min tree.

5.18.11 Arbiter

• Create full-time arbitrator task to control entry to critical section.

    bool intents[N], serving[N]; // initialize to false

_Task Client {
    int me;
    void main() {
        for ( int i = 0; i < 100; i += 1 ) {
            intents[me] = true; // entry protocol
            while ( ! serving[me] ) { // busy wait
                CriticalSection();
            serving[me] = false; // exit protocol
        }
    }
}

public:
    Client( int me ) : me( me ) {}
5.19. HARDWARE SOLUTIONS

- **Task Arbiter**

  ```
  void main() {
    int i = N; // force cycle to start at id=0
    for ( ;; ) {
      do {
        i = (i + 1) % N; // advance next client
        while ( ! intents[i] ); // not want in ?
        intents[i] = false; // retract intent on behalf of client
        serving[i] = true; // wait for exit from critical section
        while ( serving[i] ) {} // busy wait
      } while ( ! intents[i] );
    }
  }
  ```

- Mutual exclusion becomes synchronization between arbiter and clients.

- RW-unsafe due to read flicker.

- Cost is creation, management, and execution (continuous busy waiting) of arbiter task.

5.19 Hardware Solutions

- Software solutions to the critical-section problem rely on
  - shared information,
  - communication among threads,
  - (maybe) atomic memory-access.

- Hardware solutions introduce level below software level.

- Cheat by making assumptions about execution impossible at software level.

- Allows elimination of much of the shared information and the checking of this information required in the software solution.

- Special instructions to perform an **atomic read and write operation**.

- Sufficient for multitasking on a single CPU.
5.19.1 Test/Set Instruction

- Simple lock of critical section fails:

```c
int Lock = OPEN; // shared
// each task does
while ( Lock == CLOSED ); // fails to achieve (read)
Lock = CLOSED; // mutual exclusion (write)
// critical section
Lock = OPEN;
```

- The test-and-set instruction performs an atomic read and fixed assignment.

```c
int Lock = OPEN; // shared

int TestSet( int & b ) {
    // begin atomic
    int temp = b;
    b = CLOSED;
    // end atomic
    return temp;
}
```

```c
void Task::main() { // each task does
    while( TestSet( Lock ) == CLOSED );
    // critical section
    Lock = OPEN;
}
```

- In multiple CPU case, hardware (bus) must also guarantee multiple CPUs cannot interleave these special R/W instructions on same memory location.

5.19.2 Swap Instruction

- The swap instruction performs an atomic interchange of two separate values.

```c
int Lock = OPEN; // shared

void Swap( int & a, & b ) {
    int temp;
    // begin atomic
    temp = a;
    a = b;
    b = temp;
    // end atomic
}
```

```c
void Task::main() { // each task does
    int dummy = CLOSED;
    do {
        Swap( Lock, dummy );
    } while( dummy == CLOSED );
    // critical section
    Lock = OPEN;
}
```

5.19.3 Fetch and Increment Instruction

- The fetch-and-increment instruction performs an increment between the read and write.

```c
int Lock = 0; // shared

int FetchInc( int & val ) {
    // begin atomic
    int temp = val;
    val += 1;
    // end atomic
    return temp;
}
```

```c
void Task::main() { // each task does
    while ( FetchInc( Lock ) != 0 );
    // critical section
    Lock = 0;
}```
• Often fetch-and-increment is generalized to add any value \( \Rightarrow \) also decrement with negative value.

• Use ticket counter to solve both problems (Bakery Algorithm, see Section 5.18.9, p. 82):

```c++
class ticketLock {
    unsigned int tickets, serving;
public:
    ticketLock() : tickets(0), serving(0) {} // entry protocol
    void acquire() {
        int ticket = fetchInc(tickets); // obtain a ticket
        while (ticket != serving) {} // busy wait
    }
    void release() { // exit protocol
        serving += 1;
    }
};
```
6 Locks

- Package software/hardware locking into abstract type for general use.
- Locks are constructed for synchronization or mutual exclusion or both.

6.1 Lock Taxonomy

<table>
<thead>
<tr>
<th>Spinning</th>
<th>Blocking (Queueing)</th>
</tr>
</thead>
<tbody>
<tr>
<td>no yield</td>
<td>synchronization</td>
</tr>
<tr>
<td>yield</td>
<td>semaphore</td>
</tr>
<tr>
<td></td>
<td>mutex</td>
</tr>
<tr>
<td></td>
<td>(others)</td>
</tr>
<tr>
<td></td>
<td>condition</td>
</tr>
<tr>
<td></td>
<td>barrier</td>
</tr>
<tr>
<td></td>
<td>binary</td>
</tr>
<tr>
<td></td>
<td>counting</td>
</tr>
<tr>
<td></td>
<td>owner</td>
</tr>
</tbody>
</table>

- Spinning locks busy wait until an event occurs ⇒ task oscillates between ready and running states due to time slicing.
- Blocking locks do not busy wait, but block until an event occurs ⇒ some other mechanism must unblock waiting task when the event happens.

6.2 Spin Lock

- A spin lock is implemented using busy waiting, which loops checking for an event to occur.

```
while ( TestSet( Lock ) == CLOSED ); // use up time-slice (no yield)
```

- So far, when a task is busy waiting, it loops until:
  - critical section becomes unlocked or an event happens.
  - waiting task is preempted (time-slice ends) and put back on ready queue.

- To increase uniprocessor efficiency, a task can:
  - explicitly terminate its time-slice
  - move back to the ready state after only one event-check fails. (Why one?)

- Task member yield relinquishes time-slice by rescheduling running task back onto ready queue.

```
while( TestSet( Lock ) == CLOSED ) uThisTask().yield(); // relinquish time-slice
```

- To increase multiprocessor efficiency, a task can yield after N event-checks fail. (Why N?)
- Some spin-locks allow adjustment of spin duration, called adaptive spin-lock.
6.2.1 Implementation

- **µC++** provides a non-yielding spin lock, uSpinLock, and a yielding spin lock, uLock.

```cpp
class uSpinLock {
  public:
  uSpinLock(); // open
  void acquire();
  bool tryacquire();
  void release();
};

class uLock {
  public:
  uLock( unsigned int value = 1 );
  void acquire();
  bool tryacquire();
  void release();
};
```

- Both locks are built directly from an atomic hardware instruction.
- Lock starts closed (0) or opened (1); waiting tasks compete to acquire lock after release.
- synchronization

```cpp
_task T1 {  
  uLock & lk;
  void main() {
    ...
    lk.release();
    ...
  }
  public:
  T1( uLock & lk ) : lk(lk) {} }

_task T2 {  
  uLock & lk;
  void main() {
    ...
    lk.acquire();
    S1
    ...
  }
  public:
  T2( uLock & lk ) : lk(lk) {} }

int main() {
  uLock lock( 0 ); // closed
  T1 t1( lock );
  T2 t2( lock );
}
```

- mutual exclusion

```cpp
_task T {  
  uLock & lk;
  void main() {
    ...
    lk.acquire();
    // critical section
    lk.release();
    ...
    lk.acquire();
    // critical section
    lk.release();
    ...
  }
  public:
  T( uLock & lk ) : lk(lk) {} }

int main() {
  uLock lock( 1 ); // open
  T t0( lock ), t1( lock );
}
```
6.3 Blocking Locks

- For spinning locks,
  - acquiring task(s) is solely responsible for detecting an open lock after the releasing task opens it.

- For blocking locks,
  - acquiring task makes one check for open lock and blocks
  - releasing task has sole responsibility for detecting blocked acquirer and transferring lock, or just releasing lock.

- Blocking locks reduce busy waiting by having releasing task do additional work: cooperation.
  - What advantage does the releasing task get from doing the cooperation?

- Therefore, all blocking locks have
  - state to facilitate lock semantics
  - list of blocked acquirers

![Diagram of blocking lock and tasks](block_list.png)

- Which task is scheduled next from the list of blocked tasks?

6.3.1 Mutex Lock

- Mutex lock is used solely to provide mutual exclusion.

- Restricting a lock to just mutual exclusion:
  - separates lock usage between synchronization and mutual exclusion
  - permits optimizations and checks as the lock only provides one specialized function

- Mutex locks are divided into two kinds:
  - single acquisition: task that acquired the lock cannot acquire it again
  - multiple acquisition: lock owner can acquire it multiple times, called an owner lock
• Multiple acquisition can handle looping or recursion involving a lock:

```c
void f() {
    lock.acquire();
    . . . f(); // recursive call within critical section
    lock.release();
}
```

• May require only one release to unlock, or as many releases as acquires.

### 6.3.1.1 Implementation

• Multiple acquisition lock manages owner state (blue).

```c
class MutexLock {
    bool avail; // resource available?
    Task * owner; // lock owner
    queue<Task> blocked; // blocked tasks
    SpinLock lock; // mutex nonblocking lock

public:
    MutexLock() : avail(true), owner(nullptr) {}  // resource available?
    void acquire() {
        lock.acquire(); // barging
        while (! avail && owner != currThread()) { // busy waiting
            // add self to lock's blocked list
            yieldNoSchedule(); // do not reschedule to ready queue
            lock.acquire(); // reacquire spinlock
        }
        avail = false;
        owner = currThread(); // set new owner
        lock.release();
    }

    void release() {
        lock.acquire();
        if (owner != currThread()) . . . // ERROR CHECK
            owner = nullptr; // no owner
        if (! blocked.empty()) {
            // remove task from blocked list and make ready
        }
        avail = true; // reset
        lock.release(); // RACE
    }
};
```

• yieldNoSchedule yields the processor time-slice but does not reschedule thread to ready queue.

• Single or multiple unblock for multiple acquisition?

• avail is necessary as queue can be empty but critical section occupied.
• Problem: **blocking occurs holding spin lock!**

• ⇒ release lock before blocking

  // add self to blocked list of lock
  lock.release();       // allow releasing task to unblock next waiting task
  // PREEMPTION ⇒ put on ready queue
  yieldNoSchedule();

• **Race between blocking and unblocking tasks.**

• Need *magic* to atomically yield without scheduling *and* release spin lock.

• Magic is often accomplished with more cooperation:

  yieldNoSchedule( lock );

• Note, the runtime system violates order and speed of execution by being non-preemptable.

• Problem: avail and lock reset ⇒ acquiring tasks can **barge** ahead of released task.

• Released task must check again *(while)* ⇒ busy waiting ⇒ starvation

• **Barging avoidance** (cooperation): hold avail between releasing and unblocking task (bounded overtaking)

  ```c
  void acquire() {
    lock.acquire();       // baring
    if ( ! avail && owner != currThread() ) {
      // avoid barging
      // add self to lock’s blocked list
      yieldNoSchedule( lock );
      lock.acquire();       // reacquire spinlock
    }
    avail = false;
    owner = currThread();  // set new owner
    lock.release();
  }

  void release() {
    lock.acquire();
    owner = nullptr;      // no owner
    if ( ! blocked.empty() ) {
      // remove task from blocked list and make ready
    } else {
      avail = true;         // conditional reset
    }
    lock.release();        // RACE
  }
  ```

• **Barging prevention** (cooperation): hold lock between releasing and unblocking task
Example code for acquiring and releasing a lock:

```c
void acquire() {
    lock.acquire(); // prevention barging
    if ( ! avail && owner != currThread() ) { // add self to lock's blocked list
        yieldNoSchedule( lock );
        // DO NOT REACQUIRE LOCK
    }
    avail = false;
    owner = currThread(); // set new owner
    lock.release();
}

void release() {
    lock.acquire();
    owner = nullptr; // no owner
    if ( ! blocked.empty() ) { // remove task from blocked list and make ready
        // DO NOT RELEASE LOCK
    } else {
        avail = true; // conditional reset
        lock.release(); // NO RACE
    }
}
```

- Spin lock is **conceptually passed** from releasing to unblocking tasks (baton passing).

- Bargers cannot enter ⇒ no starvation as released task does not require lock.

- **Critical section is not bracketed by the spin lock when lock is passed.**

- Alternative (cooperation): leave lock owner at front of blocked list to act as availability and owner variable.
6.3. BLOCKING LOCKS

class MutexLock {
    queue<Task> blocked; // blocked tasks
    SpinLock lock;       // nonblocking lock
public:
    void acquire() {
        lock.acquire();    // prevention barging
        if (blocked.empty()) { // no one waiting ?
            node.owner = currThread();
            // add self to lock's blocked list
        } else if (blocked.head().owner != currThread()) { // not owner ?
            // add self to lock's blocked list
            yieldNoSchedule(lock);
            // DO NOT REACQUIRE LOCK
        }
        lock.release();
    }
    void release() {
        lock.acquire();    // REMOVE TASK FROM HEAD OF BLOCKED LIST
        if (!blocked.empty()) {
            // MAKE TASK AT FRONT READY BUT DO NOT REMOVE
            // DO NOT RELEASE LOCK
        } else {
            lock.release(); // NO RACE
        }
    }
};

6.3.1.2 uOwnerLock

• µC++ provides a multiple-acquisition mutex-lock, uOwnerLock:

class uOwnerLock {
public:
    uOwnerLock();
    uBaseTask *owner();
    unsigned int times();
    void acquire();
    bool tryacquire();
    void release();
};

• Must release as many times as acquire.

• Otherwise, operations same as for uLock but with blocking instead of spinning for acquire.

6.3.1.3 Mutex-Lock Release-Pattern

• To ensure a mutual exclusion lock is always released use the following patterns.

  o executable statement – finally clause
Lock always released on normal, local transfer (break/return), and exception.

Cannot be used for barging prevention. Why?

6.3.1.4 Stream Locks

Specialized mutex lock for I/O based on uOwnerLock.

Concurrent use of C++ streams can produce unpredictable results.

if two tasks execute:

\[
\begin{align*}
\text{task}_1 & : \text{cout} \ll "abc" \ll "def" \ll \text{endl} ; \\
\text{task}_2 & : \text{cout} \ll "uvw" \ll "xyz" \ll \text{endl} ;
\end{align*}
\]

any of the outputs can appear:

\[
\begin{array}{c|c|c|c|c|c}
\text{abc def} & \text{abc uvw xyz} & \text{uvw abc xyz def} & \text{abuvw c def} & \text{uvw abc def} \\
\text{uvw xyz} & \text{def} & \text{uvw c ab} & \text{xyz} & \text{y z}
\end{array}
\]

\[\mu\text{C++ provides: osacquire for output streams and isacquire for input streams.}\]

Most common usage is to create an anonymous stream lock for a cascaded I/O expression:

\[
\begin{align*}
\text{task}_1 & : \text{osacquire( cout )} \ll "abc" \ll "def" \ll \text{endl} ; \\
\text{task}_2 & : \text{osacquire( cout )} \ll "uvw" \ll "xyz" \ll \text{endl} ;
\end{align*}
\]

constraining the output to two different lines in either order:

\[
\begin{array}{c|c|c}
\text{abc def} & \text{uvw xyz} & \text{uvw xyz} \\
\text{uvw xyz} & \text{abc def} & \text{abc def}
\end{array}
\]
• Multiple I/O statements can be protected using block structure:

```cpp
{   // acquire the lock for stream cout for block duration
    osacquire acq( cout ); // named stream lock
    cout << "abc";
    osacquire( cout ) << "uvw " << "xyz " << endl; // OK?
    cout << "def";
} // implicitly release the lock when “acq” is deallocated
```

• Which locking-release pattern is used by stream locks?

6.3.2 Synchronization Lock

• **Synchronization lock** is used solely to block tasks waiting for synchronization.

• Weakest form of blocking lock as its only state is list of blocked tasks.
  
  o ⇒ *acquiring task always blocks* (no state to make it conditional)
    Need ability to yield time-slice and block versus yield and go back on ready queue.

  o ⇒ *release is lost when no waiting task* (no state to remember it)

• Often called a **condition lock**, with wait / signal(notify) for acquire / release.

6.3.2.1 Implementation

• Location of mutual exclusion classifies synchronization lock:

  • **external locking** use an external lock to protect task list,

  • **internal locking** use an internal lock to protect state (lock is extra state).

• external locking

  ```cpp
class SyncLock {
    Task * list;
  public:
    SyncLock() : list( nullptr ) {}
    void acquire() {
      // add self to task list
      yieldNoSchedule();
    }
    void release() {
      if ( list != nullptr ) {
        // remove task from blocked list and make ready
      }
    }
};
```

  o Use external state to avoid lost release.

  o Need mutual exclusion to protect task list and possible external state.
- Releasing task detects a blocked task and performs necessary cooperation.

- Usage pattern:
  - Cannot enter a restaurant if all tables are full.

```csharp
// shared variables
MutexLock m; // external mutex lock
SyncLock s; // synchronization lock
bool occupied = false; // indicate if event has occurred

// acquiring task
m.acquire(); // mutual exclusion to examine state & possibly block
if (occupied) {
    s.acquire(); // long-term block for event
    m.acquire(); // require mutual exclusion to set state
}
occupied = true; // set
m.release();
...

// releasing task
m.acquire(); // mutual exclusion to examine state
occupied = false; // reset
s.release(); // possibly unblock waiting task
m.release(); // release mutual exclusion

- Blocking occurs holding external mutual-exclusion lock!

  // acquiring task
  m.acquire(); // mutual exclusion to examine state & possibly block
  if (occupied) {
      m.release(); // release external mutex-lock
      // PREEMPTION
      s.acquire(); // block for event
      ...
```
• Race between blocking and unblocking tasks.

```cpp
void acquire( MutexLock & m ) {
    // add self to task list
    yieldNoSchedule( m );
    // possibly reacquire mutex lock
}
```

• Or, protecting mutex-lock is bound at synchronization-lock creation and used implicitly.

• Now use first usage pattern.

```cpp
// acquiring task
m.acquire();          // mutual exclusion to examine state & possibly block
if ( occupied ) {    // event not occurred ?
    s.acquire( m );  // block for event and release mutex lock
    ...
```

• Has the race been prevented?

• Problem: barging can occur when releasing task resets occupied.

• Use barging avoidance:

```cpp
// releasing task
m.acquire();        // mutual exclusion to examine state
if ( ! s.empty() ) s.release(); // unblock, no reset
else occupied = false; // reset
m.release();        // release mutual exclusion
```

or prevention:

```cpp
// releasing task
m.acquire();        // mutual exclusion to examine state
if ( ! s.empty() ) s.release(); // unblock, no reset
else { occupied = false; m.release(); } // reset & release
```

• Note, same problems as inside mutex lock but occurring outside between mutex and synchronization locks.

• internal locking

```cpp
class SyncLock {
    Task * list;        // blocked tasks
    SpinLock lock;      // internal lock

public:
    SyncLock() : list( nullptr ) {} 
    void acquire( MutexLock & m ) { // optional external lock
        lock.acquire();          // add self to task list
        m.release();            // release external mutex-lock
        CAN BE INTERRUPTED HERE
        yieldNoSchedule( lock );
        m.acquire();            // possibly reacquire after blocking
    }
```
void release() {
    lock.acquire();
    if ( list != nullptr ) {
        // remove task from blocked list and make ready
    }
    lock.release();
};

- Why does acquire still take an external lock?
- Why is the race after releasing the external mutex-lock not a problem?

- Has the busy wait been removed from the blocking lock?

6.3.2.2 uCondLock

- μC++ provides an internal synchronization-lock, uCondLock.

class uCondLock {
    public:
        uCondLock();
        void wait( uOwnerLock & lock );
        bool signal();
        bool broadcast();
        bool empty();
};

6.3.2.3 Programming Pattern

- Using synchronization locks is complex because they are weak.

- Must provide external mutual-exclusion and protect against loss signal (release).

- Why is synchronization more complex for blocking locks than spinning (uLock)?
bool done = false;

_Task T1 {
  uOwnerLock & mlk;
  uCondLock & clk;

  void main() {
    mlk.acquire(); // prevent lost signal
    if (! done) // signal occurred ?
      // signal not occurred
      clk.wait( mlk ); // atomic wait/release
      // mutex lock re-acquired after wait
    mlk.release(); // release either way
    S2;
  }

  public:
  T1( uOwnerLock & mlk,
       uCondLock & clk ) :
    mlk(mlk), clk(clk) {}
}

int main() {
  uOwnerLock mlk;
  uCondLock clk;
  T1 t1( mlk, clk );
  T2 t2( mlk, clk );
}

_Task T2 {
  uOwnerLock & mlk;
  uCondLock & clk;

  void main() {
    S1;
    mlk.acquire(); // prevent lost signal
    done = true; // remember signal occurred
    clk.signal(); // signal lost if not waiting
    mlk.release();
  }

  public:
  T2( uOwnerLock & mlk,
       uCondLock & clk ) :
    mlk(mlk), clk(clk) {}
};

6.3.3 Barrier

- A barrier coordinates a group of tasks performing a concurrent operation surrounded by sequential operations.

- Unlike previous synchronization locks, a barrier retains state about the events it manages: number of tasks blocked on the barrier.

- Since manipulation of this state requires mutual exclusion, most barriers use internal locking.

- E.g., 3 tasks must execute a section of code in a particular order: S1, S2 and S3 must all execute before S5, S6 and S7.
T1::main() {
    ...
    S1
    b.block();
    S5
    ...
}  
T2::main() {
    ...
    S2
    b.block();
    S6
    ...
}  
T3::main() {
    ...
    S3
    b.block();
    S7
    ...
}  

int main() {
    Barrier b(3);
    T1 x(b);
    T2 y(b);
    T3 z(b);
}

- Two common uses for barriers:

```
Barrier start(N+1), end(N+1); // shared Coordinator
// start N tasks so they can initialize
// general initialization
start.block(); // wait for threads to start
// do other work
end.block(); // wait for threads to end
// general close down and possibly loop

Workers
// initialize
start.block(); // wait for threads to start
// do work
end.block(); // wait for threads to end
// close down
```

- Two barriers allow Coordinator to accumulate results (subtotals) while Workers reinitialize (read next row).

- Alternative is last Worker does coordination, but prevents Workers reinitializing during coordination.

- Why not use termination synchronization and create new tasks for each computation?
6.3. BLOCKING LOCKS

6.3.3.1 uBarrier

- **#include <uBarrier.h>**

  ```cpp
  _Cormonitor uBarrier { // think _Coroutine
     protected:
       void main() { for (; ; ) suspend(); } // points of synchronization
       virtual void last() { resume(); } // called by last task to barrier
     public:
       uBarrier( unsigned int total );
       unsigned int total() const; // # of tasks synchronizing
       unsigned int waiters() const; // # of waiting tasks
       void reset( unsigned int total ); // reset # tasks synchronizing
       virtual void block(); // wait for Nth thread, which calls last, unblocks waiting thread
  }
  ```

- User barrier is built by:
  - inheriting from uBarrier
  - redefining last and/or block member and possibly coroutine main
  - possibly initializing main from constructor

- E.g., previous matrix sum (see page 69) adds subtotals in order of task termination, but barrier can add subtotals in order produced.

  ```cpp
  _Cormonitor Accumulator : public uBarrier {
     int total_ = 0, temp;
     uBaseTask * Nth_ = nullptr;
     protected:
       void last() { // reset and remember Nth task
         temp = total_; total_ = 0;
         Nth_ = &uThisTask();
       }
     public:
       Accumulator( int rows ) : uBarrier( rows ) {}
       void block( int subtotal ) {
         total_ += subtotal;
         uBarrier::block();
       }
       int total() { return temp; }
       uBaseTask * Nth() { return Nth_; }
  }
  ```
_Task Adder {
    int * row, size;
    Accumulator & acc;
    void main() {
        int subtotal = 0;
        for ( unsigned int r = 0; r < size; r += 1 ) subtotal += row[r];
        acc.block( subtotal ); // provide subtotal; block for completion
    }

public:
    Adder( int row[], int size, Accumulator & acc ) :
        size( size ), row( row ), acc( acc ) {}  
};

int main() {
    enum { rows = 10, cols = 10 };    
    int matrix[rows][cols];
    Adder * adders[rows];
    Accumulator acc( rows ); // barrier synchronizes each summation
    // read matrix
    for ( unsigned int r = 0; r < rows; r += 1 )
        adders[r] = new Adder( matrix[r], cols, acc );
    for ( unsigned int r = 0; r < rows; r += 1 )
        delete adders[r];
    cout << acc.total() << " " << acc.Nth() << endl;
}

• Why not have task delete itself after unblocking from uBarrier::block()?
    
    void block( int subtotal ) {
        total_ += subtotal; uBarrier::block();
        delete &uThisTask();
    }

• Coroutine barrier can be reused many times, e.g., read in a new matrix in Accumulator::main after each summation.

6.3.4 Binary Semaphore

• Binary semaphore (Edsger W. Dijkstra) is blocking equivalent to yielding spin-lock.
    
    Semaphore lock(0); // 0 => closed, 1 => open, default 1

• More powerful than synchronization lock as it remembers state about an event.

• acquire is P
    
    o passeren ⇒ to pass
    o prologen ⇒ (proberen) to try (verlagen) to decrease
    
    lock.P(); // wait to enter

P waits if the semaphore counter is zero and then decrements it.
6.3. BLOCKING LOCKS

• release is V
  
  o vrijgeven ⇒ to release
  o verhogen ⇒ to increase

  lock.V(); // release lock

  V increases the counter and unblocks a waiting task (if present).

• synchronization

  _Task T1 {
    BinSem & lk;
    void main() {
      ...
      S1
      lk.V();
      ...
    }
  }

  public:
  T1( BinSem & lk ) : lk(lk) {}

  int main() {
    BinSem lock( 0 ); // closed
    T1 t1( lock );
    T2 t2( lock );
  }

• mutual exclusion

  _Task T {
    BinSem & lk;
    void main() {
      ...
      lk.P();
      // critical section
      lk.V();
      ...
      lk.P();
      // critical section
      lk.V();
      ...
    }
  }

  public:
  T( BinSem & lk ) : lk(lk) {}

  int main() {
    BinSem lock( 1 ); // start open
    T t0( lock ), t1( lock );
  }
### 6.3.4.1 Implementation

- **class** BinSem {
  
  queue<T ask> blocked;  // blocked tasks
  bool avail;  // resource available?
  SpinLock lock;  // mutex nonblocking lock

- **public:**
  
  BinSem( bool start = true ) : avail( start ) {}

- **void** P() {
  
  lock.acquire();  // prevention barging
  if ( ! avail ) {
    // add self to lock’s blocked list
    yieldNoSchedule( lock );
    // DO NOT REACQUIRE LOCK
  }
  avail = false;
  lock.release();
}

- **void** V() {
  
  lock.acquire();
  if ( ! blocked.empty() ) {
    // remove task from blocked list and make ready
    // DO NOT RELEASE LOCK
  } else {
    avail = true;  // conditional reset
    lock.release();  // NO RACE
  }
}

- Same as single-acquisition mutexLock but can initialize avail.

- Higher cost for synchronization if external lock already acquired.

### 6.3.5 Counting Semaphore

- Augment the definition of P and V to allow a multi-valued semaphore.

- What does it mean for a lock to have more than open/closed (unlocked/locked)?

- Augment V to allow increasing the counter an arbitrary amount.

- synchronization
### 6.3. BLOCKING LOCKS

- **mutual exclusion**

- **Task**

  ```
  int main() {
      CntSem lk( 0 ); // closed
      T1 x( lk );
      T2 y( lk );
      T3 z( lk );
  }
  ```

### 6.3.5.1 Implementation

- Change availability into counter, and set to some maximum on creation.

```
class CntSem {
    queue<Task> blocked;  // blocked tasks
    int cnt;              // resource being used
    SpinLock lock;        // nonblocking lock

    public:
        CntSem( int start = 1 ) : cnt( start ) {}
        void P() {
            lock.acquire();
            cnt -= 1;
            if ( cnt < 0 ) {
                // add self to lock` s blocked list
                yieldNoSchedule( lock );
                // DO NOT REACQUIRE LOCK
            }
            lock.release();
        }
    }
```
void V() {
    lock.acquire();
    cnt += 1;
    if ( cnt <= 0 ) {
        // remove task from blocked list and make ready
        // DO NOT RELEASE LOCK
    } else {
        lock.release(); // NO RACE
    }
}

• In general, binary/counting semaphores are used in two distinct ways:
  1. For synchronization, if the semaphore starts at 0 ⇔ waiting for an event to occur.
  2. For mutual exclusion, if the semaphore starts at 1(N) ⇔ controls a critical section.

• µC++ provides a counting semaphore, uSemaphore, which subsumes a binary semaphore.

```cpp
#include <uSemaphore.h>
class uSemaphore {
    public:
        uSemaphore( unsigned int count = 1 );
        void P();
        bool TryP();
        void V( unsigned int times = 1 );
        int counter() const;
        bool empty() const;
};
```

6.4 Lock Programming

6.4.1 Precedence Graph

• P and V in conjunction with COBEGIN are as powerful as START and WAIT.

• E.g., execute statements so the result is the same as serial execution but concurrency is maximized.

  S1: a := 1
  S2: b := 2
  S3: c := a + b
  S4: d := 2 * a
  S5: e := c + d

• Analyse which data and code depend on each other.

• Display dependencies graphically in a precedence graph (different from process graph).
Semaphore L1(0), L2(0), L3(0), L4(0);
COBEGIN
BEGIN a := 1; V(L1); END;
BEGIN b := 2; V(L2); END;
BEGIN P(L1); P(L2); c := a + b; V(L3); END;
BEGIN P(L1); d := 2 * a; V(L4); END;
BEGIN P(L3); P(L4); e := c + d; END;
COEND

- Does this solution work?

- Optimal solution: minimum threads, M, and traverse M paths through precedence graph.
6.4.2.1 Unbounded Buffer

- Two tasks communicate through a queue of unbounded length.

![Diagram of unbounded buffer communication]

- Because tasks work at different speeds, producer may get ahead of consumer.

- Queue is shared between producer/consumer, and counting semaphore controls access.

```c
#define QueueSize ∞
int front = 0, back = 0;
int Elements[QueueSize];
uSemaphore full(0);
void Producer::main() {
  for (;;) {
    // produce an item
    // add to back of queue
    full.V();
  }
  // produce a stopping value
  full.V();
}
void Consumer::main() {
  for (;;) {
    full.P();
    // take an item from the front of the queue
    if ( stopping value ? ) break;
    // process or consume the item
  }
}
```

- Is there a problem adding and removing items from the shared queue?

- Is the full semaphore used for mutual exclusion or synchronization?

6.4.2.2 Bounded Buffer

- Two tasks communicate through a queue of bounded length.

- Use counting semaphores to account for the finite length of the shared queue.
6.4. LOCK PROGRAMMING

uSemaphore full(0), empty(QueueSize);

```cpp
void Producer::main() {
    for ( ;; ) {
        // produce an item
        empty.P();
        // add element to buffer
        full.V();
    }
    // produce a stopping value
    full.V();
}

void Consumer::main() {
    for ( ;; ) {
        full.P();
        // remove element from buffer
        if ( stopping value ? ) break;
        // process or consume the item
        empty.V();
    }
}
```

- Does this produce maximum concurrency?
- Can it handle multiple producers/consumers?

<p>| | | | | |</p>
<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>34</td>
<td>13</td>
<td>9</td>
<td>10</td>
<td>-3</td>
</tr>
</tbody>
</table>

full  | empty  
--- | ---
∅ | 5
X | A
Z | X
|  |
A | X
5 | 0

6.4.3 Lock Techniques

- Many possible solutions; need systematic approach.
- A **split binary semaphore** is a collection of semaphores where at most one of the collection has the value 1.
  - I.e., the sum of the semaphores is always less than or equal to one.
  - Used when different kinds of tasks have to block separately.
  - Cannot differentiate tasks blocked on the same semaphore (condition) lock. Why?
○ E.g., A and B tasks block on different semaphores so they can be unblocked based on kind, but collectively manage 2 semaphores like it was one.

- Split binary semaphores can be used to solve complicated mutual-exclusion problems by a technique called **baton passing**.

- The rules of baton passing are:
  ○ there is exactly one (conceptual) baton
  ○ nobody moves in the entry/exit code unless they have it
  ○ once the baton is released, cannot read/write variables in entry/exit

```
class BinSem {
    queue<Task> blocked;
    bool avail;
    SpinLock lock;
public:
    BinSem( bool start = true ) : avail( start ) {}
    void P() {
        lock.acquire(); PICKUP BATON, CAN ACCESS STATE
        if ( ! avail ) {
            // add self to lock' s blocked list
            PUT DOWN BATON, CANNOT ACCESS STATE
            yieldNoSchedule( lock );
            // UNBLOCK WITH SPIN LOCK ACQUIRED
            PASSED BATON, CAN ACCESS STATE
        }
        avail = false;
        lock.release(); PUT DOWN BATON, CANNOT ACCESS STATE
    }
    void V() {
        lock.acquire(); PICKUP BATON, CAN ACCESS STATE
        if ( ! blocked.empty() ) {
            // remove task from blocked list and make ready
            PASS BATON, CANNOT ACCESS STATE
        } else {
            avail = true;
            lock.release(); PUT DOWN BATON, CANNOT ACCESS STATE
        }
    }
};
```

- Can mutex/condition lock perform baton passing to prevent barging?
  ○ Not if signalled task must implicitly re-acquire the mutex lock before continuing.
  ○ ⇒ signaller must release the mutex lock.
  ○ There is now a race between signalled and calling tasks, resulting in barging.
6.4.4 Readers and Writer Problem

- Multiple tasks sharing a resource: some reading the resource and some writing the resource.
- Allow multiple concurrent reader tasks simultaneous access, but serialize access for writer tasks (a writer may read).
- Use split-binary semaphore to segregate 3 kinds of tasks: arrivers, readers, writers.
- Use baton-passing to help understand complexity.

6.4.4.1 Solution 1

```c
uSemaphore entry(1), rwait(0), wwait(0); // split binary semaphores
int rdel = 0, wdel = 0, rcnt = 0, wcnt = 0; // auxiliary counters

void Reader::main() {  // pickup baton
    entry.P();
    if ( wcnt > 0 ) {    // occupied ?
        rdel += 1; entry.V();  // put baton down
        rwait.P(); rdel -= 1;  // passed baton
    }  // READ
    rcnt += 1;
    if ( rdel > 0 ) {    // waiting readers ?
        rwait.V();
    } else {           // put baton down
        entry.V();
    }
}
```
void Writer::main() {
    entry.P(); // pickup baton
    if ( rcnt > 0 || wcnt > 0 ) { // occupied ?
        wdel += 1; entry.V(); // put baton down
        wwait.P(); wdel -= 1; // passed baton
    }
    wcnt += 1;
    entry.V(); // put baton down
    // WRITE
    entry.P(); // pickup baton
    wcnt -= 1;
    if ( rdel > 0 ) { // waiting readers ?
        rwait.V(); // pass baton
    } else if ( wdel > 0 ) { // waiting writers ?
        wwait.V(); // pass baton
    } else {
        entry.V(); // put baton down
    }
}

• Problem: reader only checks for writer in resource, never writers waiting to use it.

6.4.4.2 Solution 2

• Give writers priority and make the readers wait.

• Change entry protocol for reader to the following:

    entry.P(); // pickup baton
    if ( wcnt > 0 || wdel > 0 ) { // waiting writers?
        rdel += 1; entry.V(); // put baton down
        rwait.P(); rdel -= 1; // passed baton
    }
    rcnt += 1;
    if ( rdel > 0 ) {
        rwait.V(); // pass baton
    } else {
        entry.V(); // put baton down
    }

• Also, change writer’s exit protocol to favour writers:

    entry.P(); // pickup baton
    wcnt -= 1;
    if ( wdel > 0 ) { // check writers first
        wwait.V(); // pass baton
    } else if ( rdel > 0 ) {
        rwait.V(); // pass baton
    } else {
        entry.V(); // put baton down
    }
6.4. LOCK PROGRAMMING

6.4.4.3 Solution 3

- Fairness on simultaneous arrival is solved by alternation (Dekker’s solution).
- Flag is unnecessary if readers wait when there is a waiting writer, and all readers started after a writer.
- ⇒ put writer’s exit-protocol back to favour readers.

```c
entry.P(); // pickup baton
wcnt -= 1;
if ( rdel > 0 ) { // check readers first
    rwait.V(); // pass baton
} else if ( wdel > 0 ) { // pass baton
    wwait.V();
} else {
    entry.V(); // put baton down
}
```

- Arriving readers cannot barge ahead of waiting writers and unblocking writers cannot barge ahead of a waiting reader
- ⇒ alternation for simultaneous waiting.

6.4.4.4 Solution 4

- Problem: temporal barging!
- Staleness/freshness for last flag and staleness with no-flag.

![Diagram](image)

- Alternation for simultaneous waiting means when writer leaves resource:
  - both readers enter ⇒ 2:00 reader reads data that is stale; should read 1:30 write
  - writer enters and overwrites 12:30 data (never seen) ⇒ 1:00 reader reads data that is too fresh (i.e., missed reading 12:30 data)
- Staleness/freshness can lead to plane or stock-market crash.
- Have readers and writers wait on same semaphore ⇒ collapse split binary semaphore.
- **But now lose kind of waiting task!**
CHAPTER 6. LOCKS

uSemaphore entry(1), rwwait(0);          // readers/writers, temporal order  
int rwdel = 0, rcnt = 0, wcnt = 0;         // auxiliary counters  
enum RW { READER, WRITER };               // kinds of tasks  
queue<RW> rw_id;                         // queue of kinds  
void Reader::main() {  
  entry.P();                               // pickup baton  
  if ( wcnt > 0 || rwdel > 0 ) {           // anybody waiting?  
    rw_id.push( READER );                 // store kind  
    rwdel += 1; entry.V(); rwwait.P(); rwdel -= 1;  
    rw_id.pop();                         // more readers?  
  } else  
    rcnt += 1;                             // put baton down  
  // READ  
  entry.P();                               // exit protocol  
  rcnt -= 1;                               // last reader?  
  if ( rcnt <= 0 && rwdel > 0 ) {           // pass baton  
    rwwait.V();                           // put baton down  
  } else  
    entry.V();                             // pass baton  
}
void Writer::main() {
    entry.P(); // pickup baton
    if ( rcnt > 0 || wcnt > 0 ) {
        rw_id.push( WRITER ); // store kind
        rwdel += 1; entry.V(); rwwait.P(); rwdel -= 1;
        rw_id.pop();
    }
    wcnt += 1;
    entry.V(); // put baton down
    // WRITE
    entry.P(); // pickup baton
    wcnt -= 1;
    if ( rwdel > 0 ) { // anyone waiting ?
        rwwait.V(); // pass baton
    } else {
        entry.V(); // put baton down
    }
}

• Why can task pop front node on shadow queue when unblocked?

6.4.4.5 Solution 5

• Cheat on cooperation:

  o allow 2 checks for write instead of 1

  o use reader/writer bench and writer chair.
CHAPTER 6. LOCKS

uSemaphore entry(1), rwwait(0), wwait(0);
int rwdel = 0, wdel, rcnt = 0, wcnt = 0; // auxiliary counters
void Reader::main() {
    entry.P(); // pickup baton
    if ( ( wcnt > 0 || wdel > 0 || rwdel > 0 ) {
        rwdel += 1; entry.V(); rwwait.P(); rwdel -= 1;
    }
    rcnt += 1;
    if ( rwdel > 0 ) { // more readers ?
        rwwait.V(); // pass baton
    } else {
        entry.V(); // put baton down
    }
    // READ
    entry.P(); // pickup baton
    rcnt -= 1;
    if ( rcnt == 0 ) {
        if ( wdel != 0 ) { // writer waiting ?
            wwait.V(); // pass baton
        } else if ( rwdel > 0 ) { // anyone waiting ?
            rwwait.V(); // pass baton
        } else {
            entry.V(); // put baton down
        }
    } else {
        entry.V(); // put baton down
    }
}

void Writer::main() {
    entry.P(); // pickup baton
    if ( ( rcnt > 0 || wcnt > 0 ) { // first wait ?
        rwdel += 1; entry.V(); rwwait.P(); rwdel -= 1;
        if ( rcnt > 0 ) { // second wait ?
            wdel += 1; entry.V(); wwait.P(); wdel -= 1;
        }
    }
    wcnt += 1;
    entry.V(); // put baton down
    // WRITE
    entry.P(); // pickup baton
    wcnt -= 1;
    if ( rwdel > 0 ) { // anyone waiting ?
        rwwait.V(); // pass baton
    } else {
        entry.V(); // put baton down
    }
}

6.4.4.6 Solution 6

- Still temporal problem when tasks move from one blocking list to another.

- In solutions, reader/writer entry-protocols have code sequence:

  ... entry.V(); INTERRUPTED HERE Xwait.P();
• For writer:
  ◦ pick up baton and see readers using resource
  ◦ put baton down, entry.V(), but time-sliced before wait, Xwait.P().
  ◦ another writer does same thing, and this can occur to any depth.
  ◦ writers restart in any order or immediately have another time-slice
  ◦ e.g., 2:00 writer goes ahead of 1:00 writer ⇒ freshness problem.

• For reader:
  ◦ pick up baton and see writer using resource
  ◦ put baton down, entry.V(), but time-sliced before wait, Xwait.P().
  ◦ writers that arrived ahead of reader do same thing
  ◦ reader restarts before any writers
  ◦ e.g., 2:00 reader goes ahead of 1:00 writer ⇒ staleness problem.

• Need atomic block and release ⇒ magic like turning off time-slicing.
  \[X\text{wait.P(entry)}; \quad \text{// uC++ semaphore}\]

• Alternative: ticket
  ◦ readers/writers take ticket (see Section 5.18.9, p. 82) before putting baton down
  ◦ to pass baton, serving counter is incremented and then \text{WAKE ALL BLOCKED TASKS}
  ◦ each task checks ticket with serving value, and one proceeds while others reblock

• Alternative: private semaphore
  ◦ list of \text{private semaphores}, one for each waiting task, versus multiple waiting tasks on a semaphore.

\[
\begin{array}{c}
\text{Private Semaphores} \\
W\rightarrow W\rightarrow W\rightarrow W\rightarrow W\rightarrow W \\
\text{Readers & Writers} \\
W\rightarrow W\rightarrow W\rightarrow W\rightarrow W\rightarrow W \\
\text{baton} \\
W\rightarrow W\rightarrow W\rightarrow W \\
\text{Entry}
\end{array}
\]
uSemaphore entry(1);
int rcnt = 0, wcnt = 0;

struct RWNode {
    RW rw; // kinds of task
    uSemaphore sem; // private semaphore
    RWNode( RW rw ) : rw(rw), sem(0) {} 
};

queue<RWNode *> rw_id;

void Reader::main() {
    entry.P(); // pickup baton
    if ( wcnt > 0 || !rw_id.empty() ) { // anybody waiting?
        RWNode r( READER );
        rw_id.push( &r ); // store kind
        rwdel += 1; entry.V(); r.sem.P(); rwdel --;
        rw_id.pop();
    }
    rcnt += 1;
    if ( rwdel > 0 && rw_id.front() -> rw == READER ) { // more readers ?
        rw_id.front() -> sem.V(); // pass baton
    } else {
        entry.V(); // put baton down
    }
    // READ
    entry.P(); // pickup baton
    rcnt --;
    if ( rcnt == 0 && rwdel > 0 ) { // last reader ?
        rw_id.front() -> sem.V(); // pass baton
    } else {
        entry.V(); // put baton down
    }
}

void Writer::main() {
    entry.P(); // pickup baton
    if ( rcnt > 0 || wcnt > 0 ) { // resource in use ?
        RWNode w( WRITER );
        rw_id.push( &w ); // remember kind of task
        rwdel += 1; entry.V(); w.sem.P(); rwdel --;
        rw_id.pop();
    }
    wcnt += 1;
    entry.V();
    // WRITE
    entry.P(); // pickup baton
    wcnt --;
    if ( rwdel > 0 ) { // anyone waiting ?
        rw_id.front() -> sem.V(); // pass baton
    } else {
        entry.V(); // put baton down
    }
}

6.4.4.7 Solution 7

- Ad hoc solution with questionable split-binary semaphores and baton-passing.
• Tasks wait in temporal order on entry semaphore.

• Only one writer ever waits on the writer chair until readers leave resource.

• **Waiting writer blocks holding baton to force other arriving tasks to wait on entry.**

• Semaphore lock is used only for mutual exclusion.

• Sometimes acquire two locks to prevent tasks entering and leaving.

• Release in opposite order.

```cpp
uSemaphore entry(1); // two locks open
uSemaphore lock(1), wwait(0);
int rcnt = 0, wdel = 0;

void Reader::main() {
    entry.P(); // entry protocol
    lock.P();
    rcnt += 1;
    lock.V();
    entry.V(); // put baton down
    // READ
    lock.P(); // exit protocol
    rcnt -= 1;
    if ( ( rcnt == 0 && wdel == 1 ) ) { // last reader & writer waiting ?
        lock.V();
        wwait.V(); // pass baton
    } else
        lock.V();
}  
```
void Writer::main() {
    entry.P(); // entry protocol
    lock.P();
    if (rcnt > 0) {
        wdel += 1;
        lock.V();
        wwait.P(); // wait for readers
        wdel -= 1; // unblock with baton
    } else
        lock.V();
        // WRITE
    entry.V(); // exit protocol
}

• Is temporal order preserved?
• While solution is smaller, harder to reason about correctness.
• Does not generalize for other kinds of complex synchronization and mutual exclusion.
7 Concurrent Errors

7.1 Race Condition

- A race condition occurs when there is missing:
  - synchronization
  - mutual exclusion

- Two or more tasks race along assuming synchronization or mutual exclusion has occurred.
  - Race condition buried in four million lines of C code.
  - “in excess of three million online operational hours in which nothing had ever exercised that bug.”

7.2 No Progress

7.2.1 Live-lock

- Indefinite postponement: “You go first” problem on simultaneous arrival (consuming CPU)

- Caused by poor scheduling in entry protocol:

  - There always exists some mechanism to break tie on simultaneous arrival that deals effectively with live-lock (Oracle with cardboard test).

7.2.2 Starvation

- A selection algorithm ignores one or more tasks so they are never executed, i.e., lack of long-term fairness.
Like live-lock, starving task might be ready at any time, switching among active, ready and possibly blocked states (consuming CPU).

7.2.3 Deadlock

- **Deadlock** is the state when one or more processes are waiting for an event that will not occur.

- Unlike live-lock/starvation, deadlocked task is blocked so not consuming CPU.

7.2.3.1 Synchronization Deadlock

- Failure in cooperation, so a blocked task is never unblocked (stuck waiting):

  ```c
  int main() {
    uSemaphore s(0); // closed
    s.P();           // wait for lock to open
  }
  ```

7.2.3.2 Mutual Exclusion Deadlock

- Failure to acquire a resource protected by mutual exclusion.
7.3. DEADLOCK PREVENTION

- Deadlock, unless one of the cars is willing to backup.
- There are 5 conditions that must occur for a set of processes to deadlock.

  1. A **concrete** shared-resource requiring mutual exclusion, i.e., exists without a task.
     - A task “wanting to drive across the intersection” is not a resource.
  2. A process holds a resource while waiting for access to a resource held by another process (hold and wait).
  3. Once a process has gained access to a resource, the runtime system cannot get it back (no preemption).
  4. There exists a circular wait of processes on resources.
  5. These conditions must occur simultaneously.

- Simple example using semaphores:

```c
uSemaphore L1(1), L2(1); // open

task1
L1.P() L2.P() // acquire opposite locks
R1 R2  // access resource
L2.P() L1.P() // acquire opposite locks
R1 & R2 R2 & R1 // access resources
```

7.3 Deadlock Prevention

7.3.1 Synchronization Prevention

- Eliminate all synchronization from a program

7.3.2 Mutual Exclusion Prevention

  1. no mutual exclusion
  2. no hold & wait: do not give any resource, unless all resources can be given
  3. allow preemption
  4. no circular wait:

```
- Use an **ordered resource** policy:
```
divide all resources into classes $R_1$, $R_2$, $R_3$, etc.

- rule: can only request a resource from class $R_i$ if holding no resources from any class $R_j$ for $j \geq i$
- unless each class contains only one resource, requires requesting several resources simultaneously
- denote the highest class number for which $T$ holds a resource by $h(T)$
- if process $T_1$ is requesting a resource of class $k$ and is blocked because that resource is held by process $T_2$, then $h(T_1) < k \leq h(T_2)$
- as the preceding inequality is strict, a circular wait is impossible

5. prevent simultaneous occurrence:

### 7.4 Deadlock Avoidance

- Monitor all lock blocking and resource allocation to detect any potential formation of deadlock.

<table>
<thead>
<tr>
<th>deadlock</th>
<th>unsafe</th>
</tr>
</thead>
<tbody>
<tr>
<td>safe</td>
<td></td>
</tr>
</tbody>
</table>

#### 7.4.1 Banker’s Algorithm

- Demonstrate a safe sequence of resource allocations that $\Rightarrow$ no deadlock.

<table>
<thead>
<tr>
<th></th>
<th>R1</th>
<th>R2</th>
<th>R3</th>
<th>R4</th>
<th>total resources (TR)</th>
</tr>
</thead>
<tbody>
<tr>
<td>T1</td>
<td>6</td>
<td>12</td>
<td>4</td>
<td>2</td>
<td></td>
</tr>
<tr>
<td>T2</td>
<td>4</td>
<td>10</td>
<td>1</td>
<td>1</td>
<td>maximum needed for execution</td>
</tr>
<tr>
<td>T3</td>
<td>2</td>
<td>4</td>
<td>1</td>
<td>2</td>
<td>(M)</td>
</tr>
<tr>
<td>T1</td>
<td>5</td>
<td>9</td>
<td>0</td>
<td>1</td>
<td>currently allocated</td>
</tr>
<tr>
<td>T2</td>
<td>23</td>
<td>5</td>
<td>1</td>
<td>0</td>
<td>(C)</td>
</tr>
<tr>
<td>T3</td>
<td>1</td>
<td>2</td>
<td>0</td>
<td>0</td>
<td></td>
</tr>
</tbody>
</table>

resource request (T1, R1) $2 \rightarrow 3$

<table>
<thead>
<tr>
<th></th>
<th>T1</th>
<th>T2</th>
<th>T3</th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>R1</td>
<td>1</td>
<td>1</td>
<td>4</td>
<td>1</td>
</tr>
<tr>
<td>R2</td>
<td>5</td>
<td>2</td>
<td>7</td>
<td>0</td>
</tr>
<tr>
<td>R3</td>
<td>0</td>
<td>2</td>
<td>0</td>
<td>1</td>
</tr>
</tbody>
</table>

needed to execute

$(N = M - C)$
7.4. DEADLOCK AVOIDANCE

• Is there a safe order of execution that avoids deadlock should each process require its maximum resource allocation?

<table>
<thead>
<tr>
<th>current available resources</th>
<th>T2</th>
<th>T1</th>
<th>T3</th>
</tr>
</thead>
<tbody>
<tr>
<td>1 3 2 2 (CR = TR - ( \sum C_{cols} ))</td>
<td>0 1 2 0 (CR = CR - N_{T2})</td>
<td>1 0 3 1 (CR = CR - N_{T1})</td>
<td>1 3 4 1 (CR = CR - N_{T3})</td>
</tr>
<tr>
<td>2 5 3 2 (CR = CR + M_{T2})</td>
<td>2 5 3 2 (CR = CR + M_{T1})</td>
<td>6 12 4 2 (CR = CR + M_{T3})</td>
<td></td>
</tr>
</tbody>
</table>

• So a safe order exists (the left column in the table above) and hence the Banker’s Algorithm allows the resource request.

• Does task scheduling need to be adjusted to the safe sequence?

• The check for a safe order can be performed for every allocation of resource to a process (optimizations are possible, i.e., same thread asks for another resource).

7.4.2 Allocation Graphs

• One method to check for potential deadlock is to graph processes and resource usage at each moment a resource is allocated.

- resource
- with multiple instances
- task
- task is waiting for a resource instance
- task is holding a resource instance

• Multiple instances are put into a resource so that a specific resource does not have to be requested. Instead, a generic request is made.

- If any resource has several instances, a cycle \( \not\Rightarrow \) deadlock.
T1 → R1 → T2 → R3 → T3 → R2 → T1 (cycle)
T2 → R3 → T3 → R2 → T2 (cycle)

- Create isomorphic graph without multiple instances (expensive and difficult):

- If each resource has one instance, a cycle ⇒ deadlock.

- Use graph reduction to locate deadlocks:

7.5 Detection and Recovery

- Instead of avoiding deadlock let it happen and recover.

- Discovering deadlock is difficult, e.g., build and check for cycles in allocation graph.
  - not on each resource allocation, but every T seconds or every time a resource cannot be immediately allocated
7.6. WHICH METHOD TO CHOOSE?

- Try μC++ debugging macros to locate deadlock.

- Recovery involves preemption of one or more processes in a cycle.

7.6 Which Method To Chose?

- Maybe “none of the above”: just ignore the problem
  - if some process is blocked for rather a long time, assume it is deadlocked and abort it
  - do this automatically in transaction-processing systems, manually elsewhere

- Of the techniques studied, only the ordered resource policy turns out to have much practical value.
8 Indirect Communication

- P and V are low level primitives for protecting critical sections and establishing synchronization between tasks.
- Shared variables provide the actual information that is communicated.
- Split-binary semaphores and baton passing are complex.

8.1 Critical Regions

- Declare which variables are to be shared, as in:

  ```
  VAR v : SHARED INTEGER
  MutexLock v_lock;
  
  REGION v DO
    v_lock.acquire()
    // critical section
    ... // x = v; (read)  v = y (write)
  END REGION
  v_lock.release()
  ```

- Simultaneous reads are impossible!
- Modify to allow reading of shared variables outside the critical region and modifications in the region.
- Problem: reading partially updated information while a task is updating the shared variable in the region.
- Nesting can result in deadlock.

```
VAR x, y : SHARED INTEGER

task1
REGION x DO
... REGION y DO
  END REGION

task2
REGION y DO
... REGION x DO
  END REGION
```

8.2 Conditional Critical Regions

- REGION v DO
  ```
  AWAIT conditional-expression
  ... END REGION
  ```

- E.g. The consumer from the producer-consumer problem.
VAR Q : SHARED QUEUE<INT,10>

REGION Q DO
    AWA IT NOT EMPTY( Q ) buffer not empty
take an item from the front of the queue
END REGION

• If the condition is false, the region lock is released and entry is started again (busy waiting).

• If prevent busy waiting, block on queue for shared variable, and on region exit, search for
ttrue conditional-expression and unblock.

8.3 Monitor

• A **monitor** is an abstract data type that combines shared data with serialization of its modifi-
cation.

    _Monitor name {
        shared data
        members that see and modify the data
    };

• A **mutex member** (short for mutual-exclusion member) is one that does NOT begin execu-
tion if there is another active mutex member.

• **class** Mon {
    MutexLock mlock;
    int v;
    public:
    int x(...) {
        // mutex member
        mlock.acquire();
        . . . // int temp = v;
        mlock.release();
        return v; // return temp;
    }
};

• Recursive entry is allowed (owner mutex lock), i.e., one mutex member can call another or
itself.

• Unhandled exceptions raised within a monitor should always release the implicit monitor
locks so the monitor can continue to function.

• Destructor must be **mutex**, so ending a block with a monitor or deleting a dynamically allo-
cated monitor, blocks if thread in monitor.

• Atomic counter using a monitor:
8.4 Scheduling (Synchronization)

- A monitor may want to schedule tasks in an order different from the order in which they arrive (bounded buffer, readers/write with staleness/freshness).

- There are two techniques: external and internal scheduling.
  
  - *external* is scheduling tasks outside the monitor and is accomplished with the accept statement.
  
  - *internal* is scheduling tasks inside the monitor and is accomplished using condition variables with signal & wait.

8.4.1 External Scheduling

- E.g. Bounded Buffer

```c
_Monitor BoundedBuffer {
  int front = 0, back = 0, count = 0;
  int elements[20];
_public:
  _Nomutex int query() const { return count; }
  [_Mutex] void insert( int elem );
  [_Mutex] int remove();
};
void BoundedBuffer::insert( int elem ) {
  if ( count == 20 ) _Accept( remove );
  elements[back] = elem;
  back = ( back + 1 ) % 20;
  count += 1;
}
int BoundedBuffer::remove() {
  if ( count == 0 ) _Accept( insert );
  int elem = elements[front];
  front = ( front + 1 ) % 20;
  count -= 1;
  return elem;
}
```
• External scheduling is simple because unblocking (signalling) is implicit.

8.4.2 Internal Scheduling
• Scheduling among tasks inside the monitor.

• A **condition** is an external synchronization-lock (queue of waiting tasks):

  
  ```
  uCondition x, y, z[5];
  ```

• A task waits (blocks) by placing itself on a condition:

  ```
  x.wait();    // wait( mutex, condition )
  ```

• A task on a condition queue is made ready by signalling the condition:

  ```
  x.signal();
  ```

• E.g. Bounded Buffer (like binary semaphore solution):

  ```
  _Monitor BoundedBuffer {
    uCondition full, empty;
    int front = 0, back = 0, count = 0;
    int elements[20];
    public:
      _Nomutex int query() const { return count; }
      void insert( int elem ) {
        if ( count == 20 ) empty.wait();
        elements[back] = elem;
        back = ( back + 1 ) % 20;
        count += 1;
        full.signal();
      }
      int remove() {
        if ( count == 0 ) full.wait();
        int elem = elements[front];
        front = ( front + 1 ) % 20;
        count -= 1;
        empty.signal();
        return elem;
      }
  }
  ```

• General Model
8.4. SCHEDULING (SYNCHRONIZATION)

• When to use external or internal scheduling?

• However, external scheduling cannot be used if:
  
  ○ scheduling depends on member parameter value(s), e.g., compatibility code for dating
  
  ○ scheduling must block in the monitor but cannot guarantee the next call fulfills cooperation
CHAPTER 8. INDIRECT COMMUNICATION

_**Monitor** DatingService {
    enum { CCodes = 20 }; // compatibility codes
    uCondition girls[CCodes], boys[CCodes], exchange;
    int girlPhoneNo, boyPhoneNo;

    public:
    int girl( int phoneNo, int ccode ) {
        if ( boys[ccode].empty() ) { // no compatible boy ?
            girls[ccode].wait(); // wait for boy
            girlPhoneNo = phoneNo; // make phone number available
            exchange.signal(); // wake boy from chair
        } else {
            girlPhoneNo = phoneNo; // make phone number available
            boys[ccode].signal(); // wake boy
            exchange.wait(); // sit in chair
        }
        return boyPhoneNo;
    }

    int boy( int phoneNo, int ccode ) { // same as above, with boy/girl interchanged
    }
};

• Also, possible to use signal with empty bench (ccode) as chair.

8.5 Readers/Writer

• Solution 3 (Section 6.4.4.3, p. 115), no bargers, 5 rules, not temporal

_**Monitor** ReadersWriter {
    int rcnt = 0, wcnt = 0;
    uCondition readers, writers;

    public:
    void startRead() {
        if ( wcnt != 0 || ! writers.empty() ) readers.wait();
        rcnt += 1;
        readers.signal();
    }

    void endRead() {
        rcnt -= 1;
        if ( rcnt == 0 ) writers.signal();
    }
}
void startWrite() {
    if ( wcnt != 0 || rcnt != 0 ) writers.wait();
    wcnt = 1;
}

void endWrite() {
    wcnt = 0;
    if ( ! readers.empty() ) readers.signal();
    else writers.signal();
}

• Problem: has the same protocol as P and V.

ReadersWriter rw;
    
readers    writers
rw.startRead()    rw.startWrite() // 2-step protocol
// read    // write
rw.endRead()    rw.endWrite()

• Simplify protocol:

ReadersWriter rw;
    
readers    writers
rw.read(...)    rw.write(...) // 1-step protocol

• Alternative interface:

_Monitor ReadersWriter {
    _Mutex void startRead() { ... }
    _Mutex void endRead() { ... }
    _Mutex void startWrite() { ... }
    _Mutex void endWrite() { ... }
    public:
        _Nomutex void read(...) { // no const or mutable
            startRead();     // acquire mutual exclusion
            // read, no mutual exclusion
            endRead();       // release mutual exclusion
        }
        _Nomutex void write(...) { // no const or mutable
            startWrite()      // acquire mutual exclusion
            // write
            endWrite()        // release mutual exclusion
        }
    }

• Alternative interface, and remove wcnt (barging prevention):
CHAPTER 8. INDIRECT COMMUNICATION

Monitor ReadersWriter {
  _Mutex void startRead() {
    if (!writers.empty()) readers.wait();
    rcnt += 1;
    readers.signal();
  }
  _Mutex void endRead() { ... }
}

public:
  _Nomutex void read(...) { // no const or mutable
    startRead(); // acquire mutual exclusion
    // read, no mutual exclusion
    endRead(); // release mutual exclusion
  }
  void write(...) { // acquire mutual exclusion
    if (rcnt != 0 || !RWers.empty()) RWers.wait(READER);
    wcnt = 1;
    if (!RWers.empty() && RWers.front() == READER) RWers.signal();
  }
}

• Solution 4 (Section 6.4.4.4, p. 115), shadow queue

Monitor ReadersWriter {
  int rcnt = 0, wcnt = 0;
  uCondition RWers;
  enum RW { READER, WRITER };
  public:
    void startRead() {
      if (wcnt != 0 || !RWers.empty()) RWers.wait(READER);
      rcnt += 1;
      if (!RWers.empty() && RWers.front() == READER) RWers.signal();
    }
    void endRead() {
      rcnt -= 1;
      if (rcnt == 0) RWers.signal();
    }

    void startWrite() {
      if (wcnt != 0 || rcnt != 0) RWers.wait(WRITER);
      wcnt = 1;
    }
    void endWrite() {
      wcnt = 0;
      RWers.signal();
    }
};
- Use shadow queue to solve dating service, i.e., shadow with phone number.

- $\mu$C++ uCondLock and uSemaphore also support shadow queues.

- Solution 8, external scheduling

```cpp
_Monitor ReadersWriter {
    int rcnt = 0, wcnt = 0;
    public:
    void endRead() {
        rcnt -= 1;
    }
    void endWrite() {
        wcnt = 0;
    }
    void startRead() {
        if ( wcnt > 0 ) _Accept( endWrite );
        rcnt += 1;
    }
    void startWrite() {
        if ( wcnt > 0 ) _Accept( endWrite );
        else while ( rcnt > 0 ) _Accept( endRead );
        wcnt = 1;
    }
};
```

- Why has the order of the member routines changed?

- An exception raised in a monitor member propagates to the caller’s thread.

```cpp
_Monitor M {
    public:
    void mem1() {
        ... if ( ... ) _Throw E(); ... // E goes to caller
    } // uRendezvousFailure goes to “this”
    void mem2() {
        try {
            ... if ( ... ) _Accept( mem1 ); ... 
        } catch( uMutexFailure::RendezvousFailure & ) { // implicitly enabled
            // deal with rendezvous failure
        } // try
    }
};
```

- Caller in M::mem1 gets exception E propagated on its stack.
• On exiting M::mem1, caller implicitly raises non-local RendezvousFailure exception at monitor acceptor’s thread to identify failed cooperation.

• RendezvousFailure always enabled ⇒ _Enable block unnecessary.

• For multiple _Accept clauses

  _Accept( mem2, mem3, ... );

flag variable required to know which member failed.

• **Nested monitor problem**: acquire monitor (lock) M1, call to monitor M2, and wait on condition in M2.

![Diagram of nested monitors](image)

• Same problem occurs with locks.

• Releasing all locks can inadvertently release a lock, e.g., incorrectly release M0 before M1.

• Called **lock composition** problem.

### 8.6 Counting Semaphore, V, P vs. Condition, Signal, Wait

• There are several important differences between these mechanisms:

  •
  •
  •

• Possible to simulate P and V using a monitor:

```cpp
_Monitor semaphore {
    int sem;
    uCondition semcond;

    public:
    semaphore( int cnt = 1 ) : sem( cnt ) {}  
    void P() {
        if ( sem == 0 ) semcond.wait();
        sem -= 1;
    }
    void V() {
        sem += 1;
        semcond.signal();
    }
};
```

• Can this simulation be reduced?
8.7 Monitor Types

- **explicit scheduling** occurs when:
  - 
  - 

- **implicit scheduling** occurs when a task waits in or exits from a mutex member, and a new task is selected first from the A/S stack, then the entry queue.

<table>
<thead>
<tr>
<th>explicit scheduling</th>
<th>internal scheduling (signal)</th>
</tr>
</thead>
<tbody>
<tr>
<td>implicit scheduling</td>
<td>external scheduling (accept)</td>
</tr>
<tr>
<td></td>
<td>monitor selects (wait/exit)</td>
</tr>
</tbody>
</table>

- Monitors are classified by the implicit scheduling (who gets control) of the monitor when a task waits or signals or exits.

- Implicit scheduling can select from the calling (C), signalled (W), and signaller (S) queues.

![Diagram of monitor types and queues]
CHAPTER 8. INDIRECT COMMUNICATION

<table>
<thead>
<tr>
<th>relative priority</th>
<th>Condition</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>C &lt; W &lt; S</td>
<td>Useful, has Prevention</td>
</tr>
<tr>
<td>2</td>
<td>C &lt; S &lt; W</td>
<td>no barging</td>
</tr>
<tr>
<td>3</td>
<td>C = W &lt; S</td>
<td>Usable, needs Avoidance</td>
</tr>
<tr>
<td>4</td>
<td>C = S &lt; W</td>
<td>barging, prevent starvation</td>
</tr>
<tr>
<td>5</td>
<td>C = W = S</td>
<td>Rejected, Confusing</td>
</tr>
<tr>
<td>6</td>
<td>C &lt; W = S</td>
<td>arbitrary selection</td>
</tr>
<tr>
<td>7</td>
<td>S = W &lt; C</td>
<td>Rejected, Unsound</td>
</tr>
<tr>
<td>8</td>
<td>W &lt; S = C</td>
<td>uncontrolled barging, starvation</td>
</tr>
<tr>
<td>9</td>
<td>W &lt; C &lt; S</td>
<td></td>
</tr>
<tr>
<td>10</td>
<td>S &lt; W = C</td>
<td></td>
</tr>
<tr>
<td>11</td>
<td>S &lt; C &lt; W</td>
<td></td>
</tr>
<tr>
<td>12</td>
<td>W &lt; S &lt; C</td>
<td></td>
</tr>
<tr>
<td>13</td>
<td>S &lt; W &lt; C</td>
<td></td>
</tr>
</tbody>
</table>

- **Implicit Signal**
  - Monitors either have an explicit signal (statement) or an implicit signal (automatic signal).
  - The implicit signal causes a task to wait until the conditional expression is true.

```cpp
_Monitor BoundedBuffer {
    int front = 0, back = 0, count = 0;
    int elements[20];

    public:
    _Nomutex int query() const { return count; }
    void insert(int elem) {
        waitUntil count != 20; // not in uC++
        elements[back] = elem;
        back = ( back + 1 ) % 20;
        count += 1;
    }

    int remove() {
        waitUntil count != 0; // not in uC++
        int elem = elements[front];
        front = ( front + 1 ) % 20;
        count -= 1;
        return elem;
    }
};
```

- Additional restricted monitor-type requiring the signaller exit immediately from monitor (i.e., signal ⇒ return), called **immediate-return signal**.

- Remaining monitor types:
<table>
<thead>
<tr>
<th>signal type</th>
<th>priority</th>
<th>no priority</th>
</tr>
</thead>
<tbody>
<tr>
<td>Blocking</td>
<td>Priority Blocking (Hoare)</td>
<td>No Priority Blocking</td>
</tr>
<tr>
<td></td>
<td>$C &lt; S &lt; W$ ($\mu$C++ signalBlock)</td>
<td>$C = S &lt; W$</td>
</tr>
<tr>
<td>Nonblocking</td>
<td>Priority Nonblocking</td>
<td>No Priority Nonblocking</td>
</tr>
<tr>
<td></td>
<td>$C &lt; W &lt; S$ ($\mu$C++ signal)</td>
<td>$C = W &lt; S$ (Java/C#)</td>
</tr>
</tbody>
</table>

- no-priority blocking requires the **signaller task** to recheck the waiting condition in case of a barging task.
- no-priority non-blocking requires the **signalled task** to recheck the waiting condition in case of a barging task.
- implicit (automatic) signal is good for **prototyping** but have poor performance.
- priority-nonblocking has no barging and optimizes signal before return (supply cooperation).
- priority-blocking has no barging and handles internal cooperation within the monitor (wait for cooperation).

- **coroutine monitor** (**_Cormonitor_**)
  - coroutine with implicit mutual exclusion on calls to specified member routines:
    ```
    _Mutex _Coroutine  C {  // _Cormonitor
        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.

### 8.8 Java Monitor
- `public wait();`
- `public notify();`
- `public notifyAll();`
• Internal scheduling is no-priority nonblocking ⇒ barging

• Bounded buffer:

```java
class Buffer {
  // buffer declarations
  private int count = 0;
  public synchronized void insert( int elem ) {
    while ( count == Size ) wait(); // busy-waiting
    // add to buffer
    count += 1;
    if ( count == 1 ) notifyAll();
  }
  public synchronized int remove() {
    while ( count == 0 ) wait(); // busy-waiting
    // remove from buffer
    count -= 1;
    if ( count == Size - 1 ) notifyAll();
    return elem;
  }
}
```

• Erroneous Java implementation of barrier:

```java
class Barrier {
  // monitor
  private int N, count = 0;
  public Barrier( int N ) { this.N = N; }
  public synchronized void block() {
    count += 1;            // count each arriving task
    if ( count < N ) {
      try { wait(); } catch( InterruptedException e ) {} 
    } else {                // barrier full
      notifyAll();          // wake all barrier tasks
      count -= 1;           // uncount each leaving task
    }
  }
}
```

○ Nth task does notifyAll, leaves monitor and performs its i-th step, and then races back (barging) into the barrier before any notified task restarts.

○ It sees count still at N and incorrectly starts its i+1-th step before the current tasks have completed their i-th step.

• Fix by modifying code for Nth task to set count to 0 (barging avoidance).

```java
else {
  // barrier full
  count = 0;                // reset count
  notifyAll();              // wake all barrier tasks
}
```

• Technically, still wrong because of spurious wakeup ⇒ requires loop around wait.
if ( count < N )
    while ( ??? ) // cannot be count < N as count is always < N
        try { wait(); } catch( InterruptedException e ) {}}

• Requires more complex implementation.

```java
class Barrier {
    // monitor
    private int N, count = 0, generation = 0;
    public Barrier( int N ) { this.N = N; }
    public synchronized void block() {
        int mygen = generation;
        count += 1; // count each arriving task
        if ( count < N ) // barrier not full ? => wait
            while ( mygen == generation )
                try { wait(); } catch( InterruptedException e ) {} 
        else {
            // barrier full
            count = 0; // reset count
            generation += 1; // next group
            notifyAll(); // wake all barrier tasks
        }
    }
}
```

• Misconception of building condition variables in Java with nested monitors:

```java
class Condition {
    // try to build condition variable
    public synchronized void Wait() {
        try { wait(); } catch( InterruptedException ex ) {};
    }
    public synchronized void Notify() { notify(); }
}
```

```java
class BoundedBuffer {
    // buffer declarations
    private Condition full = new Condition(), empty = new Condition();
    public synchronized void insert( int elem ) {
        while ( count == NoOfElems ) empty.Wait(); // block producer
        // add to buffer
        count += 1;
        full.Notify(); // unblock consumer
    }
    public synchronized int remove() {
        while ( count == 0 ) full.Wait(); // block consumer
        // remove from buffer
        count -= 1;
        empty.Notify(); // unblock producer
        return elem;
    }
}
```
9 Direct Communication

- Monitors work well for passive objects that require mutual exclusion because of sharing.
- However, communication among tasks with a monitor is indirect.
- Problem: point-to-point with reply indirect communication:

\[
\text{Task}_1 \rightarrow \text{monitor} \rightarrow \text{Task}_2
\]

- Copy data
- Copy result
- Process data

- Point-to-point with reply direct communication:

\[
\text{Task}_1 \rightarrow \text{rendezvous} \rightarrow \text{Task}_2
\]

- Copy data
- Copy result
- Process data

9.1 Task

- A task is like a coroutine
- A task is unique
- A task is like a monitor
  - without external scheduling, tasks must call out to communicate \( \Rightarrow \) third party, or somehow emulate external scheduling with internal.
- In general, basic execution properties produce different abstractions:
• When thread or stack is missing it comes from calling object.

9.2 Scheduling
• A task may want to schedule access to itself by other tasks in an order different from the order in which requests arrive.

9.2.1 External Scheduling
• _Task BoundedBuffer {
  int front, back, count;
  int Elements[20];
  public:
    BoundedBuffer() : front(0), back(0), count(0) {}
    _Nmutex int query() const { return count; }
    void insert( int elem ) {
      Elements[back] = elem;
      back = (back + 1) % 20;
      count += 1;
    }
    int remove() {
      int elem = Elements[front];
      front = (front + 1) % 20;
      count -= 1;
      return elem;
    }
  }
  protected:
  void main() {
    for ( ;; ) { // INFINITE LOOP!!!
      // _Accept( insert, remove );
      _When( count != 20 ) _Accept( insert ) { // after call
        or _When( count != 0 ) _Accept( remove ) { // after call
          // _Accept
        }
      } // _Accept
    }
  }
};
• Is there a potential starvation problem?
• Why are accept statements moved from member routines to the task main?
• Why is BoundedBuffer::main defined at the end of the task?
• Equivalence using if statements:
  
  ```
  if ( 0 < count && count < 20 ) _Accept( insert, remove ); // not full/empty
  else if ( count < 20 ) _Accept( insert ); // not full
  else /* if ( 0 < count ) */ _Accept( remove ); // not empty
  ```

• Generalize from 2 to 3 conditionals/members:
  
  ```
  if ( C1 && C2 && C3 ) _Accept( M1, M2, M3 );
  else if ( C1 && C2 ) _Accept( M1, M2 );
  else if ( C1 && C3 ) _Accept( M1, M3 );
  else if ( C2 && C3 ) _Accept( M2, M3 );
  else if ( C1 ) _Accept( M1 );
  else if ( C2 ) _Accept( M2 );
  else if ( C3 ) _Accept( M3 );
  ```

• \( 2^N - 1 \) if statements needed to simulate \( N \) accept clauses.

- If there is a terminating _Else clause and no _Accept can be executed immediately, the terminating _Else clause is executed.

  ```
  _Accept( ... ) {
  }
  } or _Accept( ... ) {
  } _Else { ... } // executed if no callers
  ```
• To achieve greater concurrency in the bounded buffer, change to:

```c++
void insert(int elem) {
    Elements[back] = elem;
}

int remove() {
    return Elements[front];
}
```

```c++
protected:
void main() {
    for (;;) {
        _When ( count != 20 ) _Accept( insert ) {
            back = (back + 1) % 20;
            count += 1;
        } or _When ( count != 0 ) _Accept( remove ) {
            front = (front + 1) % 20;
            count -= 1;
        } // _Accept
    }
}
```

9.2.2 Internal Scheduling

• 

```c++
_Task BoundedBuffer {
    uCondition full, empty;
    int front, back, count;
    int Elements[20];

    public:
    BoundedBuffer() : front(0), back(0), count(0) {}
    _Nomutex int query() const { return count; }
    void insert(int elem) {
        if ( count == 20 ) empty.wait();
        Elements[back] = elem;
        back = (back + 1) % 20;
        count += 1;
        full.signal();
    }

    int remove() {
        if ( count == 0 ) full.wait();
        int elem = Elements[front];
        front = (front + 1) % 20;
        count -= 1;
        empty.signal();
        return elem;
    }
}
9.2. SCHEDULING

protected:
void main() {
    for ( ;; ) {
        _Accept( insert, remove );
        // do other work
    }
}
};

- Rendezvous is logically pending when wait restarts _Accept task, but post _Accept statement still executed (no RendezvousFailure).
  
  o Acceptor must eventually complete rendezvous for waiting caller.

- Try moving code to achieve greater concurrency.

void insert( int elem ) {
    if ( count == 20 ) empty.wait(); // only wait if necessary
    Elements[back] = elem;
}

int remove() {
    if ( count == 0 ) full.wait(); // only wait if necessary
    return Elements[front];
}

protected:
void postInsert() { // helper members
    back = ( back + 1 ) % size;
    count += 1;
}

void postRemove() {
    front = ( front + 1 ) % size;
    count -= 1;
}

void main() {
    for ( ;; ) {
        _Accept( insert ) {
            if ( count != 20 ) { // producer did not wait ?
                postInsert();
                if ( ! full.empty() ) { // waiting consumers ?
                    full.signal(); // wake and adjust
                    postRemove();
                }
            }
        }
    }
}
```c
} or _Accept( remove ) {
    if ( count != 0 ) {  // consumer did not wait?
        postRemove();
        if ( ! empty.empty() ) {  // waiting producers?
            empty.signal();  // wake and adjust
            postInsert();
        }
    }
    }  // _Accept
}  // for

• Must prevent starvation by producers (use _When or flip _Accept clauses).

• Must change signal to signalBlock.

9.2.3 Accepting the Destructor

• Common way to terminate a task is to have a stop member:
9.3 Increasing Concurrency

- 2 task involved in direct communication: client (caller) & server (callee)
- possible to increase concurrency on both the client and server side

9.3.1 Server Side

- Server manages a resource and server thread should introduce additional concurrency (assuming no return value).
No Concurrency

```cpp
_Task server1 {
    public:
        void mem1(...) { S1 }
        void mem2(...) { S2 }
        void main() {
            ... _Accept( mem1 );
            or _Accept( mem2 );
        }
    }
}
```

Some Concurrency

```cpp
_Task server2 {
    public:
        void mem1(...) { S1.copy-in }
        int mem2(...) { S2.copy-out }
        void main() {
            ... _Accept( mem1 ) { S1.work }
            or _Accept( mem2 ) { S2.work }
        }
    }
}
```

- Small overlap between client and server (client gets away earlier) increasing concurrency.

### 9.3.1.1 Internal Buffer

- The previous technique provides buffering of size 1 between the client and server.
- Use a larger internal buffer to allow clients to get in and out of the server faster?
- However, there are several issues:
  - Clients may need to wait for replies, in which case a buffer does not help unless there is an advantage to processing requests in non-FIFO order.
- Only way to free server’s thread to receive new requests and return finished results to clients is add another thread.
- Additional thread is a **worker task** that calls server to get work from buffer and return results to buffer.
- Note, customer (client), manager (server) and employee (worker) relationship.
- Number of workers has to balance with number of clients to maximize concurrency (bounded-buffer problem).

### 9.3.1.2 Administrator

- An **administrator** is a server managing multiple clients and worker tasks.
- The key is that an administrator does little or no “real” work; its job is to manage.
- An administrator is called by others, so an administrator is always accepting calls.
9.3. INCREASING CONCURRENCY

- An administrator usually maintains a list of work to pass to worker tasks.

- Typical workers are:

  - **timer** - prompt the administrator at specified time intervals
  - **notifier** - perform a potentially blocking wait for an external event (key press)
  - **simple worker** - do work given to them by and return the result to the administrator
  - **complex worker** - do work given to them by administrator and interact directly with client of the work
  - **courier** - perform a potentially blocking call on behalf of the administrator

9.3.2 Client Side

- While a server can attempt to make a client’s delay as short as possible, not all servers do it.

- In some cases, a client may not have to wait for the server to process a request (producer/consumer problem)
• This can be accomplished by an asynchronous call from the client to the server, where the caller does not wait for the call to complete.

• It is possible to build asynchronous facilities out of the synchronous ones and vice versa.

9.3.2.1 Returning Values

• If a client only drops off data to be processed by the server, the asynchronous call is simple.

• However, if a result is returned from the call, i.e., from the server to the client, the asynchronous call is significantly more complex.

• To achieve asynchrony in this case, a call must be divided into two calls:

  ```
  callee.start(arg); // provide arguments
  // callee performs other work asynchronously
  result = callee.wait(); // obtain result
  ```

• Not same as START/WAIT because server thread exists.

  ○ many-to-one versus one-to-one

• Time between calls allows calling task to execute asynchronously with task performing operation on the caller’s behalf.

• If result is not ready when second call is made

9.3.2.2 Tickets

• One form of protocol is the use of a token or ticket.

• The first part of the protocol transmits the arguments specifying the desired work and a ticket (like a laundry ticket) is returned immediately.

• The second call pulls the result by passing the ticket.

9.3.2.3 Call-Back Routine

• Another protocol is to transmit (register) a routine on the initial call.

• When the result is ready, the routine is called by the task generating the result, passing it the result.

• The call-back routine cannot block the server; it can only store the result and set an indicator (e.g., V a semaphore) known to the client.
9.3. INCREASING CONCURRENCY

9.3.2.4 Futures

- A future provides the same asynchrony as above but without an explicit protocol.

- The protocol becomes implicit between the future and the task generating the result.

- In detail, a future is an object that is a subtype of the result type expected by the caller.

  ```
  future = callee.work( arg ); // provide arguments, return future
  // perform other work asynchronously
  i = future + . . .; // obtain result, may block if not ready
  ```

- The general design for a future is:

  ```
  class Future : public ResultType {
    friend _Task server; // allow server to access internal state
    ResultType result;  // place result here
    uSemaphore avail;  // wait here if no result
    Future *link;      // intrusive data structure
  public:
    Future() : avail( 0 ) {}
    ResultType get() {
      avail.P(); // wait for result
      return result;
    }
  }
  ```

- Unfortunately, the syntax for retrieving the value of the future is awkward as it requires a call to the get routine.

- Also, in languages without garbage collection, the future must be explicitly deleted.

- \( \mu \text{C++} \) provides two forms of template futures, which differ in storage management (like Actors/Messages).
  - Explicit-Storage-Management future (Future_ESM<\( T \)) must be allocated and deallocated explicitly by the client.
  - Implicit-Storage-Management future (Future_ISM<\( T \)) automatically allocates and frees storage (when future no longer in use, GC).

- Basic set of operations for both types of futures, divided into client and server operations.

**Client**

- Future value:
```cpp
#include <uFuture.h>

Server server; // server thread handles async calls
Future_ISM<int> f[10];
for (int i = 0; i < 10; i += 1) {
    f[i] = server.perform(i); // asynchronous server call
}
// work asynchronously while server processes requests
for (int i = 0; i < 10; i += 1) {
    // retrieve async results
    int v = f[i]; // synchronize, read, and copy
    osacquire( cout ) << v << ' ' << f[i] + i << endl; // cheap read after synchronize
}
f[3] = 3; // DISALLOWED: OTHER THREADS READING VALUE
... 
f[3].reset(); // reset future ⇒ empty and can be reused (be careful)
...
f[3].cancel(); // attempt to stop server and clients from usage
```

- Why not combine: osacquire( cout ) << f[i]() << ' ' << f[i] + 1 << endl;

- Future pointer:

```cpp
#include <uFuture.h>

Server server; // server thread handles async calls
int val
Future_ISM<int ->> fval;
fval = server.perform(val); // async call to server (change val by reference)
// work asynchronously while server processes requests
osacquire( cout ) << *fval() << endl; // synchronize on retrieve value
val = 3; // ALLOWED: BUT FUTURE POINTER IS STILL READ-ONLY
```

- `available` – returns `true` if asynchronous call completed, otherwise `false`.

- `operator()` – (function call) returns `read-only` copy of future result.

- `operator T` – (conversion to type T) returns `read-only` copy of future result.

  *Only allowed after blocking access or call to `available` returns `true`.*

- `reset` – mark future as empty ⇒ current future value is unavailable ⇒ future can be reused.

- `cancel` – attempts to cancel the asynchronous call the future refers to.

- `cancelled` – returns `true` if the future is cancelled and `false` otherwise.

Server
_Task_ Server {
  struct Work {
    int i;  // argument(s)
    Future_ISM<int> result;  // result
    Work( int i ) : i( i ) {}  
  };
  Future_ISM<int> perform( int i ) {  // called by clients
    Work *w = new Work( i );  // create work request
    requests.push_back( w );  // add to list of requests
    return w->result;  // return future in request
  }
  // server or server's worker does
  Work *w = requests.front();  // take next work request
  requests.pop_front();  // remove request
  int r = ... w->i ...;  // compute result using argument w->i
  w->result.delivery( r );  // insert result into future
  delete w;  // CLIENT FUTURE NOT DELETED (REF COUNTING)
};

delivery( T result ) – copy result to be returned to the client(s) into the future, unblocking clients waiting for the result.

delivery( T result ) – copy result to be returned to the client(s) into the future, unblocking clients waiting for the result.

delivery( T result ) – copy result to be returned to the client(s) into the future, unblocking clients waiting for the result.

delivery( T result ) – copy result to be returned to the client(s) into the future, unblocking clients waiting for the result.

delivery( T result ) – copy result to be returned to the client(s) into the future, unblocking clients waiting for the result.

delivery( T result ) – copy result to be returned to the client(s) into the future, unblocking clients waiting for the result.

delivery( T result ) – copy result to be returned to the client(s) into the future, unblocking clients waiting for the result.

delivery( T result ) – copy result to be returned to the client(s) into the future, unblocking clients waiting for the result.

delivery( T result ) – copy result to be returned to the client(s) into the future, unblocking clients waiting for the result.

delivery( T result ) – copy result to be returned to the client(s) into the future, unblocking clients waiting for the result.

delivery( T result ) – copy result to be returned to the client(s) into the future, unblocking clients waiting for the result.

delivery( T result ) – copy result to be returned to the client(s) into the future, unblocking clients waiting for the result.

delivery( T result ) – copy result to be returned to the client(s) into the future, unblocking clients waiting for the result.

delivery( T result ) – copy result to be returned to the client(s) into the future, unblocking clients waiting for the result.

delivery( T result ) – copy result to be returned to the client(s) into the future, unblocking clients waiting for the result.

delivery( T result ) – copy result to be returned to the client(s) into the future, unblocking clients waiting for the result.

delivery( T result ) – copy result to be returned to the client(s) into the future, unblocking clients waiting for the result.

delivery( T result ) – copy result to be returned to the client(s) into the future, unblocking clients waiting for the result.

delivery( T result ) – copy result to be returned to the client(s) into the future, unblocking clients waiting for the result.

delivery( T result ) – copy result to be returned to the client(s) into the future, unblocking clients waiting for the result.

delivery( T result ) – copy result to be returned to the client(s) into the future, unblocking clients waiting for the result.

delivery( T result ) – copy result to be returned to the client(s) into the future, unblocking clients waiting for the result.

exception( uBaseEvent & cause ) – copy a server-generated exception into the future, and the exception cause is thrown at waiting clients.

  _Event_ E {};  
  Future_ISM<int> result;  
  result.exception( new E );  // deleted by future

exception deleted by reset or when future deleted

Complex Future Access

- **select statement** waits for one or more heterogeneous futures based on logical selection-criteria.

- Simplest select statement has a single _Select clause, e.g.:

  _Select( selector-expression );

- Selector-expression must be satisfied before execution continues.

- _Select( f1 );

- Multiple futures may appear in a compound selector-expression, related using logical operators || and &&:

  _Select( f1 || f2 && f3 );

- Normal operator precedence applies: _Select( ( f1 || ( f2 && f3 ) ) ) .
• For any selector expression containing an || operator, some futures in the expression may be unavailable after the selector expression is satisfied.

• or and and keywords relate the _Select clauses like operators || and && relate futures in a select-expression, including precedence.

  \[
  \texttt{Select} ( f1 | | f2 \&\& f3 ); \equiv \texttt{Select} ( f1 ) \\
  \text{or} \texttt{Select} ( f2 ) \\
  \text{and} \texttt{Select} ( f3 );
  \]

• Parentheses may be used to specify evaluation order.

  \[
  \texttt{Select} ( ( f1 | | ( f2 \&\& f3 ) ) ) \equiv ( \texttt{Select} ( f1 ) \\
  \text{or} ( \texttt{Select} ( f2 ) \\
  \text{and} \texttt{Select} ( f3 ) ) );
  \]

• A _Select clause may be guarded with a logical expression and have code executed after a future receives a value:

  \[
  \texttt{When} ( \text{conditional-expression} ) \texttt{Select} ( f1 ) \\
  \text{statement-1} // \text{action, future available}
  \text{or} \\
  \texttt{When} ( \text{conditional-expression} ) \texttt{Select} ( f2 ) \\
  \text{statement-2} // \text{action, future available}
  \text{and} \texttt{When} ( \text{conditional-expression} ) \texttt{Select} ( f3 ) \\
  \text{statement-3} // \text{action, future available}
  \]

• Each _Select-clause action is executed when its sub-selector expression is satisfied, i.e., when each future becomes available.

• However, control does not continue until the selector expression associated with the entire statement is satisfied.

• If a guard is false, execution continues without waiting for that future to become available (like future is available).

• Assume only f3 becomes available:

  \[
  \texttt{When} ( \text{true} ) \texttt{Select} ( f1 ) \{ \ldots \}
  \text{or} \texttt{When} ( \text{false} ) \texttt{Select} ( f2 ) \{ \ldots \}
  \text{and} \texttt{When} ( \text{true} ) \texttt{Select} ( f3 ) \{ \ldots \}
  \]

  execution continues.

• An action statement is triggered only once for its selector expression, even if the selector expression is compound.

  \[
  \texttt{Select} ( f1 ) \\
  \text{statement-1}
  \text{or} \texttt{Select} ( f2 \&\& f3 ) \\
  \text{statement-2} // \text{triggered once after both available}
  \]
• However, for ||:

```cpp
>Select( f1 || f2 )
statement-1 // triggered once after one available
and _Select( f3 )
statement-2
```

• In statement-1, only one future f1 or f2 caused the action to be triggered.

• A select statement can be non-blocking using a terminating _Else clause, e.g.:

```cpp
_Select( selector-expression )
statement // action
_When ( conditional-expression ) _Else // terminating clause
statement // action
```

```
Future_ISM<int> fi;
Future_ISM<double> fd;
struct Msg { int i, j; }; Future_ISM<Msg> fm;
struct Stop {}; Future_ISM<Stop> fs;
struct Cont {}; Future_ISM<Cont> fc;
_Task Worker {
    void main() {
        for ( ;; ) {
            _Select( fi ) { cout << fi() << endl; fi.reset(); }
            and _Select( fd ) { cout << fd() << endl; fd.reset(); }
            and _Select( fm ) { Msg m = fm();
                cout << m.i << " " << m.j << endl; fm.reset(); }
            or _Select( fs ) { cout << "stop" << endl; break; }
            fc.delivery( (Cont){} ); // synchronize
        }
    }
};
int main() {
    Worker worker;
    for ( int i = 0; i < 10; i += 1 ) {
        fi.delivery( i );
        fd.delivery( i + 2.5 );
        fm.delivery( (Msg){ i, 2 } );
        fc(); fc.reset(); // wait for 3 futures to be processed
    }
    fs.delivery( (Stop){} );
} // wait for worker to terminate
```
10 Optimization

• A computer with infinite memory and speed requires no optimizations to use less memory or run faster (space/time).

• With finite resources, optimization is useful/necessary to conserve resources and for good performance.

• Furthermore, most programs are not written in optimal order or in minimal form.

• General forms of optimizations are:
  - reordering: data and code are reordered to increase performance in certain contexts.
  - eliding: removal of unnecessary data, data accesses, and computation.
  - replication: processors, memory, data, code are duplicated because of limitations in processing and communication speed (speed of light).

• Kinds of optimizations are restricted by the kind of execution environment.

10.1 Sequential Optimizations

• Most programs are sequential; even concurrent programs are
  - (large) sections of sequential code per thread connected by
  - small sections of concurrent code where threads interact (protected by synchronization and mutual exclusion (SME))

• Sequential execution presents simple semantics for optimization.
  - operations occur in program order, i.e., sequentially

• Dependencies result in partial ordering among a set of statements (precedence graph):
  - data dependency (R ⇒ read, W ⇒ write)
    - \( R_x \rightarrow R_x \)
    - \( W_x \rightarrow R_x \)
    - \( R_x \rightarrow W_x \)
    - \( W_x \rightarrow W_x \)
    - \( y = x; \)
    - \( x = 0; \)
    - \( y = x; \)
    - \( x = 0; \)
    - \( z = x; \)
    - \( y = x; \)
    - \( x = 3; \)
    - \( x = 3; \)

  Which statements can be reordered?
  - control dependency
    - \( 1 \) if ( \( x == 0 \) )
    - \( 2 \) \( y = 1; \)

• To achieve better performance, compiler/hardware make changes:
  - reorder disjoint (independent) operations (variables have different addresses)
\[ R_x \rightarrow R_y \quad W_x \rightarrow R_y \quad R_x \rightarrow W_y \quad W_x \rightarrow W_y \]

\[
\begin{align*}
t &= x; & x &= 0; & x &= 1; & y &= 0; \\
s &= y; & y &= 1; & y &= 3; & x &= 3;
\end{align*}
\]

Which statements can be reordered?

2. elide unnecessary operations (transformation/dead code)

\[
x = 0; \quad // \text{unnecessary, immediate change}
\]

\[
x = 3;
\]

\[
\text{for ( int } i = 0; i < 10000; i += 1 ); \quad // \text{unnecessary, no loop body}
\]

\[
\text{int factorial( int } n, \text{ int } acc \} \quad // \text{tail recursion}
\]

\[
\text{if ( } n == 0 \text{) return } acc; \quad // \text{convert to loop}
\]

\[
\text{return factorial( } n - 1, n * acc \];
\]

3. execute in parallel if multiple functional-units (adders, floating units, pipelines, cache)

- Very complex reordering, reducing, and overlapping of operations allowed.
- Overlapping implies parallelism, \textit{but limited capability in sequential execution}.

## 10.2 Memory Hierarchy

- Complex memory hierarchy:

  \[
  \begin{array}{c}
  \text{CPU} \\
  \begin{array}{c}
  \text{registers} \\
  \text{replication} \\
  \text{memory}
  \end{array}
  \end{array}
  \quad \begin{array}{c}
  \text{CPU} \\
  \begin{array}{c}
  \text{registers} \\
  \text{replication} \\
  \text{cache} \\
  \text{memory}
  \end{array}
  \end{array}
  \]

- Optimizing data flow along this hierarchy defines a computer’s speed.
- Hardware aggressively optimizes data flow for sequential execution.

### 10.2.1 Cache Review

- Problem: CPU 100(0) times faster than memory (100,00(0) times faster than disk).
- Solution: copy data from general memory into very, very fast local-memory (registers).
- Problem: billions of bytes of memory but only 6–256 registers.
- Solution: move highly accessed data \textit{within} a program from memory to registers for as long as possible and then back to memory.
10.2. MEMORY HIERARCHY

- Problem: quickly run out of registers as more data accessed.
  - ⇒ must rotate data from memory through registers dynamically.
  - compiler attempts to keep highly used variables in registers (LRU, requires oracle)

- Problem: does not handle highly accessed data among programs (threads).
  - each context switch saves and restores most registers to memory
  - registers are private and cannot be shared

- Solution: use hardware cache (automatic registers) to stage data without pushing to memory and allow sharing of data among programs.

```
int x, y, z;
x += 1;  
```

- Caching transparently hides the latency of accessing main memory.
- Cache loads in 64/128/256 bytes, called cache line, with addresses multiple of line size.
- When x is loaded into register 1, a cache line containing x, y, and z are implicitly copied up the memory hierarchy from memory through caches.
- When cache is full, data evicted, i.e., remove old cache-lines to bring in new (LRU).
- When program ends, its addresses are flushed from the memory hierarchy.

- In theory, cache can eliminate registers, but registers provide small addressable area (register window) with short addresses (3-8 bits for 8-256 registers) ⇒ shorter instructions.
10.2.2 Cache Coherence

- Multi-level caches used, each larger but with diminishing speed (and cost).

- E.g., 64K L1 cache (32K Instruction, 32K Data) per core, 256K L2 cache per core, and 8MB L3 cache shared across cores.

- Data reads logically percolate variables from memory up the memory hierarchy, making cache copies, to registers.

- Why is it necessary to eagerly move reads up the memory hierarchy?

- Data writes from registers to variables logically percolate down the memory hierarchy through cache copies to memory.

- Why is it advantageous to lazily move writes down the memory hierarchy?

- If OS moves program to another processor, all caching information is invalid and the program’s data-hierarchy reforms.

- Unlike registers, all cache values are shared across the computer.

- Hence, variable can be replicated in a large number of locations.

- No cache coherence for shared variable x (madness)
10.2. MEMORY HIERARCHY

- With cache coherence for shared variable $x$

- **Cache coherence** is hardware protocol ensuring update of duplicate data.

- **Cache consistency** addresses *when* processor sees update $\Rightarrow$ bidirectional synchronization.

- **Prevent flickering and scrambling during simultaneous R/W or W/W.**

- Eager cache-consistency means data changes appear instantaneous by waiting for acknowledge from all cores (complex/expensive).

- Lazy cache-consistency allows reader to see own write before acknowledgement $\Rightarrow$ concurrent programs read stale data!
  
  - writes eventually appear in (largely) same over as written
critical section works as writes to shared variable appear before write to lock release
otherwise, spin (lock) until write appears

- If threads continually read/write same memory locations, they invalidate duplicate cache lines, resulting in excessive cache updates.
  - called cache thrashing
  - updated value bounces from one cache to the next

- Because cache line contains multiple variables, cache thrashing can occur inadvertently, called false sharing.

- Thread 1 read/writes x while Thread 2 read/writes y ⇒ no direct shared access, but indirect sharing as x and y share cache line.
  - Fix by separating x and y with sufficient storage (padding) to be in next cache line.
  - Difficult for dynamically allocated variables as memory allocator positions storage.

```
thread 1 thread 2
int *x = new int int *y = new int;
```
x and y may or may not be on same cache line.

### 10.3 Concurrent Optimizations

- In sequential execution, strong memory ordering: reading always returns last value written.

- In concurrent execution, weak memory ordering: reading can return previously written value or value written in future.
  - happens on multi-processor because of scheduling and buffering (see scrambling/flickering in Section 5.18.6, p. 78 and freshness/staleness in Section 6.4.4.4, p. 115).
  - notion of current value becomes blurred for shared variables unless everyone can see values assigned simultaneously.

- SME control order and speed of execution, otherwise non-determinism causes random results or failure (e.g., race condition, Section 7.1, p. 123).

- Sequential sections accessing private variables can be optimized normally but not across concurrent boundaries.

- Concurrent sections accessing shared variables can be corrupted by sequential optimizations ⇒ restrict optimizations to ensure correctness.

- For correctness and performance, identify concurrent code and only restrict its optimization.

- What/how to restrict depends on what sequential assumptions are implicitly applied by hardware and compiler (programming language).

- Following examples show how sequential optimizations cause failures in concurrent code.
10.3. CONCURRENT OPTIMIZATIONS

10.3.1 Disjoint Reordering

- \( R_x \rightarrow R_y \) allows \( R_y \rightarrow R_x \)

Reordering disjoint reads does not cause problems. Why?

- \( W_x \rightarrow R_y \) allows \( R_y \rightarrow W_x \)

  - In Dekker entry protocol (see Section 5.18.6, p. 78)
    
    | Cons | Prod |
    |------|------|
    | 1 \( me = \text{WantIn}; \ // W \) | 1 \( \text{temp} = \text{you}; \ // R \) |
    | 2 while (\( you == \text{WantIn} \) ) \{ \ // R | 2 while (\( \text{temp} == \text{WantIn} \) ) \{ |
    | 3 \ldots |

  both threads read \text{DontWantIn}, both set \text{WantIn}, both see \text{DontWantIn}, and proceed.

- \( R_x \rightarrow W_y \) allows \( W_y \rightarrow R_x \)

  - In synchronization flags (see Section 5.12, p. 72), allows interchanging lines 1 & 3 for Cons:
    
    \[
    \begin{array}{l}
    1 \text{while (} ! \text{Insert} \text{)} ; \ // R \\
    2 \text{Insert} = \text{false} ; \\
    3 \text{data} = \text{Data} ; \ // W \\
    \end{array}
    \quad \begin{array}{l}
    1 \text{while (} ! \text{Insert} \text{)} ; \ // R \\
    2 \text{Insert} = \text{false} ; \\
    3 \text{data} = \text{Data} ; \ // W \\
    \end{array}
    \]

  allows reading of uninserted data

- \( W_x \rightarrow W_y \) allows \( W_y \rightarrow W_x \)

  - In synchronization flags (see Section 5.12, p. 72), allows interchanging lines 1 & 2 in Prod and lines 3 & 4 in Cons:
    
    \[
    \begin{array}{l}
    1 \text{Data} = i ; \ // W \\
    2 \text{Insert} = \text{true} ; \ // W \\
    \end{array}
    \quad \begin{array}{l}
    1 \text{Data} = i ; \ // W \\
    2 \text{Insert} = \text{true} ; \ // W \\
    \end{array}
    \]

  allows reading of uninserted data

  - In Peterson’s entry protocol, allows interchanging lines 1 & 2 (see Section 5.18.7, p. 80):
    
    \[
    \begin{array}{l}
    1 \text{me} = \text{WantIn} ; \ // W \\
    2 ::\text{Last} = \&\text{me} ; \ // W \\
    \end{array}
    \quad \begin{array}{l}
    1 \text{me} = \text{WantIn} ; \ // W \\
    2 ::\text{Last} = \&\text{me} ; \ // W \\
    \end{array}
    \]

  allows race before either task sets its intent and both proceed

- Compiler uses all of these reorderings to break mutual exclusion:

  \[
  \begin{array}{l}
  \text{lock.acquire()} \quad \text{lock.acquire()} \quad \text{lock.acquire()} \\
  \quad \text{lock.acquire()} \quad \text{lock.release()} \quad \text{lock.release()} ; \quad \text{lock.release()} ; \ // \text{critical section}
  \end{array}
  \]

  - moves lock entry/exit after/before critical section because entry/exit variables not used in critical section.
E.g., **double-check locking** for singleton-pattern:

```cpp
int * ip = nullptr; // shared (volatile for correctness)
...
if ( ip == nullptr ) {
    lock.acquire(); // attempt to get storage (race)
    if ( ip == nullptr ) {
        ip = new int( 0 ); // still no storage ? (double check)
    }
    lock.release();
}
```

Why do the first check? Why do the second check?

- Fails if last two writes are reordered, \( W_{\text{malloc}} \) and \( W_{\text{ip}} \), disjoint variables:

```plaintext
call malloc // new storage address returned in r1
st #0,(r1) // initialize storage
st r1,ip // initialize pointer
```

see `ip` but uninitialized.

### 10.3.2 Eliding

- For high-level language, compiler decides when/which variables are loaded into registers and for how long.

  - Elide reads (loads) by copying (replicating) value into a register:

    ```plaintext
    Task_1
    \ldots
    flag = false // write
    Task_2
    register = flag; // one read, auxiliary variable
    while ( register ); // cannot see change by T1
    ```

    - Hence, variable logically disappears for duration in register.

    - \( \Rightarrow \) task spins forever in busy loop if R before W.

    - Also, elide meaningless sequential code:

      ```plaintext
      sleep( 1 ); // unnecessary in sequential program
      ```

      \( \Rightarrow \) task misses signal by not delaying

### 10.3.3 Replication

- Why is there a benefit to reorder R/W?

  - Modern processors increase performance by executing multiple instructions in parallel (data flow, precedence graph) on **replicated hardware**.

    - internal pool of instructions taken from program order
    - begin simultaneous execution of instructions with inputs
    - collect results from finished instructions
    - feed results back into instruction pool as inputs
10.4 Memory Model

• Manufacturers define set of optimizations performed implicitly by processor.

• Set of optimizations indirectly define a memory model.

<table>
<thead>
<tr>
<th>Relaxation Model</th>
<th>W → R</th>
<th>R → W</th>
<th>W → W</th>
<th>Lazy cache update</th>
</tr>
</thead>
<tbody>
<tr>
<td>atomic consistent (AT)</td>
<td></td>
<td></td>
<td>✓</td>
<td>✓</td>
</tr>
<tr>
<td>sequential consistency (SC)</td>
<td></td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
</tr>
<tr>
<td>total store order (TSO)</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
</tr>
<tr>
<td>partial store order (PSO)</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
</tr>
<tr>
<td>weak order (WO)</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
</tr>
<tr>
<td>release consistency (RC)</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
</tr>
</tbody>
</table>

• AT has events occur instantaneously ⇒ slow or impossible (distributed).

• SC accepts all events cannot occur instantaneously ⇒ may read old values

• SC still strong enough for software mutual-exclusion (Dekker 5.18.6 / Peterson 5.18.7).
  ○ SC often considered minimum model for concurrency (Java provides SC)

• No hardware supports just AT/SC.
  ○ TSO (x86/SPARC), PSO (ARM), WO (Alpha), RC (PowerPC)

10.5 Preventing Optimization Problems

• All optimization problems result from races on shared variables.

• If shared data is protected by locks (implicit or explicit),
  ○ locks define the sequential/concurrent boundaries,
  ○ boundaries must preclude optimizations that affect concurrency.

• Called race free as synchronization and mutual exclusion preclude races.

• However, race free does have races.

• Two approaches:
○ ad hoc: programmer manually augments all data races with pragmas to restrict compiler/hardware optimizations: not portable but often optimal.

○ formal: language has memory model and mechanisms to abstractly define races in program: portable but often baroque and suboptimal.

• data access / compiler (C/C++): volatile qualifier

  ○ Force variable loads and stores to/from registers (at sequence points)

  ○ created for longjmp or force access for memory-mapped devices

  ○ for architectures with few registers, practically all variables are implicitly volatile. Why?

    ○ Java volatile / C++11 atomic stronger ⇒ prevent eliding and disjoint reordering.

• program order / compiler (static): disable inlining, asm("": "memory");

• memory order / runtime (dynamic): sfence, lfence, mfence (x86)

  ○ guarantee previous stores and/or loads are completed, before continuing.

• atomic operations test-and-set, which often imply fencing

• cache is normally invisible and does not cause issues (except for DMA)

• mechanisms to fix issues are specific to compiler or platform

  ○ difficult, low-level, diverse semantics, not portable ⇒ tread carefully!

• Dekker for TSO:
10.5. PREVENTING OPTIMIZATION PROBLEMS

#define CALIGN __attribute__(( aligned (64) )) // cache-line alignment
#define Fence() __asm__ __volatile__ ( "mfence" ) // prevent hardware reordering
#define Pause() __asm__ __volatile__ ( "pause" : : ) // efficient busy wait

enum Intent { DontWantIn, WantIn } Last;
_Task Dekker {
  volatile Intent &me, &you, *&Last;
  void main() {
    for ( int i = 1; i <= 1000; i += 1 ) {
      for ( ;; ) { // entry protocol
        me = WantIn; // high priority
        Fence();
        if ( you == DontWantIn ) break;
        if ( Last == &me ) { // high priority
          me = DontWantIn;
          while ( Last == &me ) Pause(); // low priority
        }
        Pause();
      }
      CriticalSection(); // critical section
      Last = &me; // exit protocol
      me = DontWantIn;
    }
  }
}

public:
  Dekker( volatile Intent &me, volatile Intent &you, volatile Intent *&Last ) :
    me(me), you(you), Last(Last) {}
};

int main() {
  volatile Intent me CALIGN = DontWantIn, you CALIGN = DontWantIn,
  *Last CALIGN = rand() % 2 ? &me : &you;
  Dekker t0(me, you, Last), t1(you, me, Last);
};

• Locks built with these features ensure SC for protected shared variables.
  o no user races and strong locks ⇒ SC memory model
11 Other Approaches

11.1 Atomic (Lock-Free) Data-Structure

- **Lock free** data-structure have operations, which are critical sections, but performed without **ownership**.

- Lock-free is still locking (misnomer) ⇒ spin for conceptual lock ⇒ busy-waiting (starvation).

- If guarantees eventual progress, called **wait free**.

11.1.1 Compare and Assign Instruction

- The compare-and-assign(set) instruction performs an atomic compare and conditional assignment CAA (erroneously called compare-and-swap, CAS).

```c
int Lock = OPEN; // shared

bool CAA( int & val, int comp, int nval ) {
    // begin atomic
    if ( val == comp ) {
        val = nval;
        return true;
    }
    return false;
    // end atomic
}

void Task::main() { // each task does
    while ( ! CAA( Lock, OPEN, CLOSED ) );
    // critical section
    Lock = OPEN;
}
```

- Alternative implementation assigns comparison value with the value when not equal.

```c
bool CAV( int & val, int & comp, int nval ) {
    // begin atomic
    if (val == comp) {
        val = nval;
        return true;
    }
    comp = val; // return changed value
    return false;
    // end atomic
}
```

- Assignment when unequal useful to restart operations with new changed value.

11.1.2 Lock-Free Stack

- E.g., build a stack with lock-free push and pop operations.
class Stack {
    Node * top;       // pointer to stack top

public:
    struct Node {
        // data
        Node * next;       // pointer to next node
    };
    void push( Node & n );
    Node * pop();
};

• Use CAA to atomically update top pointer when nodes pushed or popped concurrently.

    void Stack::push( Node & n ) {
        for ( ;; ) {
            // busy wait
            n.next = top;     // link new node to top node
            if ( CAA( top, n.next, &n ) ) break;  // attempt to update top node
        }
    }

    Node * Stack::pop() {
        Node * t;
        for ( ;; ) {
            // busy wait
            t = top;        // copy current top
            if ( t == nullptr ) return t;  // empty list ?
            if ( CAA( top, t, t->next ) ) return t;  // attempt to update top node
        }
    }

    ○ Create new node, n, at 0x4ffb8 to be added.
    ○ Set n.next to top.
    ○ CAA tries to assign new top &n to top.
    ○ CAA fails if top changed since copied to n.next
    ○ If CAA failed, update n.next to top, and try again.
    ○ CAA succeeds when top == n.next, i.e., no push or pop between setting n.next and trying to assign &n to top.
    ○ CAV copies changed value to n.next, so eliminates resetting t = top in busy loop.
11.1. A TOMIC (LOCK-FREE) DATA-STRUCTURE

- Copy top node, 0x4ffb8, to t for removal.
- If not empty, attempt CAA to set new top to next node, t->next.
- CAA fails if top changed since copied to t.
- If CAA failed, update t to top, and try again.
- CAA succeeds when top == t->next, i.e., no push or pop between setting t and trying to assign t->next to top.
- CAV copies the changed value into t, so eliminates resetting t = top in busy loop.

- Note, load of t->next can access stolen node, and fail if storage freed and address-space shortened.

11.1.3 ABA problem

- Pathological failure for series of pops and pushes, called ABA problem.
- Given stack with 3 nodes:
  \[ \text{top} \rightarrow A \rightarrow B \rightarrow C \]

11.1.4 Hardware Fix

- Probabilistic solution for stack exists using double-wide CAVD instruction, which compares and assigns 64/128-bit values for 32/64-bit architectures.

```c
bool CAVD( uintS_t &val, uintS_t &comp, uintS_t nval ) {
  // begin atomic
  if ( val == comp ) { // 64/128-bit compare
    val = nval; // 64/128-bit assignment
    return true;
  }
  comp = val; // 64/128-bit assignment
  return false;
  // end atomic
}
```

- Now, associate counter (ticket) with header node:
class Stack {
    union Link {
        struct {
            Node * top; // pointer to stack top
            uintptr_t count; // count each push
        };
        uintS_t atom; // 64/128-bit integer
    } link;
    public:
    struct Node {
        // resource data
        Link next; // pointer to next node/count (resource)
    };
    Stack() { link.atom = 0; }
    void push( Node & n );
    Node * pop();
};

• Increment counter in push so pop can detect ABA if node re-pushed.

void Stack::push( Node & n ) {
    n.next = link; // atomic assignment unnecessary
    for ( ;; ) { // busy wait
        if ( CAVD( link.atom, n.next.atom,
            (Link){ &n, n.next.count + 1 }.atom ) ) break;
    }
}

○ CAVD used to copy entire header to n.next, as structure assignment (2 fields) is not atomic.
○ In busy loop, copy local idea of top to next of new node to be added.
○ CAVD tries to assign new top-header to (h).
○ If top has not changed since copied to n.next, update top to n (new top), and increment counter.
○ If top has changed, CAVD copies changed values to n.next, so try again.

Node * Stack::pop() {
    Link t = link; // atomic assignment unnecessary
    for ( ;; ) { // busy wait
        if ( t.top == nullptr ) return nullptr; // empty stack ?
        if ( CAVD( link.atom, t.atom,
            (Link){ t.top->next.top, t.count }.atom ) ) return t.top;
    }
}

○ CAVD used to copy entire header to t, as structure assignment (2 fields) is not atomic.
○ In busy loop, check if pop on empty stack and return nullptr.
○ If not empty, CAVD tries to assign new top t.top->next.top,t.count to h.
11.1. ATOMIC (LOCK-FREE) DATA-STRUCTURE

○ If top has not changed since copied to t, update top to t.top->next.top (new top).
○ If top has changed, CAVD copies changed values to t, so try again.

• ABA problem (mostly) fixed:
  top,3 → A → B → C

• Popping task, \( T_i \), has t set to A,3 and dereferenced B from t.top->next in argument of CAVD.

• \( T_i \) is time-sliced, and while blocked, nodes A and B are popped, and A is pushed again:
  top,4 → A → C  // adding A increments counter

• When \( T_i \) restarts, CAVD fails as header A,3 not equal top A,4.

• Only probabilistic correct as counter finite (like ticket counter).

• Finally, none of the programs using CAA ensure eventual progress; therefore, rule 5 is broken.

11.1.5 Hardware/Software Fix

• Fixing ABA with CAA/V and more code is extremely complex (100s of lines of code), as is implementing more complex data structures (queue, deque, hash).

• All solutions require complex determination of when a node has no references (like garbage collection).
  ○ each thread maintains a list of accessed nodes, called hazard pointers
  ○ thread updates its hazard pointers while other threads are reading them
  ○ thread removes a node by hiding it on a private list and periodically scans the hazard lists of other threads for references to that node
  ○ if no pointers are found, the node can be freed

• For lock-free stack: x, y, z are memory addresses
  ○ first thread puts x on its hazard list
  ○ second thread cannot reuse x, because of hazard list
  ○ second thread must create new object at different location
  ○ first thread detects change

• Summary: locks versus lock-free
  ○ lock-free has no ownership (hold-and-wait) \( \Rightarrow \) no deadlock
  ○ lock-free can only handle limited set of critical sections
    lock can protect arbitrarily complex critical section versus
  ○ lock-free no panacea, performance unclear
  ○ combine lock and lock-free?
11.2 Exotic Atomic Instruction

- VAX computer has instructions to atomically insert and remove a node to/from the head or tail of a circular doubly linked list.

```c
struct links {
    links *front, *back;
};

bool INSQUE( links &entry, links &pred ) {
    // atomic execution
    // insert entry following pred
    return entry.front == entry.back; // first node inserted ?
}

bool REMQUE( links &entry ) {
    // atomic execution
    // remove entry
    return entry.front == null; // last node removed ?
}
```

- MIPS processor has two instructions that generalize atomic read/write cycle: LL (load locked) and SC (store conditional).

  - LL instruction loads (reads) a value from memory into a register, and sets a hardware reservation on the memory from which the value is fetched.
  
  - Register value can be modified, even moved to another register.
  
  - SC instruction stores (writes) new value back to original or another memory location.
  
  - However, store is conditional and occurs only if no interrupt, exception, or write has occurred at LL reservation.
  
  - Failure indicated by setting the register containing the value to be stored to 0.
  
  - E.g., implement test-and-set with LL/SC:

    ```c
    int testSet( int &lock ) {
        // atomic execution
        int temp = lock; // read
        lock = 1; // write
        return temp; // return previous value
    }
    
    testSet: // register $4 contains pointer to lock
    ll $2,$(4) // read and lock location
    or $8,$2,1 // set register $8 to 1 (lock | 1)
    sc $8,$(4) // attempt to store 1 into lock
    beq $8,$0,testSet // retry if interference between read and write
    j $31 // return previous value in register $2
    ```

  - Does not suffer from ABA problem.
Node *pop( Header &h ) {
    Node *t, next;
    for ( ;; ) {
        t = LL( top );
        if ( t == nullptr ) break; // empty list ?
        next = t->next
        if ( SC( top, next ) ) break; // attempt to update top node
    }
    return t;
}

○ SC detects change to top instead of indirect check if value in top changes (CAA).
○ However, most architectures support weak LL/SC.
    * reservation granularity may be cache line or memory block rather than word
    * no nesting or interleaving of LL/SC pairs, and prohibit memory access between
      LL and SC.
○ Cannot implement atomic swap of 2 memory locations as two reservations are neces-
  sary (register to memory swap is possible).

• Hardware transactional memory allows 4, 6, 8 reservations, e.g., Advanced Synchronization
  Facility (ASF) proposal in AMD64.

• Like database transaction that optimistically executes change, and either commits changes,
  or rolls back and restarts if interference.
  ○ SPECULATE : start speculative region and clear zero flag ; next instruction checks for
    abort and branches to retry.
  ○ LOCK : MOV instructions indicates location for atomic access, but moves not visible to
    other CPUs.
  ○ COMMIT : end speculative region
    * if no conflict, make MOVs visible to other CPUs.
    * if conflict to any move locations, set failure, discard reservations and restore reg-
      isters back to instruction following SPECULATE

• Can implement several data structures without ABA problem.

• Software Transactional Memory (STM) allows any number of reservations.
  ○ atomic blocks of arbitrary size:

    void push( header & h, node & n ) {
        atomic {
            n.next = top;          // LOCK/MOV
            top = &n
            }                     // COMMIT
        }
    }
    ○ records all memory locations read and written, and all values mutated.
bookkeeping costs and rollbacks typically result in performance degradation
  
  alternative implementation inserts locks to protect shared access

finding all access is difficult and ordering lock acquisition is complex

11.3 General-Purpose GPU (GPGPU)

- Graphic Processing Unit (GPU) is a coprocessor to main computer, with separate memory and processors.

- GPU is a Single-Instruction Multiple-Data(Thread) (SIMD(T)) architecture versus Multiple-Instruction Multiple-Data (MIMD)

  ![SIMD and MIMD Diagram]

- In branching code

  ```java
  if ( a[i] % 2 == 0 ) {
      a[i] /= 2; // true threads
  } else {
      a[i] += 3; // false threads
  }
  ```

  - all threads test the condition (create mask of true and false)
  - true mask
    - true threads execute instructions
    - false threads execute NOP (no-operation)
  - negate mask
    - false threads execute instructions
    - true threads execute NOP

- In general, critical path is time to execute both clauses of if (no speedup).

- Complex contortions to eliminate different forms of branching.

- GPU structure
  - kernel manages multiple blocks (loaded/controlled by CPU)
  - block executes the same code
  - warp synchronizes execution (one instruction decoder per warp)
11.3. GENERAL-PURPOSE GPU (GPGPU)

- **thread** computes value

![Diagram](image)

- blocks may be barrier-synchronized
- synchronization among blocks ⇒ finishing kernel and launching new one

- Instead of cache to optimize latency in warp, large register file is used to optimize throughput.
  - GPUs have enough duplicate registers to store state of several warps.

- Kernel is memory-bound ⇒ data layout extremely important performance consideration.

```c
// kernel routine, handle contiguous matrix, different ID for each thread
kernel void GPUsum( float *matrix[], float subtotals[], int rows ) {
  # define sub(m, r, c) ((typeof(m[0][0]) *)m)[r * rows + c]
  subtotals[ID] = 0.0;
  for ( int r = 0; r < rows; r += 1 )
    subtotals[ID] += sub( matrix, r, ID );
}
```

- Add rows by columns.

```
threads
subtotals +
threads
warp
```

- Warps scheduled to run when their required data is loaded from memory.
- CPU sets up GPU memory, loads memory, launches code, retrieves results.
int main() {
    int rows, cols;
    cin >> rows >> cols;  // matrix size
    // optimal to use contiguous matrix
    float matrix[rows][cols], subtotals[rows], total = 0.0;
    // ... fill matrix
    float *matrix_d, *subtotals_d;  // matrix/subtotals buffer on GPU
    // allocate space on GPU
    GPUMalloc( &matrix_d, sizeof(matrix) );
    GPUMalloc( &subtotals_d, sizeof(subtotals) );
    // copy matrix to GPU
    GPUMemcpy( matrix_d, matrix, sizeof(matrix), GPUMemcpyHostToDevice );
    // compute matrix sum on GPU
    GPUsum<<< 1, cols >>>( matrix_d, subtotals_d, rows );
    // do asynchronous work!!!
    // copy subtotals from GPU, may block
    GPUMemcpy( subtotals, subtotals_d, sizeof(subtotals), GPUMemcpyDeviceToHost );
    for (int i = 0; i < cols; i += 1) total += subtotals[i];
    cout << total << endl;
}

• Most modern multi-core CPUs have similar model using vector-processing.
  ○ Simulate warps and use concurrency framework (μC++) to schedule blocks.

11.4 Concurrency Languages
11.4.1 Ada 95
• E.g., monitor bounded-buffer, restricted implicit (automatic) signal:

  protected type buffer is -- __Monitor
  entry insert( elem: in ElemType ) when count < Size is -- mutex member
    begin
      -- add to buffer
      count := count + 1;
    end insert;
  entry remove( elem: out ElemType ) when count > 0 is -- mutex member
    begin
      -- remove from buffer, return via parameter
      count := count - 1;
    end remove;
  private:
    ... // buffer declarations
    count : Integer := 0;
  end buffer;

• The when clause is only be used at start of entry routine not within.

• The when expression can contain only global-object variables; parameter or local variables are disallowed ⇒ no direct dating-service.
11.4. CONCURRENCY LANGUAGES

• Eliminate restrictions and dating service is solvable.

```java
_Monitor DatingService {
    AUTOMATIC_SIGNAL;
    int girls[noOfCodes], boys[noOfCodes]; // count girls/boys waiting
    bool exchange; // performing phone–number exchange
    int girlPhoneNo, boyPhoneNo; // communication variables

    public:
    int girl(int phoneNo, int ccode) {
        girls[ccode] += 1;
        if (boys[ccode] == 0) { // no boy waiting?
            WAITUNTIL(boys[ccode] != 0, , ); // use parameter, not at start
            boys[ccode] -= 1; // decrement dating pair
            girls[ccode] -= 1;
            girlPhoneNo = phoneNo; // girl's phone number for exchange
            exchange = false; // wake boy
        } else {
            girlPhoneNo = phoneNo; // girl's phone number before exchange
            exchange = true; // start exchange
            WAITUNTIL(!exchange, , ); // wait until exchange complete, not at start
        }
        EXIT();
        return boyPhoneNo;
    }
}
```

• E.g., task bounded-buffer:

```java
task type buffer is -- _Task
    ... -- _buffer declarations
    count : integer := 0;
begin -- _thread starts here (task main)
    loop
        select -- _Accept
            when count < Size => -- guard
            accept insert(elem : in ElemType) do -- _mutex member
                -- add to buffer
                count := count + 1;
            end;
            -- executed if this accept called
        or
            when count > 0 => -- guard
            accept remove(elem : out ElemType) do -- _mutex member
                -- remove from buffer, return via parameter
                count := count - 1;
            end;
        end select;
    end loop;
end buffer;
var b : buffer -- create a task
```

• `select` is external scheduling and only appears in `task` main.
• Hence, Ada has no direct internal-scheduling mechanism, i.e., no condition variables.

• Instead a requeue statement can be used to make a blocking call to another (usually non-public) mutex member of the object.

• The original call is re-blocked on that mutex member’s entry queue, which can be subsequently accepted when it is appropriate to restart it.

• However, all requeue techniques suffer the problem of dealing with accumulated temporary results:
  o If a call must be postponed, its temporary results must be returned and bundled with the initial parameters before forwarding to the mutex member handling the next step,
  o or the temporary results must be re-computed at the next step (if possible).

• In contrast, waiting on a condition variable automatically saves the execution location and any partially computed state.

11.4.2 SR/Concurrent C++

• SR and Concurrent C++ have tasks with external scheduling using an accept statement.

• But no condition variables or requeue statement.

• To ameliorate lack of internal scheduling add a when and by clause on the accept statement.

• when clause is allowed to reference caller’s arguments via parameters of mutex member:

  ```
  select
    accept mem( code : in Integer )
      when code % 2 = 0 do . . .  -- accept call with even code
  or
    accept mem( code : in Integer )
      when code % 2 = 1 do . . .  -- accept call with odd code
  end select;
  ```

• when placed after the accept clause so parameter names are defined.

• when referencing parameter ⇒ implicit search of waiting tasks on mutex queue ⇒ locking mutex queue.

• Select longest waiting if multiple true when clauses.

• by clause is calculated for each true when clause and the minimum by clause is selected.

  ```
  select
    accept mem( code : in Integer )
      when code % 2 = 0 by -code do . . .  -- accept largest even code
  or
    accept mem( code : in Integer )
      when code % 2 = 1 by code do . . .  -- accept smallest odd code
  end select;
  ```
11.4. CONCURRENCY LANGUAGES

- Select longest waiting if multiple by clauses with same minimum.

- **by** clause exacerbates the execution cost of computing accept clause.

- While **when/by** removes some internal scheduling and/or requeues, constructing expressions can be complex.

- Still situations that cannot be handled, e.g., if selection criteria involves multiple parameters:
  - select lowest even value of code1 and highest odd value of code2 if there are multiple lowest even values.
  - selection criteria involves information from other mutex queues such as the dating service (girl must search the boy mutex queue).

- Often simplest to unconditionally accept a call allowing arbitrarily examination, and possibly postpone (internal scheduling).

### 11.4.3 Java

- Java’s concurrency constructs are largely derived from Modula-3.

```java
class Thread implements Runnable {
    public Thread();
    public Thread(String name);
    public String getName();
    public void setName(String name);
    public void run(); // uC++ main
    public synchronized void start();
    public static Thread currentThread();
    public static void yield();
    public final void join();
}
```

- Thread is like \(\mu\)C++ uBaseTask, and all tasks must explicitly inherit from it:

```java
class MyTask extends Thread { // inheritance
    private int arg;            // communication variables
    private int result;
    public MyTask() {...}      // task constructors
    public void run() {...}    // task main
    public int result() {...}  // return result
    // unusual to have more members
}
```

- Thread starts in member run.

- Java requires explicit starting of a thread by calling start after the thread’s declaration.
  - coding convention to start thread or inheritance is precluded (can only start a thread once)

- Termination synchronization is accomplished by calling join.
• Returning a result on thread termination is accomplished by member(s) returning values from the task’s global variables.

```java
mytask th = new MyTask(...); // create and initialized task
th.start(); // start thread
// concurrency
th.join(); // wait for thread termination
a2 = th.result(); // retrieve answer from task object
```

• Like µC++, when the task’s thread terminates, it becomes an object, hence allowing the call to `result` to retrieve a result.

• (see Section 8.8, p. 143 for monitors)

• While it is possible to have public synchronized members of a task:
  - no mechanism to manage direct calls, i.e., no accept statement
  - ⇒ complex emulation of external scheduling with internal scheduling for direct communication

11.4.4 Go

• Non-object-oriented, light-weight (like µC++) non-preemptive threads (called goroutine).
  - ⇒ busy waiting only on multicore (Why?)

• `go` statement (like start/fork) creates new user thread running in routine.

```go
go foo( 3, f ) // start thread in routine foo
```

• Arguments may be passed to goroutine but return value is discarded.

• **Cannot reference goroutine object** ⇒ no direct communication.

• All threads terminate silently when program terminates.

• Threads synchronize/communicate via channel (CSP)
  - ⇒ **paradigm shift from routine call**.

• Channel is a typed shared buffer with 0 to N elements.

```go
ch1 := make( chan int, 100 ) // integer channel with buffer size 100
ch2 := make( chan string ) // string channel with buffer size 0
ch2 := make( chan chan string ) // channel of channel of strings
```

• Buffer size > 0 ⇒ up to N asynchronous calls; otherwise, synchronous call.

• Operator `<-` performs send/receive.
  - send: ch1 <- 1
  - receive: s <- ch2
• Channel can be constrained to only send or receive; otherwise bi-directional.

```go
package main
import "fmt"

func main() {
    type Msg struct { i, j int }
    ch1 := make(chan int)
    ch2 := make(chan float32)
    ch3 := make(chan Msg)
    hand := make(chan string)
    shake := make(chan string)
    gortn := func() {
        var i int;
        var f float32;
        var m Msg
        L:
        for {
            select {
                // wait for message
                case i = <- ch1: fmt.Println(i)
                case f = <- ch2: fmt.Println(f)
                case m = <- ch3: fmt.Println(m)
                case <- hand: break L // sentinel
            }
        }
        shake <- "SHAKE" // completion
    }
    go gortn() // start thread in gortn
    ch1 <- 0 // different messages
    ch2 <- 2.5
    ch3 <- Msg{1, 2}
    hand <- "HAND" // sentinel value
    <−shake // wait for completion
}
```

• Locks

```go
#include <iostream>
using namespace std;

_Task Gortn {
    public:
        struct Msg { int i, j; };
        void mem1( int i ) { Gortn::i = i; }
        void mem2( float f ) { Gortn::f = f; }
        void mem3( Msg m ) { Gortn::m = m; }
    private:
        int i; float f; Msg m;
    void main() {
        for ( ;; ) {
            _Accept( mem1 ) cout << i << endl;
            or _Accept( mem2 ) cout << f << endl;
            or _Accept( mem3 ) cout << "(" << m.i
                << ", " << m.j << ")" << endl;
            or_Accept( ~Gortn ) break;
        }
    }
}

int main() {
    Gortn gortn;
    gortn.mem1( 0 );
    gortn.mem2( 2.5 );
    gortn.mem3( (Gortn::Msg){ 1, 2 } );
    } // wait for completion
```

```cpp
#include <iostream>
using namespace std;

_Task Gortn {
    public:
        struct Msg { int i, j; };
        void mem1( int i ) { Gortn::i = i; }
        void mem2( float f ) { Gortn::f = f; }
        void mem3( Msg m ) { Gortn::m = m; }
    private:
        int i; float f; Msg m;
    void main() {
        for ( ;; ) {
            _Accept( mem1 ) cout << i << endl;
            or _Accept( mem2 ) cout << f << endl;
            or _Accept( mem3 ) cout << "(" << m.i
                << ", " << m.j << ")" << endl;
            or_Accept( ~Gortn ) break;
        }
    }
}

int main() {
    Gortn gortn;
    gortn.mem1( 0 );
    gortn.mem2( 2.5 );
    gortn.mem3( (Gortn::Msg){ 1, 2 } );
    } // wait for completion
}
```

```c
#define _Task Gortn {
    public:
        struct Msg { int i, j; };
        void mem1( int i ) { Gortn::i = i; }
        void mem2( float f ) { Gortn::f = f; }
        void mem3( Msg m ) { Gortn::m = m; }
    private:
        int i; float f; Msg m;
    void main() {
        for ( ;; ) {
            _Accept( mem1 ) cout << i << endl;
            or _Accept( mem2 ) cout << f << endl;
            or _Accept( mem3 ) cout << "(" << m.i
                << ", " << m.j << ")" << endl;
            or_Accept( ~Gortn ) break;
        }
    }
}

int main() {
    Gortn gortn;
    gortn.mem1( 0 );
    gortn.mem2( 2.5 );
    gortn.mem3( (Gortn::Msg){ 1, 2 } );
    } // wait for completion
```
CHAPTER 11. OTHER APPROACHES

```go
type WaitGroup // countdown lock
    func (wg *WaitGroup) Add(delta int)
    func (wg *WaitGroup) Done()
    func (wg *WaitGroup) Wait()
```

- Atomic operations

```go
func AddInt32(val *int32, delta int32) (new int32)
func AddInt64(val *int64, delta int64) (new int64)
func AddUint32(val *uint32, delta uint32) (new uint32)
func AddUint64(val *uint64, delta uint64) (new uint64)
func AddUintptr(val *uintptr, delta uintptr) (new uintptr)
func CompareAndSwapInt32(val *int32, old, new int32) (swapped bool)
func CompareAndSwapInt64(val *int64, old, new int64) (swapped bool)
func CompareAndSwapPointer(val *unsafe.Pointer, old, new unsafe.Pointer) (swapped bool)
func CompareAndSwapUint32(val *uint32, old, new uint32) (swapped bool)
func CompareAndSwapUint64(val *uint64, old, new uint64) (swapped bool)
func CompareAndSwapUintptr(val *uintptr, old, new uintptr) (swapped bool)
func LoadInt32(addr *int32) (val int32)
func LoadInt64(addr *int64) (val int64)
func LoadPointer(addr *unsafe.Pointer) (val unsafe.Pointer)
func LoadUint32(addr *uint32) (val uint32)
func LoadUint64(addr *uint64) (val uint64)
func LoadUintptr(addr *uintptr) (val uintptr)
func StoreInt32(addr *int32, val int32)
func StoreInt64(addr *int64, val int64)
func StorePointer(addr *unsafe.Pointer, val unsafe.Pointer)
func StoreUint32(addr *uint32, val uint32)
func StoreUint64(addr *uint64, val uint64)
func StoreUintptr(addr *uintptr, val uintptr)
```

11.4.5 C++11 Concurrency

- C++11 library can be sound as C++ now has strong memory-model (SC).

- compile: g++ -std=c++11 -pthread ...

- Thread creation: start/wait (fork/join) approach.

```go
class thread {
    public:
        template <class Fn, class... Args>
            explicit thread( Fn && fn, Args &... args );
        void join(); // termination synchronization
        bool joinable() const; // true => joined, false otherwise
        void detach(); // independent lifetime
        id get_id() const; // thread id
};
```

- Passing multiple arguments uses C++11’s variadic template feature to provide a type-safe call chain via thread constructor to the callable routine.
• Any entity that is callable (functor) may be started:

```cpp
#include <thread>

void hello( const string & s ) { // callable
    cout << "Hello " << s << endl;
}

class Hello { // functor
    int result;
    public:
        void operator() (const string & s) { // callable
            cout << "Hello " << s << endl;
        }
};

int main() {
    thread t1( hello, "Peter" ); // start thread in routine "hello"
    Hello h; // thread object
    thread t2( h, "Mary" ); // start thread in functor "h"
    // work concurrently
    t1.join(); // termination synchronization
    t2.join(); // termination synchronization
} // must join before closing block
```

• Thread starts implicitly at point of declaration.

• Instead of join, thread can run independently by detaching:

```cpp
t1.detach(); // "t1" must terminate for program to end
```

• Beware dangling pointers to local variables:

```cpp
{
    string s( "Fred" ); // local variable
    thread t( hello, s );
    t.detach();
} // "s" deallocated and "t" running with reference to "s"
```

• **It is an error to deallocate thread object before join or detach.**

• Locks
  
  - mutex, recursive, timed, recursive-timed

```cpp
class mutex {
    public:
        void lock(); // acquire lock
        void unlock(); // release lock
        bool try_lock(); // nonblocking acquire
};
```

  - condition
class condition_variable {
    public:
    void notify_one();  // unblock one
    void notify_all();  // unblock all
    void wait( mutex &lock );  // atomically block & release lock
};

• Scheduling is no-priority nonblocking ⇒ barging ⇒ wait statements must be in while loops to recheck conditions.

#include <mutex>
class BoundedBuffer {    // simulate monitor
    // buffer declarations
    mutex mlock;            // monitor lock
    condition_variable empty, full;
    void insert( int elem ) {
        mlock.lock();
        while (count == Size) empty.wait( mlock );   // release lock
        // add to buffer
        count += 1;
        full.notify_one();
        mlock.unlock();
    }

    int remove() {
        mlock.lock();
        while (count == 0) full.wait( mlock );   // release lock
        // remove from buffer
        count -= 1;
        empty.notify_one();
        mlock.unlock();
        return elem;
    }
};

• Futures

#include <future>
big_num pi( int decimal_places ) {...}
int main() {
    future<big_num> PI = async( pi, 1200 );  // PI to 1200 decimal places
    // work concurrently
    cout << "PI " << PI.get() << endl;  // block for answer
}

• Atomic types/operations

atomic_flag, atomic_bool, atomic_char, atomic_schar, atomic_uchar, atomic_short, atomic_ushort,
atomic_int, atomic_uint, atomic_long, atomic_ulong, atomic_llong, atomic_ullong, atomic_wchar_t,
avtomic_address, atomic<T>
typedef struct atomic_itype {
    bool operator==(int-type) volatile;
    void store(int-type) volatile;
    int-type load() const volatile;
    int-type exchange(int-type) volatile;
    bool compare_exchange(int-type &old_value, int-type new_value) volatile;
    int-type fetch_add(int-type) volatile;
    int-type fetch_sub(int-type) volatile;
    int-type fetch_and(int-type) volatile;
    int-type fetch_or(int-type) volatile;
    int-type fetch_xor(int-type) volatile;
    int-type operator++() volatile;
    int-type operator++(int) volatile;
    int-type operator--() volatile;
    int-type operator--(int) volatile;
    int-type operator+=(int-type) volatile;
    int-type operator-=(int-type) volatile;
    int-type operator&=(int-type) volatile;
    int-type operator|=(int-type) volatile;
    int-type operator^=(int-type) volatile;
} atomic_itype;

11.5 Threads & Locks Library

11.5.1 java.util.concurrent

- Java library is sound because of memory-model and language is concurrent aware.

- Synchronizers: Semaphore (counting), CountDownLatch, CyclicBarrier, Exchanger, Condition, Lock, ReadWriteLock

- Use new locks to build a monitor with multiple condition variables.

```java
class BoundedBuffer {
    final Lock mlock = new ReentrantLock();
    final Condition empty = mlock.newCondition();
    final Condition full = mlock.newCondition();

    public void insert(Object elem) throws InterruptedException {
        mlock.lock();
        try {
            while (count == Size) empty.await();
            count += 1;
            full.signal();
        } finally { mlock.unlock(); } // ensure monitor lock is unlocked
    }
```
public Object remove() throws InterruptedException {
    mlock.lock();
    try {
        while (count == 0) full.await();  // release lock
        // remove from buffer
        count -= 1;
        empty.signal();
        return elem;
    } finally { mlock.unlock(); } // ensure monitor lock is unlocked
}

- Condition is nested class within ReentrantLock ⇒ condition implicitly knows its associated (monitor) lock.
- Scheduling is still no-priority nonblocking ⇒ barging ⇒ wait statements must be in while loops to recheck condition.
- No connection with implicit condition variable of an object.
- Do not mix implicit and explicit condition variables.

• Executor/Future :

- Executor is a server with one or more worker tasks (worker pool).
- Call to executor submit is asynchronous and returns a future.
- Future is closure with work for executor (Callable) and place for result.
- Result is retrieved using get routine, which may block until result inserted by executor.

    import java.util.ArrayList;
    import java.util.List;
    import java.util.concurrent.*;

    public class Matrix {
        public static void main(String[] args) throws InterruptedException, ExecutionException {
            class Adder implements Callable<Integer> {
                int row[], cols;
                // communication
                public Integer call() {
                    int subtotal = 0;
                    for (int c = 0; c < cols; c += 1) subtotal += row[c];
                    return subtotal;
                }
                Adder(int[] r, int c) { row = r; cols = c; }
            }
        }
    }
int rows = 10, cols = 10;
int matrix[][] = new int[rows][cols], total = 0;
// read matrix
ExecutorService executor = Executors.newFixedThreadPool(4);
List<Future<Integer>> subtotals = new ArrayList<Future<Integer>>();
for (int r = 0; r < rows; r += 1) // send off work for executor
    subtotals.add(executor.submit(new Adder(matrix[r], cols)));
for (int r = 0; r < rows; r += 1) // wait for results
    total += subtotals.get(r).get(); // retrieve result
System.out.println(total);
executor.shutdown();

• µC++ also has fixed thread-pool executor.

struct Adder {
    // routine, functor or lambda
    int * row, cols;
    // communication
    int operator()() {
        // functor-call operator
        int subtotal = 0;
        for (int c = 0; c < cols; c += 1) subtotal += row[c];
        return subtotal;
    }
}
Adder(int row[], int cols) : row(row), cols(cols) {}

int main() {
    int rows = 10, cols = 10;
    int matrix[rows][cols], total = 0;
    // read matrix
    Executor executor(4); // kernel threads
    Future<Integer> subtotals[rows];
    Adder * adders[rows];
    for (int r = 0; r < rows; r += 1) {
        // send off work for executor
        adders[r] = new Adder(matrix[r], cols);
        subtotals[r] = executor.sendrecv(+adders[r]);
    }
    for (int r = 0; r < rows; r += 1) {
        // wait for results
        total += subtotals[r]();
        delete adders[r];
    }
    cout << total << endl;
}

• Collections : LinkedBlockingQueue, ArrayBlockingQueue, SynchronousQueue, PriorityBlockingQueue, DelayQueue, ConcurrentHashMap, ConcurrentSkipListMap, ConcurrentSkipListSet, CopyOnWriteArrayList, CopyOnWriteArraySet.

  ○ Create threads that interact indirectly through atomic data structures, e.g., producer-/consumer interact via LinkedBlockingQueue.

• Atomic Types using compare-and-assign (see Section 11.1.1, p. 175) (i.e., lock-free).

```java
int v;
AtomicInteger i = new AtomicInteger();
i.set(1);
System.out.println( i.get() );
v = i.addAndGet(1); // i += delta
System.out.println( i.get() + " + v ");
v = i.decrementAndGet(); // --i
System.out.println( i.get() + " + v ");
v = i.getAndAdd(1); // i += delta
System.out.println( i.get() + " + v ");
v = i.getAndDecrement(); // i--
System.out.println( i.get() + " + v ");
```

11.5.2 Pthreads

- Several libraries exist for C (pthreads) and C++ ($\mu$C++).

- C libraries built around routine abstraction and mutex/condition locks ("attribute" parameters not shown).

```c
int pthread_create( pthread_t * new_thread_ID, 
                   void * (+ start_func)(void *), void * arg );
int pthread_join( pthread_t target_thread, void ** status );
pthread_t pthread_self( void );

int pthread_mutex_init( pthread_mutex_t * mp );
int pthread_mutex_lock( pthread_mutex_t * mp );
int pthread_mutex_unlock( pthread_mutex_t * mp );
int pthread_mutex_destroy( pthread_mutex_t * mp );
int pthread_cond_init( pthread_cond_t * cp );
int pthread_cond_wait( pthread_cond_t * cp, pthread_mutex_t * mutex );
int pthread_cond_signal( pthread_cond_t * cp );
int pthread_cond_broadcast( pthread_cond_t * cp );
int pthread_cond_destroy( pthread_cond_t * cp );
```

- Thread starts in routine start_func via pthread_create.
  Initialization data is single void * value.

- Termination synchronization is performed by calling pthread_join.

- Return a result on thread termination by passing back a single void * value from pthread_join.
11.5. THREADS & LOCKS LIBRARY

```c
void * rtn( void * arg ) { ... }
int i = 3, r, rc;
pthread_t t;       // thread id
rc = pthread_create( &t, rtn, (void *)i ); // create and initialized task
if ( rc != 0 ) ... // check for error
// concurrency
rc = pthread_join( t, &r ); // wait for thread termination and result
if ( rc != 0 ) ... // check for error
```

- All C library approaches have type-unsafe communication with tasks.
- No external scheduling ⇒ complex direct-communication emulation.
- Internal scheduling is no-priority nonblocking ⇒ barging ⇒ wait statements must be in while loops to recheck conditions

```c
typedef struct { // simulate monitor
    // buffer declarations
    pthread_mutex_t mutex;       // mutual exclusion
    pthread_cond_t full, empty; // synchronization
} buffer;

// write your own constructor/destructor
void ctor( buffer * buf ) {     // constructor
    ...
    pthread_mutex_init( &buf->mutex );
    pthread_cond_init( &buf->full );
    pthread_cond_init( &buf->empty );
}

void dtor( buffer * buf ) { // destructor
    pthread_mutex_lock( &buf->mutex ); // must be mutex
    ...
    pthread_cond_destroy( &buf->empty );
    pthread_cond_destroy( &buf->full );
    pthread_mutex_destroy( &buf->mutex );
}

void insert( buffer * buf, int elem ) {
    pthread_mutex_lock( &buf->mutex );
    while ( buf->count == Size )
        pthread_cond_wait( &buf->empty, &buf->mutex );
    // add to buffer
    buf->count += 1;
    pthread_cond_signal( &buf->full );
    pthread_mutex_unlock( &buf->mutex );
}
```
int remove( buffer * buf ) {
    pthread_mutex_lock( &buf->mutex );
    while ( buf->count == 0 )
        pthread_cond_wait( &buf->full, &buf->mutex );
    // remove from buffer
    buf->count -= 1;
    pthread_cond_signal( &buf->empty );
    pthread_mutex_unlock( &buf->mutex );
    return elem;
}

• Since there are no constructors/destructors in C, explicit calls are necessary to ctor/dtor before/after use.

• All locks must be initialized and finalized.

• Mutual exclusion must be explicitly defined where needed.

• Condition locks should only be accessed with mutual exclusion.

• pthread_cond_wait atomically blocks thread and releases mutex lock, which is necessary to close race condition on baton passing.

11.6 OpenMP

• Shared memory, implicit thread management (programmer hints), 1-to-1 threading model (kernel threads), some explicit locking.

• Communicate with compiler with #pragma directives.

    #pragma omp ...

• fork/join model

  □ fork: initial thread creates a team of parallel threads (including itself)
  □ each thread executes the statements in the region construct
  □ join: when team threads complete, synchronize and terminate, except initial thread which continues

• compile: gcc -std=c99 -fopenmp openmp.c -lgomp

• COBEGIN/COEND: each thread executes different section:
#include <omp.h>

int main() {
  int i;
  #pragma omp parallel sections num_threads(4) // fork "4" threads
  { // COBEGIN
    #pragma omp section
    { i = 1; }
    #pragma omp section
    { p1(5); }
    #pragma omp section
    { p2(7); }
    #pragma omp section
    { p3(9); }
  } // COEND (synchronize)
}

• for directive specifies each loop iteration is executed by a team of threads (COFOR)

int main() {
  const unsigned int rows = 10, cols = 10; // sequential
  int matrix[rows][cols], subtotals[rows], total = 0;
  // read matrix
  #pragma omp parallel for // fork "rows" threads
  for (unsigned int r = 0; r < rows; r += 1) { // concurrent
    subtotals[r] = 0;
    for (unsigned int c = 0; c < cols; c += 1)
      subtotals[r] += matrix[r][c];
  }
  for (unsigned int r = 0; r < rows; r += 1) // sequential
    total += subtotals[r];
  printf("total:%d\n", total);
} // main

• In this case, sequential code directly converted to concurrent via #pragma.

• Variables outside section are shared; variables inside are thread private.

• Programmer responsible for sharing in vector/matrix manipulation.

• barrier

int main() {
  #pragma omp parallel num_threads(4) // fork "4" threads
  { sleep(omp_get_thread_num());
    printf("%d\n", omp_get_thread_num());
    #pragma omp barrier // wait for all block threads to arrive
    printf("sync\n");
  }
}
• Without `omp section`, all threads run same block (like `omp parallel for`).
• Barrier’s trigger is the number of block threads.
• Threads sleeps for different times, but all print "sync" at same time.
• Also critical section and atomic directives.
Index

_Accept, 133, 148, 150–152
destructor, 152
_At, 30, 72
_Coroutine, 24
_Disable, 35
_Enable, 34, 35
_Event, 29
_Monitor, 132
_Nomutex, 133
_Resume, 30
_Select, 159
_Task, 66
_Throw, 30, 34, 72
_When, 148, 150
1:1 threading, 61

ABA problem, 177
activation, 5
active, 21
Actor, 67
actor, 67
Ada 95, 184
adaptive spin-lock, 89
administrator, 154
worker tasks, 155
allocation
heap, 3
stack, 3
allocation graphs, 127
alternation, 76
Amdahl’s law, 62
arbiter, 84
atomic, 73, 80, 132, 198
atomic, 172
atomic consistent, 171
atomic instruction
compare/assign, 175
fetch-and-increment, 86
swap, 86
test/set, 86
automatic signal, 142
bakery, 82
banker’s algorithm, 126
barge, 93
barging, 75, 143, 197
barging avoidance, 93, 144
barging prevention, 93
barrier, 101, 103, 144
baton passing, 112
binary semaphore, 104, 134
BinSem, 106
acquire, 106
release, 106
bottlenecks, 58
bounded buffer, 110, 133, 134, 142, 148, 150–152
bounded overtaking, 80, 93
break, 3
labelled, 2
limitations, 3
buffering, 109
bounded, 110
unbounded, 110
busy wait, 72, 89, 91, 100
busy waiting, 75
C, 8
C++11, 190
atomic, 172
cache, 165
cache coherence, 167
coherence, 167
consistency, 167
eviction, 165
flush, 165
cache coherence, 167
cache consistency, 167
cache line, 165
cache thrashing, 168
call, 5
call-back, 156
catch, 14
catch-any, 51
channel, 188
client side, 155
call-back, 156
future, 157
returning values, 156
ticket, 156
COBEGIN, 64, 66, 108
cocall, 41
COEND, 64, 66
coherence, 167
communication, 72
direct, 147
compare-and-assign(set) instruction, 175
concurrency, 57
difficulty, 57
increasing, 153
why, 57
Concurrent C++, 186
concurrent error
indefinite postponement, 123
live lock, 123
race condition, 123
starvation, 123
concurrent exception, 14, 71
concurrent execution, 57
concurrent hardware
structure, 58
concurrent systems
explicit, 62
implicit, 61
structure, 61
condition, 134, 140
condition lock, 97, 100
conditional critical region, 131
consistency, 167
context switch, 24, 58
continue
labelled, 2
control dependency, 163
cooporation, 91, 93, 94
coprocessor, 182
coroutine, 21
coroutine main, 29, 47
critical path, 63, 182
critical region, 131
critical section, 73, 89
hardware, 85
compare/assign, 175
fetch-and-increment, 86
swap, 86
test/set, 86
self testing, 75
CUDA, 182
data dependency, 163
dating service, 136
deadlock, 124, 125, 131
allocation graphs, 127
avoidance, 126
banker’s algorithm, 126
detection/recovery, 128
mutual exclusion, 124
ordered resource, 125
prevention, 125
synchronization, 124
declare intent, 76
Dekker, 78, 172
delivered, 14
dependent execution, 63
derived exception-types, 51
destructor
__Accept, 152
detach, 191
detection/recovery, 128
direct communication, 147
disjoint, 169
distributed system, 59
divide-and-conquer, 69
double-check locking, 170
dynamic multi-level exit, 5, 7, 16
dynamic propagation, 16
eliding, 163
exception, 13
  concurrent, 14
  handling, 12
  handling mechanism, 12
  hierarchy, 29
  inherited members, 30
  list, 53
  nonlocal, 14
  parameters, 52
  type, 13, 29, 30
exception handling, 12
exception handling mechanism, 12
exception list, 53
exception parameters, 52
exception type, 13
exceptional event, 12
execution, 13
execution location, 21
execution state, 21
execution states, 21
  blocked, 59
  halted, 59
  new, 59
  ready, 59
  running, 59
execution status, 21
exit
  dynamic multi-level, 5
  static multi-exit, 1
  static multi-level, 2
explicit scheduling, 141
explicit signal, 142
external scheduling, 133, 148
failed cooperation, 140
failure exception, 53
false sharing, 168
faulting execution, 14, 33
fetch-and-increment instruction, 86
Fibonacci, 22
fix-up routine, 10
flag variable, 2, 140
flickering, 167
forward branch, 3
freshness, 115
fresh, 115
full coroutine, 41
full-coroutine, 21
functor, 191
future, 157
Future_ISM
  available, 158
  cancel, 158
  cancelled, 158
  delivery, 159
  exception, 159
  operator T(), 158
  operator(), 158
  reset, 159
garbage collection, 67
Gene Amdahl, 62
general-purpose GPU, 182
generalize kernel threading, 61
Go, 188
goroutine, 188
goto, 2, 3, 7
GPGPU, 182
guarded block, 14, 18, 51
handled, 14
handler, 14
  resumption, 18, 31
handlers, 29
  resumption, 29
  termination, 29
hazard pointers, 179
heap, 3
Heisenbug, 58
Hesselink, 79
immediate-return signal, 142
implicit scheduling, 141
implicit signal, 142
inactive, 21
increasing concurrency, 153
indefinite postponement, 74, 123
independent execution, 63
inherited members
  exception type, 30
internal scheduling, 134, 150
interrupt, 58, 59, 180
invocation, 5
isacquire, 96
istream
   isacquire, 96
iterator, 37
Java, 187
   volatile, 172
Java monitor, 143
jmp_buf, 8
kernel threading, 61
kernel threads, 60
keyword, additions
   _Accept, 133
   _At, 30
   _Coroutine, 24
   _Disable, 35
   _Enable, 35
   _Event, 29
   _Monitor, 132
   _Nomutex, 133
   _Resume, 30
   _Select, 159
   _Task, 66
   _Throw, 30
   _When, 148
lexical link, 32
linear, 62
linear speedup, 62
livelock, 74
liveness, 74
local exception, 14
lock, 75
   taxonomy, 89
   techniques, 111
lock composition, 140
lock free, 175
lock programming
   buffering, 109
      bounded buffer, 110
      unbounded buffer, 110
lock-release pattern, 95
longjmp, 8, 172

loop
   mid-test, 1
   multi-exit, 1
M:1 threading, 61
M:N threading, 61
main
   task, 66
match, 14
memory model, 171
mid-test loop, 1
modularization, 5
monitor, 132
   condition, 134, 140
   external scheduling, 133
   internal scheduling, 134
   scheduling, 133
   signal, 134, 140
   wait, 134, 140
monitor type
   no priority blocking, 143
   no priority implicit signal, 143
   no priority nonblocking, 143
   priority blocking, 143
   priority implicit signal, 143
   priority nonblocking, 143
monitor types, 141
multi-exit
   Multi-exit loop, 1
   mid-test, 1
      multi-level
         dynamic, 16
multi-level exit
   dynamic, 5
   static, 2
multiple acquisition, 91
multiple outcomes, 5
multiprocessing, 58
multiprocessor, 59
multitasking, 58
mutex lock, 91, 92, 106, 132
mutex member, 132
MutexLock, 92, 132
   acquire, 92, 132
   release, 92, 132
mutual exclusion, 73, 110
alternation, 76
deadlock, 124
declare intent, 76
Dekker, 78
Dekker-Hesselink, 79
game, 74
lock, 75, 86
N-thread
arbiter, 84
bakery, 82
prioritized entry, 81
tournament, 82
Peterson, 80
prioritized retract intent, 77
retract intent, 77

N:N threading, 61
nano threads, 61
nested monitor problem, 140
no priority blocking, 143
no priority implicit signal, 143
no priority nonblocking, 143
non-linear, 62
speedup, 62
non-preemptive, 58
scheduling, 58
nonlocal exception, 14, 33
nonlocal transfer, 7, 16

object
threading, 65
OpenMP, 198
operating system, 60, 61
optimization, 163
ordered resource, 125, 129
ostream
osacquire, 96
owner lock, 91, 95
ownership, 175

P, 104, 124, 125, 131, 140
parallel execution, 57
partial store order, 57
passeren, 104
Peterson, 80

precedence graph, 108
preemptive, 58
scheduling, 58
prioritized entry, 81
prioritized retract intent, 77
priority blocking, 143
priority implicit signal, 143
priority nonblocking, 143
private semaphore, 119
process, 57
processor
multi, 59
uni, 58
program order, 163
prolagen, 104
propagation, 14, 29
dynamic, 16
static, 15
propagation mechanism, 14
pthreads, 196

race condition, 123
race free, 171
raise, 13, 29, 30
resuming, 29, 30
throwing, 29, 30
readers/writer, 136
freshness, 115
monitor
solution 3, 136
solution 4, 138
solution 8, 139
semaphore, 113
solution 1, 113
solution 2, 114
solution 3, 115
solution 4, 115
solution 5, 117
solution 6, 118
solution 7, 120
staleness, 115
real time, 62
release consistency, 171
RendezvousFailure
failed cooperation, 140
reordering, 163
replication, 163
reraise, 14
reservation, 180
resume, 24, 34, 42
resumption, 14, 18
resumption handler, 18, 31
retract intent, 77
retry, 18
return code, 9
return union, 10
routine
  activation, 5
routine abstraction, 196
rw-safe, 78

safety, 74
scheduling, 58, 133, 148
  explicit, 141
  external, 133, 148
  implicit, 141
  internal, 134, 150
scrambling, 167
select statement, 159
self testing, 75
semaphore, 124, 125
  binary, 134
  counting, 106, 140
  integer, 106
  P, 104, 124, 125, 131, 140
  private, 119
  split binary, 111
  V, 105, 131, 140
semi-coroutine, 21, 41
sequel, 15
sequence points, 172
sequential consistency, 171
server side
  administrator, 154
  buffer, 154
setjmp, 8
shared-memory, 60
signal, 140
  automatic, 142
  explicit, 142
  immediate-return, 142
  implicit, 142
signal, 134, 151, 152
single acquisition, 91
software transactional memory, 181
source execution, 13, 33
speedup. 62
  linear, 62
  non-linear, 62
  sub-linear, 62
  super linear, 62
spin lock, 89
  implementation, 90
split binary semaphore, 111
spurious wakeup, 144
SR, 186
stack allocation, 3
stack unwinding, 8, 14
staleness, 115
stale, 115
START, 65, 66, 108
starter, 41
starvation, 74, 123
state transition, 59
static exit
  multi-exit, 1
  multi-level, 2
static multi-level exit, 2
static propagation, 15
static variable, 73
status flag, 9
stream lock, 96
strong memory ordering, 168
sub-linear, 62
  speedup, 62
super linear, 62
super-linear speedup, 62
suspend, 24
swap instruction, 86
synchronization, 110, 133
  communication, 72
  deadlock, 124
  during execution, 72
  termination, 69
synchronization lock, 97
11.6. OPENMP

- task, 57
- exceptions, 71
- external scheduling, 148
- internal scheduling, 150
- main, 66
- scheduling, 148
- static variable, 73
- terminate, 16
- terminated, 21
- termination, 14
- termination synchronization, 69, 102
- test-and-set instruction, 86
- thread, 57
  - communication, 64
  - creation, 64
  - synchronization, 64
- thread graph, 64
- thread object, 65
- threading model, 60
- throw, 13
- ticket, 119, 156
- time-slice, 89, 119, 179
- total store order, 171
- tournament, 82
- transaction, 181
- TSO, 172

μC++, 17, 157

uCondition, 134, 151, 152
  - signal, 134
  - wait, 134
uCondLock, 100
  - broadcast, 100
  - empty, 100
  - signal, 100
  - wait, 100
uLock, 90
  - acquire, 90
  - release, 90
  - tryacquire, 90
unbounded buffer, 110
unbounded overtaking, 79
unfairness, 75
unguarded block, 14
uniprocessor, 58
uOwnerLock, 95
  - acquire, 95
  - release, 95
  - times, 95
  - tryacquire, 95
uSemaphore, 108, 124, 125
  - counter, 108
  - empty, 108
  - P, 108, 124, 125
  - TryP, 108
  - V, 108
user threading, 61
user time, 62
uSpinLock, 90
  - acquire, 90
  - release, 90
  - tryacquire, 90
V, 105, 131, 140
virtual machine, 61
virtual processors, 60
volatile, 172

WAIT, 65, 66
wait, 140
wait, 134, 151, 152
wait free, 175
weak memory ordering, 168
weak order, 171
worker task, 154
worker tasks, 155
  complex, 155
courier, 155
notifier, 155
simple, 155
timer, 155

yield, 89