Concurrency in C++

Peter A. Buhr

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

Except as otherwise noted, the content of this presentation is licensed under the Creative Commons Attribution 2.5 License.
## Contents

1 Introduction .............................................. 1
   1.1 Why Concurrency ...................................... 1
   1.2 Why C++ .................................................. 1
   1.3 Concurrency in C++ ..................................... 1
   1.4 High-Level Concurrency ................................. 2
   1.5 µC++ : Advanced Control Flow for C++ ................. 2
   1.6 Outline .................................................. 3

2 Coroutine .................................................. 5
   2.1 Semi-Coroutine .......................................... 6
      2.1.1 Fibonacci Sequence .................................. 6
      2.1.1.1 Direct ............................................. 6
      2.1.1.2 Routine ........................................... 7
      2.1.1.3 Class .............................................. 8
      2.1.1.4 Coroutine ......................................... 9
      2.1.2 Formatted Output .................................... 10
      2.1.2.1 Direct ............................................. 10
      2.1.2.2 Routine ........................................... 11
      2.1.2.3 Class .............................................. 12
      2.1.2.4 Coroutine ......................................... 13
      2.1.3 Correct Coroutine Usage ............................. 13
      2.1.4 Device Driver ........................................ 14
      2.1.5 Producer-Consumer .................................. 16
   2.2 Full Coroutines ......................................... 18
      2.2.1 Producer-Consumer .................................. 20
   2.3 Nonlocal Exceptions .................................... 22

3 Concurrency ............................................... 25
   3.1 Why Write Concurrent Programs ......................... 25
   3.2 Why Concurrency is Difficult ............................ 25
   3.3 Structure of Concurrent Systems ......................... 26
   3.4 Structure of Concurrent Hardware ....................... 27
   3.5 Execution States ........................................ 28
   3.6 Thread Creation ......................................... 28
   3.7 Termination Synchronization ............................. 31
3.8 Synchronization and Communication during Execution ....................... 32
3.9 Communication ........................................................................... 32
3.10 Exceptions .................................................................................. 33
3.11 Critical Section ........................................................................... 34
3.12 Static Variables ........................................................................... 34
3.13 Mutual Exclusion Game ................................................................. 35
3.14 Self-Testing Critical Section ............................................................ 36
3.15 Software Solutions ....................................................................... 36
  3.15.1 Lock ....................................................................................... 36
  3.15.2 Alternation .............................................................................. 37
  3.15.3 Declare Intent ......................................................................... 37
  3.15.4 Retract Intent .......................................................................... 38
  3.15.5 Prioritize Entry ....................................................................... 39
  3.15.6 Dekker .................................................................................... 40
  3.15.7 Peterson .................................................................................. 41
  3.15.8 N-Thread Prioritized Entry ....................................................... 42
  3.15.9 N-Thread Bakery (Tickets) . ..................................................... 43
  3.15.10 Tournament ........................................................................... 44
  3.15.11 Arbiter .................................................................................. 44
3.16 Hardware Solutions ...................................................................... 45
  3.16.1 Test/Set Instruction .................................................................. 46
  3.16.2 Swap Instruction ....................................................................... 47
  3.16.3 Compare/Assign Instruction ...................................................... 47
4 Lock Abstraction .............................................................................. 49
  4.1 Lock Taxonomy ............................................................................. 49
  4.2 Spin Lock ...................................................................................... 49
    4.2.1 Spin Lock Details .................................................................... 50
  4.3 Blocking Locks ............................................................................ 51
    4.3.1 Mutex Lock ............................................................................. 51
      4.3.1.1 Mutex Lock Implementation ............................................. 52
      4.3.1.2 uOwnerLock Details ......................................................... 54
      4.3.1.3 Stream Locks ................................................................... 54
    4.3.2 Synchronization Lock ............................................................... 55
      4.3.2.1 Synchronization Lock Implementation ................................ 55
      4.3.2.2 uCondLock Details ............................................................ 56
    4.3.3 Barrier .................................................................................... 57
    4.3.4 Semaphore ............................................................................. 58
    4.3.5 Counting Semaphore ............................................................... 60
      4.3.5.1 uSemaphore Details ........................................................ 61
  4.4 Semaphore Programming ............................................................... 61
# 8 Other Approaches

## 8.1 Languages with Concurrency Constructs

- **8.1.1 Ada 95**
- **8.1.2 Modula-3/Java/C#**

## 8.2 Threads & Locks Library

## 8.3 Threads & Message Passing

- **8.3.1 Nonblocking Send**
- **8.3.2 Blocking Send**
- **8.3.3 Send-Receive-Reply**

## 8.4 Message Format

## Index
1 Introduction

1.1 Why Concurrency

- Processor speed has slowed (stopped).
- Use transistors from Moore’s law for parallelism to increase speed.
- But concurrent programming is necessary to utilize parallel hardware.
- Some success in implicitly discovering concurrency in sequential programs.
- Fundamental limits to finding parallelism and only for certain kinds of problems.
- Alternatively, programmer explicitly thinks about and specifies the concurrency.
- Implicit and explicit approaches are complementary, i.e., can appear together.
- However, the limitations of the implicit approach mandate an explicit approach.

1.2 Why C++

- C++ is one of the dominant programming languages.
- Why?
  - based on C, which has a large programmer and code base,
  - as efficient as C in most situations,
  - low-level features, e.g., direct memory access, needed for:
    * systems programming
    * memory management
    * embedded / real-time
  - high-level features, e.g., objects, polymorphism, exception handling, STL, etc., which programmers now demand,
  - allows direct interaction with UNIX/Windows.

1.3 Concurrency in C++

- C++ has no concurrency!
- Many different concurrency approaches for C++ have been implemented with only varying degrees of adoption.
- No de facto approach dominating concurrent programming in C++.
  - C has two dominant but incompatible concurrency libraries: pthreads and Win32.
- C++’s lack of concurrency limits its future in the parallel domain.
Therefore, it is imperative C++ be augmented with concurrency facilities to extend its programming base.

Interestingly, concurrency CANNOT be safely added to ANY language via library code.

1.4 High-Level Concurrency

- Want a single consistent high-level powerful concurrency mechanism, but what should it look like?

- In theory, any high-level concurrency paradigm/model can be adapted into C++.

- However, C++ does not support all concurrency approaches equally well, e.g., tuple space, message passing, channels.

- C++ is fundamentally based on a class model using routine call, and its other features leverage this model.

- Any concurrency approach matching the C++ model is better served because its concepts interact consistently with the language.

- Apply “Principle of Least Astonishment” whenever possible.

- Let C++ dictate which concurrency approaches fits best via its design principles.

- For OO language, thread/stack is best associated with class, and mutual-exclusion/synchronization with member routines.

1.5 \( \mu \text{C++} \): Advanced Control Flow for C++

- integrate advanced control flow tightly into C++
  - leverage all class features
  - threads are light-weight: M:N thread model versus 1:1

- use mutex objects to contain mutual exclusion and synchronization
  - communication using routine call (versus messages/channels)
  - statically typed

- handle multi-processor environment
1.6 Outline

- coroutines: suspending and resuming (concurrency precursor)
- concurrency: introduce multiple threads
- locking: thread control through synchronization and mutual exclusion
- errors: new concurrency issues
- monitors: high-level thread control
- tasks: active objects
- other concurrent approaches: different concurrency paradigms
2 Coroutine

- A coroutine is a routine that can also be suspended at some point and resume 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.
    ⇒ each coroutine has its own stack, containing its local variables and those of any routines it calls.
  - 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).

- A coroutine executes synchronously with other coroutines; hence, no concurrency among coroutines.

- Coroutines are the precursor to concurrent tasks, and introduce the complex concept of suspending and resuming on separate stacks.

- 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.

### 2.1 Semi-Coroutine

#### 2.1.1 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 the sequence: 0, 1, 1, 2, 3, 5, 8, 13, 21, ...

#### 2.1.1.1 Direct

- Assume output can be generated at any time.

```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 program into a routine that generates a sequence of Fibonacci numbers on each call (no output in routine):

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

- Examine different solutions.
2.1.1.2 Routine

```c
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;
}
```

- unencapsulated global variables necessary to retain state between calls
- only one fibonacci generator can run at a time
- execution state must be explicitly retained
2.1.1.3 Class

```cpp
class fibonacci {
  int fn, fn1, fn2, state; // global class variables
public:
  fibonacci() : state(1) {} 
  int next() {
    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;
  }
};

int main() {
  fibonacci f1, f2;
  for ( int i = 1; i <= 10; i += 1 ) {
    cout << f1.next() << " " << f2.next() << endl;
  } // for
}
```

- unencapsulated program global variables becomes encapsulated object global variables

- multiple fibonacci generators (objects) can run at a time

- execution state must still be explicitly retained
2.1.4 Coroutine

```cpp
#include <uC++.h>  // first include file
#include <iostream>
using namespace std;

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 next() {
            resume(); // transfer to last suspend
            return fn;
        }
};

void uMain::main() { // argc, argv class variables
    fibonacci f1, f2;
    for (int i = 1; i <= 10; i += 1) {
        cout << f1.next() << " " << f2.next() << endl;
    }
}
```

- **no explicit execution state!** (see direct solution)
- distinguished member main (coroutine main) can be suspended and resumed
- first resume starts main on new stack (cocall); subsequent resumes restart last suspend.
- suspend restarts last resume
- object becomes a coroutine on first resume; coroutine becomes an object when main ends
- both statements cause a context switch between coroutine stacks
- routine frame at the top of the stack knows where to activate execution

- coroutine main does not have to return before coroutine object is deleted

- uMain is the initial coroutine started by μC++

- argc and argv are implicitly defined in uMain::main

- <uC++.h> must be first include

- compile with u++ command

2.1.2 Formatted Output

Unstructured input:

abcdefghijklmnopqrstuvwxyzabcdefghijklmnopqrstuvwxy z

Structured output:

abcd efgh ijkl mnop qrst uvwx yzab cdef ghi jklmnopqrstuvwxyz

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

2.1.2.1 Direct

- Assume input can be obtained at any time.
2.1. SEMI-COROUTINE

```cpp
int main() {
    int g, b;
    char ch;

    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
                cin >> ch; // read one character
                if ( cin.eof() ) goto fini; // eof ? multi-level exit
                cout << ch; // print character
            }
            cout << " "; // print block separator
        }
        cout << endl; // print group separator
    }
    fini: ;
    if ( g != 0 || b != 0 ) cout << endl; // special case
}

• Convert program into a routine passed one character at a time to generate structured output (no input in routine).

2.1.2.2 Routine

```cpp
int g, b; // global variables
void fmtLines( char ch ) {
    if ( ch == EOF ) { cout << endl; return; }
    if ( ch == '\n' ) return; // ignore newline characters
    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;
    }
}
```
```cpp
int main() {
    char ch;
    cin >> noskipws; // turn off white space skipping
    for ( ;; ) { // for as many characters
        cin >> ch;
        if ( cin.eof() ) break; // eof ?
        fmtLines( ch );
    }
    fmtLines( EOF );
}
```

- must retain variables `b` and `g` between successive calls.
- only one instance of formatter
- routine `fmtLines` must flattening two nested loops into assignments and `if` statements.

### 2.1.2.3 Class

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

public:
    FmtLines() : g( 0 ), b( 0 ) {}  
    ~FmtLines() { if ( g != 0 || b != 0 ) cout << endl; }
    void prt( char ch ) {  
        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;
        }
    }
};
```

```cpp
int main() {
    FmtLines fmt;
    char ch;
    for ( ;; ) { // for as many characters
        cin >> ch; // read one character
        if ( cin.eof() ) break; // eof ?
        fmt.prt( ch );
    }
}
```
• Solves encapsulation and multiple instances issues, but still explicitly managing execution state.

2.1.2.4 Coroutine

```cpp
Coroutine FmtLines {
  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();
          cout << ch;     // print character
        }
        cout << " ";    // block separator
      }
      cout << endl;    // group separator
    }
  }
}
```

public:

```cpp
FmtLines() { resume(); }   // start coroutine
~FmtLines() { if ( g != 0 || b != 0 ) cout << endl; }
void prt( char ch ) { FmtLines::ch = ch; resume(); }
};
```

```cpp
void uMain::main() {
  FmtLines fmt;
  char ch;
  for ( ;; ) {
    cin >> ch;             // read one character
    if ( cin.eof() ) break; // eof ?
    fmt.prt( ch );
  }
}
```

• resume in constructor allows coroutine main to get to 1st input suspend.

2.1.3 Correct Coroutine Usage

• Eliminate unnecessary computation or flag variables used to retain information about execution state.
E.g., sum the even and odd digits of a 10-digit number, where each digit is passed to the 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>        odd += digit;</code></td>
</tr>
<tr>
<td><code>        suspend();</code></td>
<td><code>        suspend();</code></td>
</tr>
</tbody>
</table>

Right example illustrates the “Zen” of the coroutine; let it do the work.

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

```c
void main() {
    int fn1, fn2, state = 1;
    for ( ;; ) {
        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;
        }
        suspend();
    }
}
```

Uses explicit flag variables to control execution state and a single suspend at the end of an enclosing loop.

None of the coroutine’s capabilities are used, and the program structure is lost in `switch` statement.

Must do more than just `activate` the coroutine main to demonstrate an understanding of retaining data and execution state within a coroutine.

2.1.4 Device Driver

Called by interrupt handler for hardware serial port.
• Parse transmission protocol and return message text.

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

_Coroutine SerialDriver {
    unsigned char byte;
    int status;
    unsigned char *msg;

public:
    driver( unsigned char *msg ) : msg( msg ) { resume(); }
    int next( unsigned char b ) {
        // called by interrupt handler
        byte = b;
        resume();
        return status;
    }

private:
    void main() {
        newmsg:
        for (; ; ) {
            // parse messages
            status = CONT;
            int lnth = 0, sum = 0;
            do {
                suspend();
            } while ( byte != STX ) // look for start of message
            eomsg:
            for (; ; ) {
                suspend(); // parse message bytes
                switch ( byte ) {
                    case STX: // protocol violation
                        status = ERROR;
                        continue newMsg; // uC++ labelled continue
                    case ETX: // end of message
                        break eomsg; // uC++ labelled break
                    case ESC: // escape next character
                        suspend(); // get escaped character
                        break;
                } // switch
                msg[lnth] = byte; // store message
                lnth += 1;
                sum += byte; // compute CRC
            } // for
            suspend(); // obtain 1st CRC byte
            int crc = byte;
            suspend(); // obtain 2nd CRC byte
            crc = (crc << 8) | byte;
            status = crc == sum ? MSG : ERROR;
            msgcomplete( msg, lnth ); // return message to OS
        } // for
    } // main
}; // SerialDriver
2.1.5 Producer-Consumer

```cpp
_Coroutine Cons {
    int p1, p2, status; bool done;
    void main() { // starter prod
        int money = 1;
        status = 0;
        // 1st resume starts here
        for ( ;; ) {
            if ( done ) break;
            cout << "receives:" << p1 << ", " << p2;
            cout << " and pays $" << money << endl;
            status += 1;
            suspend(); // activate delivery or stop
            money += 1;
        }
        cout << "Cons stops" << endl;
    } // suspend / resume(starter)

    public:
    Cons() : 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
};
```
```cpp
Coroutine Prod {
    Cons &c;
    int N;
    void main() { // starter umain
        int i, p1, p2, status;
        // 1st resume starts here
        for (i = 1; i <= N; i += 1) {
            p1 = rand() % 100;
            p2 = rand() % 100;
            cout << "delivers:" << p1 << ", " << p2 << endl;
            status = c.delivery(p1, p2);
            cout << "gets status:" << status << endl;
        }
        cout << "Prod stops" << endl;
        c.stop();
    } // suspend / resume(starter)

    public:
        Prod( Cons &c ) : c(c) {}
        void start( int N ) {
            Prod::N = N;
            resume(); // activate main
        }
    }
};

void uMain::main() { // instance called umain
    Cons cons; // create consumer
    Prod prod( cons ); // create producer
    prod.start( 5 ); // start producer
} // resume(starter)
```
• Do both Prod and Cons need to be coroutines?
• When coroutine main returns, it activates the coroutine that *started* main.
• prod started cons.main, so control goes to prod suspended in stop.
• uMain started prod.main, so control goes back to uMain suspended in start.

2.2 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-routine.

```cpp
_Coroutine fc {
    void main() { // starter umain
        mem();    // ?
        resume(); // ?
        suspend(); // ?
    }
    public:
    void mem() { resume(); }
};
void uMain::main() {
    fc x;
    x.mem();
}
```

**control flow semantics**

<table>
<thead>
<tr>
<th>inactive</th>
<th>active</th>
</tr>
</thead>
<tbody>
<tr>
<td>suspend</td>
<td>last resumer</td>
</tr>
<tr>
<td>uThisCoroutine() resume this context switch</td>
<td></td>
</tr>
</tbody>
</table>

```dia
diagram showing control flow and coroutine states
```
• Suspend inactivates the current active coroutine (\texttt{uThisCoroutine}), and activates last resumer.

• Resume inactivates the current active coroutine (\texttt{uThisCoroutine}), and activates the current object (\texttt{this}).

• Hence, the current object \textit{must} be a non-terminated coroutine.

• Note, \texttt{this} and \texttt{uThisCoroutine} change at different times.

• Exception: last resumer not changed when resuming self because no practical value.

• 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

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

• The problem is mutually recursive references:
  \[ \texttt{fc x(y), y(x)}; \]

• One solution is to make closing the cycle a special case:
  \[ \texttt{fc x, y(x);} \]
  \[ x.\text{partner}( y ); \]

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

• Stopping can be as complex as starting, because a coroutine goes back to its starter.

• In many cases, it is unnecessary to terminate all coroutines, just delete them.

• But it is necessary to activate \texttt{uMain} for the program to finish (unless \texttt{exit} is used).
2.2.1 Producer-Consumer

```cpp
_Coroutine Prod {
   Cons *c;
   int N, money, receipt;
   void main() { // starter umain
      int i, p1, p2, status;
      // 1st resume starts here
      for ( i = 1; i <= N; i += 1 ) {
         p1 = rand() % 100;
         p2 = rand() % 100;
         cout << "delivers:" << p1 << " , " << p2 << endl;
         status = c->delivery( p1, p2 );
         cout << " gets status:" << status << endl;
         receipt += 1;
      }
      cout << "Prod stops" << endl;
      c->stop();
   }
   public:
   int payment( int money ) {
      Prod::money = money;
      cout << " gets payment $" << money << endl;
      resume(); // activate Cons::delivery
      return receipt;
   }
   void start( int N, Cons &c ) {
      Prod::N = N; Prod::c = &c;
      receipt = 0;
      resume(); // activate main
   }
};
```
2.2. FULL COROUTINES

Coroutine Cons {
    Prod &p;
    int p1, p2, status;
    bool done;
    void main() { // starter prod
        int money = 1, receipt;
        status = 0;
        // 1st resume starts here
        for ( ;; ) {
            if ( done ) break;
            if ( done ) break;
            cout << "receives: "
                << p1 << " , " << p2
                << " and pays $";
            status += 1;
            receipt = p.payment(money);
            cout << "gets receipt #";
                << receipt << endl;
            money += 1;
        }
        cout << "Cons stops" << endl;
    }
    public:
        Cons( Prod &p ) : p(p), done(false) {}{ public:
        int delivery( int p1, int p2 ) {
            Cons::p1 = p1; Cons::p2 = p2;
            resume(); // activate Cons::main 1st time
            return status; // afterwards Prod::payment
        }
        void stop() {
            done = true;
            resume(); // activate main
        }
    };
    void uMain::main() {
        Prod prod;
        Cons cons( prod );
        prod.start( 5, cons );
    }
}
2.3 Nonlocal Exceptions

- Exception handling is based on traversing a call stack.

- With coroutines there are multiple call stacks.

- Nonlocal exceptions move from one coroutine stack to another.

  ```
  _Throw [ throwable-event [ _At coroutine-id ] ]
  ```

- Hence, exceptions can be handled locally within a coroutine or nonlocally among coroutines.

- Local exceptions within a coroutine are the same as for exceptions within a routine/class, with one nonlocal difference:

  - 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 {};  // uC++ exception type
    _Coroutine C {
      void main() { _Throw E(); }
      public:
        void mem() { resume(); }
    }
    void uMain::main() {
      C c;
      try { c.mem();
      } catch ( uBaseCoroutine::UnhandledException ) {} ...
    }
    ```

  - 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 `uMain`, since it last resumed `c`.

  - 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.
2.3. NONLOCAL EXCEPTIONS

```cpp
_CORoutine C {  
    void main() {  
        for ( int i = 0; i < 5; i += 1 ) {  
            try {  
                _Enable { // allow nonlocal exceptions  
                    ... suspend(); ...  
                }  
            } catch( E ) { ... }  
        }  
    }  
}

public:  
    C() { resume(); } // prime loop  
    void mem() { resume(); }
};  

void uMain::main() {  
    C c;  
    for ( int i = 0; i < 5; i += 1 ) {  
        _Throw E() _At c; // exception pending  
        c.mem(); // trigger exception  
    }  
}
```

- **Nonlocal delivery is initially disabled for a coroutine**, so handlers can be set up before any exception can be delivered (also see Section 3.10, p. 33).

- Hence, nonlocal exceptions must be explicitly enabled before delivery can occur with `_Enable`.

- µC++ allows dynamic enabling and disabling of nonlocal event delivery.

  ```cpp
  _Enable <E1><E2>... {  
      // exceptions E1, E2 are enabled  
  }
  _Disable <E1><E2>... {  
      // exceptions E1, E2 are disabled  
  }
  ```

- Specifying no exceptions enables/disables all nonlocal exceptions.

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

- The source coroutine delivers the nonlocal exception immediately but does not propagate it; propagation only occurs when the faulting coroutine becomes active.

  ⇒ must call one of the faulting coroutine’s members that does a `resume`. 
3 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.

3.1 Why Write Concurrent Programs

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

• Expressing a problem with multiple executing threads may be the natural (best) way of describing it.

• Multiple executing threads can enhance the execution-time efficiency, by taking advantage of inherent concurrency in an algorithm and of any parallelism available in the computer system.

3.2 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.

  – Especially when the things interact with one another.

• 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?
  – 1,2,3, ... N, where N is the number of items of furniture
  – more than N?
• Where are the bottlenecks?
  – the door out of the room, items in front of other items, large items
• What communication is necessary between the helpers?
  – which item to take next
  – some are fragile and need special care
  – big items need several helpers working together

3.3 Structure of Concurrent Systems
• Concurrent systems can be divided into 3 major types:
  1. those that attempt to discover 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
• In case 1, there is a fundamental limit to how much parallelism can be found and current techniques only work on certain kinds of programs.
• In case 2, threads are accessed indirectly via specialized mechanisms (e.g., pragmas or parallel for) and implicitly managed.
• In case 3, threads are directly access and explicitly managed.
• Cases 1 & 2 are always built from case 3.
• To solve all concurrency problems, threads needs to be explicit.
Both implicit and explicit mechanisms are complementary, and hence, can appear together in a single programming language.

However, the limitations of implicit mechanisms require that explicit mechanisms always be available to achieve maximum concurrency.

μC++ only supports explicit mechanisms, but nothing in its design precludes implicit mechanisms.

Some concurrent systems provide a single technique or paradigm that must be used to solve all concurrent problems.

While a particular paradigm may be very good for solving certain kinds of problems, it may be awkward or preclude other kinds of solutions.

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.

Fundamentally, as the amount of concurrency increases, so does the complexity to express and manage it.

### 3.4 Structure of Concurrent Hardware

Concurrent execution of threads is possible on a computer which has only one CPU (uniprocessor); multitasking for multiple tasks or multiprocessing for multiple processes.

- Parallelism is simulated by rapidly context switching the CPU back and forth between threads.
- Unlike coroutines, task switching may occur at non-deterministic program locations, i.e., between any two machine instructions.
- Switching is usually based on a timer interrupt that is independent of program execution.

Or on the same computer, which has multiple CPUs, using separate CPUs but sharing the same memory (multiprocessor):
These tasks run in parallel with each other.

- Processes may be on different computers using separate CPUs and separate memories (distributed system):

```
<table>
<thead>
<tr>
<th>Computer 1</th>
<th>Computer 2</th>
</tr>
</thead>
<tbody>
<tr>
<td>CPU process</td>
<td>CPU process</td>
</tr>
<tr>
<td>state</td>
<td>program</td>
</tr>
<tr>
<td>100</td>
<td>7</td>
</tr>
</tbody>
</table>
```

These processes run in parallel with each other.

- By examining the first case, which is the simplest, all of the problems that occur with parallelism can be illustrated.

### 3.5 Execution States

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

\[
\text{new} \rightarrow \text{ready} \rightarrow \text{running} \rightarrow \text{halted}
\]

\[\text{blocked} \text{ (waiting)}\]

- **state transitions** are initiated in response to events:
  
  - timer alarm (running $\rightarrow$ ready)
  - completion of I/O operation (blocked $\rightarrow$ ready)
  - exceeding some limit (CPU time, etc.) (running $\rightarrow$ halted)
  - exceptions (running $\rightarrow$ halted)

### 3.6 Thread Creation

- Concurrency requires the ability to specify the following 3 mechanisms in a programming language.

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

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

- ⇒ need new construct to create a thread and define where the thread starts execution, e.g., COBEGIN/COEND:

\[
\begin{align*}
\text{BEGIN} & \quad \text{initial thread creates internal threads}, \\
\text{COBEGIN} & \quad \text{one for each statement in this block} \\
\text{BEGIN} & \quad i := 1; \ldots \text{END;} \\
p1(5); & \quad \text{order and speed of execution} \\
p2(7); & \quad \text{of internal threads is unknown} \\
p3(9); & \\
\text{COEND} & \quad \text{initial thread waits for all internal threads to finish (synchronize) before control continues}
\end{align*}
\]

- A thread graph represents thread creations:

- Restricted to creating trees (lattice) of threads.

- In \(\mu\)C++, a task must be created for each statement of a COBEGIN using a `Task` object:

```cpp
_Task T1 {  
    void main() { i = 1; } 
};
_TASK T2 {  
    void main() { p1(5); } 
};
_TASK T3 {  
    void main() { p2(7); } 
};
_TASK T4 {  
    void main() { p3(9); } 
};
void uMain::main() {
    // { int i, j, k; } ???
    { // \text{COBEGIN}
        T1 t1; T2 t2; T3 t3; T4 t4;
    } // \text{COEND}
}
void p1(...) {
    { // \text{COBEGIN}
        T5 t5; T6 t6; T7 t7; T8 t8;
    } // \text{COEND}
}
```

- Unusual to create objects in a block and not use them.

- For task objects, the block waits for each task’s thread to finish.
• Alternative approach for thread creation is START/WAIT, which can create arbitrary thread graph:

```plaintext
PROGRAM p
PROC p1(. . .) . . .
FUNCTION f1(. . .) . . .
INT i;
BEGIN
  (fork) START p1(5);
  thread starts in p1
  s1  continue execution, do not wait for p1
  (fork) START f1(8);
  thread starts in f1
  s2
  (join) WAIT p1; wait for p1 to finish
  s3
  (join) WAIT i := f1; wait for f1 to finish
  s4
```

• COBEGIN/COEND can only approximate this thread graph:

```plaintext
COBEGIN
  p1(5);
  BEGIN s1; COBEGIN f1(8); s2; COEND END // wait for f1!
COEND
s3; s4;
```

• START/WAIT can simulate COBEGIN/COEND:

```plaintext
COBEGIN
  START p1(. . .)
  p1(. . .) START p2(. . .)
  p2(. . .) WAIT p2
COEND
  WAIT p1
```

• In \(\mu\)C++:

```plaintext
void uMain::main() {
  T1 *p1p = new T1; // start a T1
  ... s1 ...
  T2 *f1p = new T2; // start a T2
  ... s2 ...
  delete p1p; // wait for p1
  ... s3 ...
  delete f1p; // wait for f1
  ... s4 ...
}
```

• Variable i cannot be assigned until the delete of f1p, otherwise the value could change in s2/s3.

• Allows same routine to be started multiple times with different arguments.
3.7 Termination Synchronization

- A thread terminates when:
  - it finishes normally
  - it finishes with an error
  - it is killed by its parent (or sibling) (not supported in \( \mu \text{C++} \))
  - because the parent terminates (not supported in \( \mu \text{C++} \))

- Children can continue to exist even after the parent terminates (although this is rare).
  - E.g. sign off and leave child process(s) running

- Synchronizing at termination is possible for independent threads.

- Termination synchronization may be used to perform a final communication.

- E.g., sum the rows of a matrix concurrently:

  ```
  # Task Adder {
  int *row, size, &subtotal;
  void main() {
    subtotal = 0;
    for ( int r = 0; r < size; r += 1 ) {
      subtotal += row[r];
    }
  }
  
  public:
  Adder( int row[], int size, int &subtotal ) : 
    row( row ), size( size ), subtotal( subtotal ) {} 
};

  void uMain::main() {
    int rows = 10, cols = 10;
    int matrix[rows][cols], subtotals[rows], total = 0, r;
    Adder *adders[rows];
    // read in matrix
    for ( r = 0; r < rows; r += 1 ) { // start threads to sum rows
      adders[r] = new Adder( matrix[r], cols, subtotals[r] );
    }
    for ( r = 0; r < rows; r += 1 ) { // wait for threads to finish
      delete adders[r];
      total += subtotals[r];
    }
    cout << total << endl;
  }
  ```

3.8 Synchronization and Communication during Execution

- Synchronization occurs when one thread waits until another thread has reached a certain point in its code.

- One place synchronization is needed is in transmitting data between threads.
  
  - One thread has to be ready to transmit the information and the other has to be ready to receive it, simultaneously.
  
  - Otherwise one might transmit when no one is receiving, or one might receive when nothing is transmitted.

```cpp
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 ) {}}
```

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

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

- 2 infinite loops! No, because of implicit switching of threads.

- cons synchronizes (waits) until prod transfers some data, then prod waits for cons to remove the data.

- Are 2 synchronization flags necessary?

3.9 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 (VAR parameters).

- If the threads are not in the same memory (distributed), then transferring information by value is straightforward but by address is difficult.
3.10 Exceptions

- Exceptions can be handled locally within a task, or nonlocally among coroutines, or concurrently among tasks.
  
  - All concurrent exceptions are nonlocal, but nonlocal exceptions can also be sequential.

- 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 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:

  ```c
  _Task searcher {
    searcher &partner;
    void main() {
      try {
        ...  
        if ( key == ... )
          _Throw stopEvent() _At partner;
      } catch ( stopEvent ) { ... }
  }
  ```

  When one task finds the key, it informs the other task to stop searching.

- For a concurrent raise, the source execution may only block while queueing the event for delivery at the faulting execution.

- After the event is delivered, the faulting execution propagates it at the soonest possible opportunity (next context switch); i.e., the faulting task is not interrupted.

- **Nonlocal delivery is initially disabled for a task**, so handlers can be set up before any exception can be delivered.

  ```c
  void main() {
    // initialization, no nonlocal delivery
    try { // setup handlers
      _Enable { // enable delivery of exceptions
        // rest of the code
      }
    } catch( nonlocal-exception ) {
      // handle nonlocal exception
    }
    // finalization, no nonlocal delivery
  }
  ```
3.11 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).

- This means no other thread can modify any partial results during the operation on the object (but the thread can be interrupted).

- 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 thread is called mutual exclusion.

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

- One way to handle this is to detect any sharing and serialize all access; wasteful if threads are only reading.

- 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) to maximize concurrency.

3.12 Static Variables

- Static variables in a class are shared among all objects generated by that class.

- However, shared variables may need mutual exclusion for correct usage.

- There are 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 is own name:
3.13 Mutual Exclusion Game

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

```cpp
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
}
```

- These approaches only work if one task creates all the objects so creation is performed serially.

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

### 3.13 Mutual Exclusion Game

- Is it possible to write (in your favourite programming language) some code that guarantees that a statement (or group of statements) is always serially executed by 2 threads?

- Rules of the Game:

1. Only one thread can be in its critical section at a time.

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

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

4. In selecting a thread for entry to the critical section, the selection cannot be postponed indefinitely. Not satisfying this rule is called **indefinite postponement**.

5. There must exist a bound on the number of other threads that are allowed to enter the critical section after a thread has made a request to enter it. Not satisfying this rule is called **starvation**.
3.14  Self-Testing Critical Section

```cpp
void CriticalSection() {
    ::CurrTid = &uThisTask();
    for (int i = 1; i <= 100; i += 1) { // work
        if (::CurrTid != &uThisTask()) {
            uAbort( "interference" );
        }
    }
}
```

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

3.15  Software Solutions

3.15.1  Lock

```cpp
enum Yale {CLOSED, OPEN} Lock = OPEN; // shared

class PermissionLock { public:
    PermissionLock() {}
};

void uMain::main() {
    PermissionLock t0, t1;
}
```

Breaks rule 1
3.15.2 Alternation

```cpp
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) {};

void uMain::main() {
    Alternation t0( 0 ), t1( 1 );
}

Breaks rule 3
```

3.15.3 Declare Intent

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

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

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

void uMain::main() {
    Intent me = DontWantIn, you = DontWantIn;
    DeclIntent t0( me, you ), t1( you, me );
}

Breaks rule 4
3.15.4 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) {}
};

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

Breaks rule 4
3.15. SOFTWARE SOLUTIONS

3.15.5 Prioritize Entry

```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 ) {
            if ( priority == HIGH ) { // entry protocol
                me = WantIn;
                while ( you == WantIn ) {}
            } else {
                for ( ;; ) {
                    me = WantIn;
                    if ( you == DontWantIn ) break;
                    me = DontWantIn;
                    while ( you == WantIn ) {}
                }
            }
        }
        CriticalSection(); // critical section
        me = DontWantIn; // exit protocol
    }
}

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

void uMain::main() {
    Intent me = DontWantIn, you = DontWantIn;
    PriorityEntry t0( HIGH, me, you ), t1( low, you, me );
} // main
```

Breaks rule 5
3.15.6 Dekker

```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
                me = WantIn;
                if ( you == DontWantIn ) break;
                if ( ::Last == &me ) {
                    me = DontWantIn;
                    while ( ::Last == &me ) {} // you == WantIn
                }
            }
        }
        CriticalSection(); // critical section
        ::Last = &me; // exit protocol
        me = DontWantIn;
    }
}

public:
    Dekker( Intent &me, Intent &you ) : me(me), you(you) {}
};
void uMain::main() {
    Intent me = DontWantIn, you = DontWantIn;
    ::Last = &me;
    Dekker t0( me, you ), t1( you, me );
}
```
3.15.7 Peterson

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

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

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

void uMain::main() {
    Intent me = DontWantIn, you = DontWantIn;
    Peterson t0(me, you), t1(you, me);
}
```

- Differences between Dekker and Peterson

  - Dekker’s algorithm makes no assumptions about atomicity, while Peterson’s algorithm assumes assignment is an atomic operation.

  - Dekker’s algorithm works on a machine where bits are scrambled during simultaneous assignment; Peterson’s algorithm does not.

- Prove Dekker’s algorithm has no simultaneous assignments.
3.15.8 N-Thread Prioritized Entry

```cpp
enum Intent { WantIn, DontWantIn }

_Task NTask { 
    Intent *intents;  // position & priority
    int 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();  // critical section
            intents[priority] = DontWantIn;  // exit protocol
        }
    }

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

Breaks rule 5
3.15.9 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
                if ( max < ticket[j] && ticket[j] < INT_MAX )
                    max = ticket[j];
            }
            ticket[priority] = max + 1;  // advance ticket
            // step 2, wait for ticket to be selected
            for (int j = 0; j < N; j += 1) {
                while ( ticket[j] < ticket[priority] ||
                    (ticket[j] == ticket[priority] && j < priority) ) {} // check tickets
            }
            CriticalSection();
            ticket[priority] = INT_MAX;  // exit protocol
        }
    }
}
```

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

- ticket value of $\infty$ (INT_MAX) $\Rightarrow$ don’t want in
- low ticket and position value $\Rightarrow$ high priority
- ticket selection is unusual
- tickets are not unique $\Rightarrow$ use position as secondary priority
- ticket values cannot increase indefinitely $\Rightarrow$ could fail
3.15.10 Tournament

- \( N \)-thread Prioritized Entry uses \( N \) bits.
- However, no known solution for all 5 rules using only \( N \) bits.
- N-Thread Bakery uses \( NM \) bits, where \( M \) is the ticket size (e.g., 32 bits), but is only probabilistically correct (limited ticket size).
- Other \( N \)-thread solutions are possible using more memory.
- The tournament approach uses a minimal binary tree with \( \lceil N/2 \rceil \) start nodes (i.e., full tree with \( \lceil \log N \rceil \) levels).
- Each node is a Dekker or Peterson 2-thread algorithm.
- Each thread is assigned to a particular start node, where it begins the mutual exclusion process.

![Tournament Diagram]

- At each node, one pair of threads is guaranteed to make progress; therefore, each thread eventually reaches the root of the tree.
- With 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., Last, me, you, next node).

3.15.11 Arbiter

- Create full-time arbitrator task to control entry to critical section.
3.16. HARDWARE SOLUTIONS

- Mutual exclusion becomes a synchronization between arbiter and each waiting client.
- Arbiter cycles through waiting clients ⇒ no starvation.
- Does not require atomic assignment ⇒ no simultaneous assignments.
- Cost is creation, management, and execution (continuous spinning) of the arbiter task.

3.16 Hardware Solutions

- Software solutions to the critical section problem rely on nothing other than shared information and communication between threads.
- Hardware solutions introduce level below software level.
- At this level, it is possible to make assumptions about execution that are impossible at the software level. E.g., that certain instructions are executed atomically.

```c
bool intent[5]; // initialize to false
bool serving[5]; // initialize to false

_TASK Client {
    int me;
    void main() {
        for ( int i = 0; i < 100; i += 1 ) {
            intent[me] = true;  // entry protocol
            while ( ! serving[me] ) {}  // entry protocol
            CriticalSection();
            intent[me] = false;  // exit protocol
            while ( serving[me] ) {}  // exit protocol
        }
    }
    public:
    Client( int me ) : me( me ) {}  
};

_TASK Arbiter {
    void main() {
        int i = 0;
        for ( ;; ) {
            // cycle for request => no starvation
            for ( ; ! intent[i]; i = (i + 1) % 5 ) {}
            serving[i] = true;
            while ( intent[i] ) {}  // cycle for request
            serving[i] = false;
        }
    }
};
```
• This allows elimination of much of the shared information and the checking of this information required in the software solution.

• Certain special instructions are defined to perform an atomic read and write operation.

• This is sufficient for multitasking on a single CPU.

• Simple lock of critical section failed:

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

• Imagine if the C conditional operator ? is executed atomically.

```c
while((Lock==CLOSED? CLOSED : (Lock=CLOSED),OPEN) ==CLOSED);
// critical section
Lock = OPEN;
```

• Works for N threads attempting entry to critical section and only depend on one shared datum (lock).

• However, rule 5 is broken, as there is no bound on service.

• Unfortunately, there is no such atomic construct in C.

• Atomic hardware instructions can be used to achieve this effect.

### 3.16.1 Test/Set Instruction

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

```c
int Lock = OPEN; // shared
int TestSet( int &b ) { void Task::main() { // each task does
  /* begin atomic */
  while( TestSet( Lock ) == CLOSED );
  /* critical section */
  int temp = b;
  b = CLOSED;
  Lock = OPEN;
  /* end atomic */
  return temp;
}

  if test/set returns open ⇒ loop stops and lock is set to closed
  if test/set returns closed ⇒ loop executes until the other thread sets lock to open
```

• In the multiple CPU case, memory must also guarantee that multiple CPUs cannot interleave these special R/W instructions on the same memory location.
3.16. HARDWARE SOLUTIONS

3.16.2 Swap Instruction

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

```c
int Lock = OPEN; // shared
Swap( int &a, &b )
{
    int temp;
    /* begin atomic */
    temp = a;
    a = b;
    b = temp;
    /* end atomic */
}
```

- if swap returns open ⇒ loop stops and lock is set to closed
- if swap returns closed ⇒ loop executes until the other thread sets lock to open

3.16.3 Compare/Assign Instruction

- The compare-and-assign instruction performs an atomic compare and conditional assignment (erronously called compare-and-swap).

```c
int Lock = OPEN; // shared
bool CAssn( int &val, void Task::main( ) { // each task does
    int comp, int nval )
    { // begin atomic
    if (val == comp) { Lock = OPEN;
        val = nval;
        return true;
    }
    return false;
    /* end atomic */
}
```

- if compare/assign returns open ⇒ loop stops and lock is set to closed
- if compare/assign returns closed ⇒ loop executes until the other thread sets lock to open

- compare/assign can build other solutions (stack data structure) with a bound on service but with short busy waits.
- However, these solutions are complex.
4 Lock Abstraction

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

4.1 Lock Taxonomy

• Lock implementation is divided into two general categories: spinning and blocking:

  - 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 the waiting task when the event happens.
  - Within each category, different kinds of spinning and blocking locks exist.

4.2 Spin Lock

• A spin lock is implemented using busy waiting, which spins in a loop checking for an event to occur.

• In the examples so far, if a task is busy waiting, it loops until:

  1. A critical section becomes unlocked or an event happens.
  2. The waiting task is preempted (time-slice ends) and put back on the ready queue.

Hence, the CPU is wasting time constantly checking the event.

• To increase efficiency in the uniprocessor case, a task could explicitly terminate its time-slice and move back to the ready state after the first event check fails.

• The multiprocessor case can have a spinning task terminate its time-slice and move back to the ready state after \( N \) checks have failed.

• Some systems allow the duration of spinning to be adjusted, called an adaptive spin-lock.

• Depending on how the spin lock is implemented, it may break rule 5, i.e., no bound on service, resulting in starvation of one or more tasks.

• Nevertheless, a spin lock is appropriate and necessary in situations where there is no other work to do.
4.2.1 Spin Lock Details

- \(\mu\text{C++}\) 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();
};
```

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

- Both locks are built directly from an atomic hardware instruction.
- Locks are either closed (0) or opened (1), and waiting tasks compete to acquire the lock after it is released.
- In theory, starvation could occur; in practice, it is seldom a problem.
- uSpinLock is non-preemptive, meaning no other task may execute once the lock is acquired, to permit optimizations and additional error checking in the \(\mu\text{C++}\) kernel.
- A non-preemptive lock can only be used for mutual exclusion because the task acquiring the lock must release it as no other task may execute.
- Hence, uSpinLock's constructor does not take a starting state and its instances are initialized to open.
- uLock does not have the non-preemptive restriction and can be used for both synchronization and mutual exclusion.
- tryacquire makes one attempt to acquire the lock, i.e., it does not wait.
- Any number of releases can be performed on a lock as a release only (re)sets the lock to open (1).
- It is not meaningful to read or to assign to a lock variable, or copy a lock variable, e.g., pass it as a value parameter.
- synchronization

```cpp
(Task) T1 { uLock &lk; void main() {
    ...
    lk.acquire();
    S2
    ...
} public:
    T1( uLock &lk ) : lk(lk) {}
};
```

```cpp
(Task) T2 { uLock &lk; void main() {
    ...
    S1
    lk.release();
    ...
} public:
    T2( uLock &lk ) : lk(lk) {}
};
```
4.3 Blocking Locks

- Blocking locks reduce busy waiting by making the task releasing the lock do additional work, called cooperation.

- Hence, the responsibility for detecting an open lock is not borne solely by an acquiring task, but is shared with the releasing task.

4.3.1 Mutex Lock

- A mutex lock is used only for mutual exclusion.

- Tasks waiting on these locks block rather than spin when an event has not occurred.

- 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 (lock owner) cannot acquire it again

– **multiple acquisition**: lock owner can acquire it multiple times, called an **owner lock**

• Single acquisition cannot handle looping or recursion involving a lock:

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

• Multiple-acquisition locks may require only one release to unlock, or as many releases as acquires.

### 4.3.1.1 Mutex Lock Implementation

• Implementation of a mutex lock requires a:

  – blocking task to link itself onto a list and yield its time slice
  – releasing task to unblock a waiting task (cooperation)

• However, operations like adding and removing a node to/from a list are not atomic $\Rightarrow$ mutual exclusion.
class MutexLock {
    spinlock lock;       // nonblocking lock
    queue<Task> blocked; // blocked tasks
    bool inUse;          // resource being used?
    Task *owner;         // optional

public:
    MutexLock() : inUse( false ), owner( NULL ) {} 
    void acquire() { 
        lock.acquire();
        if ( inUse
            && owner != thistask() ) { // (optional)
            // add self to lock's blocked list
            lock.release();    // release before blocking
            // yield
            lock.acquire();    // re-acquire lock
        }
        inUse = true;
        owner = thistask();   // set new owner (optional)
        lock.release();
    }
    void release() { 
        lock.acquire();
        owner = NULL;         // no owner (optional)
        if ( ! blocked.empty() ) { 
            // remove task from blocked list and make ready
        } else { 
            inUse = false;
        }
        lock.release();       // always release lock
    }
};

- Which task is scheduled next from the list of blocked tasks?
- Has the busy wait been removed completely?
- An alternative approach is not to release the spin lock if there is a waiting task, hence:
  - the mutual exclusion is transfered from the releasing task to the unblocking task
  - and the critical section is not bracketed by the spin lock.
4.3.1.2 uOwnerLock Details

- $\mu$C++ only provides an owner lock, uOwnerLock, which subsumes a mutex lock.

```cpp
class uOwnerLock {
public:
  uOwnerLock();
  void acquire();
  bool tryacquire();
  void release();
};
```

- The operations are the same as for uLock but with blocking instead of spinning for acquire.

4.3.1.3 Stream Locks

- Concurrent use of C++ streams can produce unpredictable and undesirable results; e.g., if two tasks execute:

```cpp
task1 : cout << "abc " << "def " << endl;
task2 : cout << "uvw " << "xyz " << endl;
```

any of the outputs can appear:

```
abc def  |  abc uvw xyz  |  uvw abc xyz def  |  abuvw defyz  |  uvw abc def
uvw xyz  |  def           |                         | yz            | xyz
```

- $\mu$C++ uses owner locks to provide mutual exclusion for streams: 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:

```cpp
task1 : osacquire( cout ) << "abc " << "def " << endl;
task2 : osacquire( cout ) << "uvw " << "xyz " << endl;
```

constraining the output to two different lines in either order:

```
abc def  | uvw xyz
uvw xyz  | abc def
```

- 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
```
4.3. BLOCKING LOCKS

4.3.2 Synchronization Lock

- A synchronization lock is used solely for synchronization, for the same reasons as a mutex lock is used only for mutual exclusion.

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

4.3.2.1 Synchronization Lock Implementation

- Synchronization lock implementation has two forms:

  **external locking** uses an external mutex lock to protect its state,

  **internal locking** uses an internal mutex lock to protect its state.

- Like the mutex lock, the blocking task waits on a list and the releasing task performs the necessary cooperation.

- external locking

```cpp
class SyncLock {
    Task *list;

public:
    SyncLock() : list( NULL ) {}  
    void acquire( MutexLock &mutexlock ) {
        // add self to lock's blocked list
        mutexlock.release();
        // yield
    }
    void release() {
        if ( list != NULL ) {
            // remove task from blocked list and make ready
        }
    }
};
```

- The protecting mutex-lock is passed to acquire to be released by the blocking task.

- Alternatively, the protecting mutex-lock is bound at synchronization-lock creation and used implicitly.

- internal locking
class SyncLock {
    spinlock mlock; // nonblocking lock
    Task *list; // blocked tasks

public:
    SyncLock() : list( NULL ) {}
    void acquire( MutexLock &userlock ) {
        mlock.acquire(); // add self to task list
        userlock.release(); // release before blocking
        mlock.release(); // yield
    }
    void release() {
        mlock.acquire();
        if ( list != NULL ) {
            // remove task from blocked list and make ready
        }
        mlock.release();
    }
};

• It is still useful (necessary) to be able to unlock an external mutex lock before blocking.
• Otherwise, there is a race between releasing the mutex lock and blocking on the synchronization lock.

4.3.2.2 uCondLock Details
• µC++ only provides an internal synchronization lock, uCondLock.

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

• empty() returns false if there are tasks blocked on the queue and true otherwise.
• wait and signal are used to block a thread on and unblock a thread from the queue of a condition, respectively.
• wait atomically blocks the calling task and releases the argument owner-lock;
• wait re-acquires its argument owner-lock before returning.
• signal releases tasks in FIFO order.
• broadcast is the same as the signal routine, except all waiting tasks are unblocked.

```cpp
bool done = false;

_Task T1 {
    uOwnerLock &mlk;
    uCondLock &clk;

    void main() {
        mlk.acquire();
        if (!done) clk.wait(mlk);
        else mlk.release();
        S2;
    }

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

void uMain::main() {
    uOwnerLock mlk;
    uCondLock clk;
    T1 t1( mlk, clk );
    T2 t2( mlk, clk );
}

_Task T2 {
    uOwnerLock &mlk;
    uCondLock &clk;

    void main() {
        S1;
        mlk.acquire();
        done = true;
        clk.signal();
        mlk.release();
    }

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

4.3.3 Barrier

• A barrier is used to repeatedly coordinate a group of tasks performing a concurrent operation surrounded by a series of sequential operations.

• Hence, a barrier is specifically for synchronization and cannot be used to build mutual exclusion.

• Unlike previous synchronization locks, a barrier retains some state about the events it manages.

• 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.
The barrier is initialized to control 3 tasks and passed to each task by reference (not copied).

The barrier works by blocking each task at the call to block until all 3 tasks have reached their call to block on barrier b.

The last task to call block detects that all the tasks have arrived at the barrier, and releases all the tasks (cooperation).

Hence, all 3 tasks continue execution together after they have all arrived at the barrier.

Notice, it is necessary to know in advance the total number of block operations executed before the tasks are released.

Why not use termination synchronization and create new tasks for each computation?

The reason is that the creation and deletion of computation tasks may be an unnecessary expense that can be eliminated by using a barrier.

4.3.4 Semaphore

A semaphore lock provides synchronization and mutual exclusion (Edsger W. Dijkstra).

`semaphore lock(0); // 0 => closed, 1 => open, default 1`

Like the barrier, a semaphore retains state (counter) to “remember” releases.

names for acquire and release from Dutch terms

acquire is P

– passeren ⇒ to pass
– prolagen ⇒ (proberen) to try (verlagen) to decrease

`lock.P(); // wait to enter`

P decrements the semaphore counter and waits if it is zero.
• release is V
  – vrijgeven ⇒ to release
  – verhogen ⇒ to increase

  lock.V();  // release lock

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

• When the semaphore counter has only two values (0, 1), it is called a **binary semaphore**.

• Semaphore implementation is similar to mutex/synchronization locks, with the counter.

• For example, 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.

• I.e., statement S1 and S2 are independent ⇒ can execute in either order or at the same time.

• Statement S3 is dependent on S1 and S2 because it uses both results.

• Display dependences graphically in a **precedence graph** (which is different from a 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

- process graph

Does this solution work?

### 4.3.5 Counting Semaphore

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

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

- E.g. Three tasks must execute a section of code in a particular order. S2 and S3 only execute after S1 has completed.

```c
X::main() { Y::main() { Z::main() {
  ... s.P(); s.P(); s.V(); // s.V(2)
  S2 S3 s.V();
  ... ... ...}
}
}
```

```c
void uMain::main() {
  uSemaphore s(0);
  X x(s);
  Y y(s);
  Z z(s);
}
```

- The problem is that you must know in advance the total number of P’s on the semaphore (like a barrier).
4.3.5.1 uSemaphore Details

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

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

- P decrements the semaphore counter; if the counter is greater than or equal to zero, the calling task continues, otherwise it blocks.
- TryP returns true if the semaphore is acquired and false otherwise (never blocks).
- V wakes up the task blocked for the longest time if there are tasks blocked on the semaphore and increments the semaphore counter.
- If V is passed a positive integer value, the semaphore is V-ed that many times.
- The member routine counter returns the value of the semaphore counter:
  - negative means abs(N) tasks are blocked waiting to acquire the semaphore, and the semaphore is locked;
  - zero means no tasks are waiting to acquire the semaphore, and the semaphore is locked;
  - positive means the semaphore is unlocked and allows N tasks to acquire the semaphore.
- The member routine empty returns false if there are threads blocked on the semaphore and true otherwise.

4.4 Semaphore Programming

- 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) ⇒ used to control a critical section.
- E.g. Unbounded-Buffer Problem
  - Two tasks are communicating unidirectionally through a queue of unbounded length.
  - Producer adds results to the end of a queue of unbounded length; consumer removes items from the front of the queue unless the queue is empty.
– Because they work at different speeds, the producer may get ahead of the consumer.

– The producer never has to wait as the buffer is infinitely long, but the consumer may have to wait if the buffer is empty.

– The queue is shared between the producer/consumer, and a counting semaphore controls access.

```c
#define QueueSize ∞

int front = 0, back = 0;
int Elements[QueueSize];
uSemaphore signal(0);

void Producer::main() {
    for (;;) {
        // produce an item
        // add to back of queue
        signal.V();
    }
    // produce a stopping value
}

void Consumer::main() {
    for (;;) {
        signal.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 signal semaphore used for mutual exclusion or synchronization?

• E.g. Bounded-Buffer Problem

  – Two tasks are communicating unidirectionally through a buffer of bounded length.

  – Because of the bounded length, the producer may have to wait for the consumer to empty it. As well, the buffer may be empty so the consumer may have to wait until items are produced.

  – Use counting semaphores to account for the finite length of the shared queue.
4.4. SEMAPHORE PROGRAMMING

```c
uSemaphore full(0), empty(QueueSize);
void Producer::main() {
    for ( . . . ) {
        // produce an item
        empty.P();
        // add to back of queue
        full.V();
    }
    // produce a stopping value
}

void Consumer::main() {
    for ( . . . ) {
        full.P();
        // take an item from the front of the queue
        if ( stopping value ? ) break;
        // process or consume the item
        empty.V();
    }
}
```

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

```
<table>
<thead>
<tr>
<th></th>
<th>34</th>
<th>13</th>
<th>9</th>
<th>10</th>
<th>-3</th>
</tr>
</thead>
<tbody>
<tr>
<td>full</td>
<td>Φ</td>
<td>Φ′</td>
<td>Φ</td>
<td>Φ</td>
<td>Φ</td>
</tr>
<tr>
<td>empty</td>
<td>Φ′</td>
<td>Φ′</td>
<td>Φ′</td>
<td>Φ′</td>
<td>Φ′</td>
</tr>
</tbody>
</table>
```

- E.g. Readers and Writer Problem (Solution 1)
  - Multiple tasks are sharing a resource, and some of them want to read the resource and some want to write or modify the resource.
  - Allow multiple concurrent reader tasks simultaneous access, but serialize access for writer tasks (a writer may read).
  - 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.
– 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 baton
  * nobody moves in the entry/exit code unless they have it
  * once the baton is released, cannot read/write variables in entry/exit
uSemaphore entry_q(1);    // split binary semaphore
uSemaphore read_q(0), write_q(0);
int r_cnt = 0, w_cnt = 0;  // auxiliary
int r_del = 0, w_del = 0;

void Reader::main() {
    entry_q.P();             // entry protocol
    if ( w_cnt > 0 ) {
        r_del += 1; entry_q.V(); read_q.P();
    }
    r_cnt += 1;
    if ( r_del > 0 ) {
        r_del -= 1; read_q.V();    // pass baton
    } else {
        entry_q.V();
    }

    yield();                // pretend to read

    entry_q.P();             // exit protocol
    r_cnt -= 1;
    if ( r_cnt == 0 && w_del > 0 ) {
        w_del -= 1; write_q.V();    // pass baton
    } else {
        entry_q.V();
    }
}

void Writer::main() {
    entry_q.P();             // entry protocol
    if ( r_cnt > 0 || w_cnt > 0 ) {
        w_del += 1; entry_q.V(); write_q.P();
    }
    w_cnt += 1;
    entry_q.V();

    yield();                // pretend to write

    entry_q.P();             // exit protocol
    w_cnt -= 1;
    if ( r_del > 0 ) {
        r_del -= 1; read_q.V();    // pass baton
    } else if ( w_del > 0 ) {
        w_del -= 1; write_q.V();    // pass baton
    } else {
        entry_q.V();
    }
}
– Problem: continuous stream of readers ⇒ no writer can get in, so starvation.

• E.g. Readers and Writer Problem (Solution 2)

  – Give writers priority and make the readers wait.
  – Change entry protocol for reader to the following:

```c
entry_q.P();  // entry protocol
if ( w_cnt > 0 || w_del > 0 ) {
    r_del += 1; entry_q.V(); read_q.P();
}
 r_cnt += 1;
if ( r_del > 0 ) {
    r_del -= 1; read_q.V();
} else {
    entry_q.V();
}
```

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

```c
entry_q.P();  // exit protocol
w_cnt -= 1;
if ( w_del > 0 ) {
    // check writers first
    w_del -= 1; write_q.V();
} else if ( r_del > 0 ) {
    r_del -= 1; read_q.V();
} else {
    entry_q.V();
}
```

  – Now readers can starve.

• E.g. Readers and Writer Problem (Solution 3)

  – Readers wait if there is a waiting writer, all readers are started after a writer (i.e. alternate group of readers then a writer)
  – There is a problem: staleness/freshness.

• E.g. Readers and Writer Problem (Solution 4)
– Service readers and writers in a first-in first-out (FIFO) fashion, but allow multiple concurrent readers.

– Must put readers and writers on same condition variable to maintain temporal order but then lose what kind of task.

– Maintain a shadow list to know the kind of task is waiting at front of condition variable:

  semaphore  t_0 \rightarrow t_1 \rightarrow t_2 \rightarrow t_3 \rightarrow t_4 \rightarrow \ldots
  shadow-list  w \rightarrow r \rightarrow w \rightarrow r \rightarrow r \rightarrow \ldots
5 Concurrent Errors

5.1 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.
- Can be very difficult to locate (thought experiments).

5.2 No Progress

5.2.1 Live-lock

- indefinite postponement: the “You go first” problem on simultaneous arrival
- Caused by poor scheduling:

  ![Diagram of live-lock](image)

- There always exists some scheduling algorithm that can deal effectively with live-lock.

5.2.2 Starvation

- A selection algorithm ignores one or more tasks so they are never executed.
- I.e., lack of long-term fairness.
- Long-term (infinite) starvation is extremely rare, but short-term starvation can occur and is a problem.

5.2.3 Deadlock

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

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

```cpp
void uMain::main() {
    uSemaphore s(0); // closed
    s.P(); // wait for lock to open
}
```

5.2.3.2 Mutual Exclusion Deadlock

- Failure to acquire a resource protected by mutual exclusion.

Deadlock, unless one of the cars is willing to backup.

- Simple example using semaphores:

```cpp
uSemaphore L1(1), L2(1); // open
task1 task2
L1.P() L2.P() // acquire opposite locks
R1 R2 // access resource
L2.P() L1.P() // acquire opposite locks
R1 & R2 R2 & R1 // access resource
```

- There are 5 conditions that must occur for a set of processes to get into Deadlock.

1. There exists a shared resource requiring mutual exclusion.
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 O/S cannot get it back (no preemption).
4. There exists a circular wait of processes on resources.
5. These conditions must occur simultaneously.
5.3 Deadlock Prevention

- Eliminate one or more of the conditions required for a deadlock from an algorithm ⇒ deadlock can never occur.

5.3.1 Synchronization Prevention

- Eliminate all synchronization from a program
  - ⇒ no communication
  - all tasks must be completely independent, generating results through side-effects.

5.3.2 Mutual Exclusion Prevention

- Deadlock can be prevented by eliminating one of the 5 conditions:
  1. no mutual exclusion: impossible in many cases
  2. no hold & wait: do not give any resource, unless all resources can be given
     - ⇒ poor resource utilization
     - possible starvation
  3. allow preemption:
     - Preemption is dynamic ⇒ cannot apply statically.
  4. no circular wait:
     - Control the order of resource allocations to prevent circular wait:

```c
uSemaphore L1(1), L2(1); // open

// acquire same locks
L1.P() L1.P() R1 R1
R2.R2

// access resource
L2.P() L2.P() R2
R2
```

- Use an ordered resource policy:

```
R_1 < R_2 < R_3 ...
```

- 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
– in some cases there is a natural division of resources into classes that makes this
  policy work nicely
– in other cases, some processes are forced to acquire resources in an unnatural
  sequence, complicating their code and producing poor resource utilization

5. prevent simultaneous occurrence:
   • Show previous 4 rules cannot occur simultaneously.

5.4 Deadlock Avoidance
   • Monitor all lock blocking and resource allocation to detect any potential formation of dead-
     lock.

   ![Deadlock Diagram]

   • Achieve better resource utilization, but additional overhead to avoid deadlock.

5.4.1 Banker’s Algorithm
   • Demonstrate a safe sequence of resource allocations to processes that $\Rightarrow$ no deadlock.
   • However, to do this requires that a process state its maximum resource needs.

<table>
<thead>
<tr>
<th></th>
<th>R1</th>
<th>R2</th>
<th>R3</th>
<th>R4</th>
</tr>
</thead>
</table>
   | T1  | 4  | 10 | 1  | 1  | maximum needed
   | T2  | 2  | 4  | 1  | 2  | for execution
   | T3  | 5  | 9  | 0  | 1  | (M)
   | T1  | 23 | 5  | 1  | 0  | currently
   | T2  | 1  | 2  | 1  | 0  | allocated
   | T3  | 1  | 2  | 0  | 0  | (C)

   resource request (T1, R1) 2 $\rightarrow$ 3

<table>
<thead>
<tr>
<th></th>
<th>R1</th>
<th>R2</th>
<th>R3</th>
<th>R4</th>
</tr>
</thead>
</table>
   | T1  | 1  | 5  | 0  | 1  | needed to
   | T2  | 1  | 2  | 0  | 2  | execute
   | T3  | 4  | 7  | 0  | 1  | $(N = M - C)$
• Is there a safe order of execution that avoids deadlock should each process require its maximum resource allocation?

<table>
<thead>
<tr>
<th>total available resources</th>
</tr>
</thead>
<tbody>
<tr>
<td>R1</td>
</tr>
<tr>
<td>6</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>current available resources</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
</tr>
<tr>
<td>T2</td>
</tr>
<tr>
<td></td>
</tr>
</tbody>
</table>

| T1 | 1  | 0  | 3  | 1  | (CR = CR - N_{T1}) |
|    | 5  | 10 | 4  | 2  | (CR = CR + M_{T1}) |

| T3 | 1  | 3  | 4  | 1  | (CR = CR - N_{T3}) |
|    | 6  | 12 | 4  | 2  | (CR = CR + M_{T3}) |

• If there is a choice of processes to choose for execution, it does not matter which path is taken.

• Example: If T1 or T3 could go to their maximum with the current resources, then choose either. A safe order starting with T1 exists if and only if a safe order starting with T3 exists.

• So a safe order exists (the left column in the table above) and hence the Banker’s Algorithm allows the resource request.

• The check for a safe order is performed for every allocation of resource to a process and then process scheduling is adjusted appropriately.

5.4.2 Allocation Graphs

• One method to check for potential deadlock is to graph processes and resource usage at each moment a resource is allocated.

• 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 a graph contains no cycles, no process in the system is deadlocked.

• If each resource has one instance, a cycle $\Rightarrow$ deadlock.

T1 $\rightarrow$ R1 $\rightarrow$ T2 $\rightarrow$ R3 $\rightarrow$ T3 $\rightarrow$ R2 $\rightarrow$ T1 (deadlock)
T2 $\rightarrow$ R3 $\rightarrow$ T3 $\rightarrow$ R2 $\rightarrow$ T2 (deadlock)

• If any resource has several instances, a cycle $\not\Rightarrow$ deadlock.
  
  – If T4 releases its resource, the cycle is broken.

• Create isomorphic graph without multiple instances (expensive and difficult):

• Use graph reduction to locate deadlocks:
Problems:
- for large numbers of processes and resources, detecting cycles is expensive.
- there may be large number of graphs that must be examined, one for each particular allocation state of the system.

5.5 Detection and Recovery

- Instead of avoiding deadlock let it happen and recover.
  - ⇒ ability to discover deadlock
  - ⇒ preemption

- Discovering deadlock is not easy, 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

- Recovery involves preemption of one or more processes in a cycle.
  - decision is not easy and must prevent starvation
  - The preemption victim must be restarted, from beginning or some previous checkpoint state, if you cannot guarantee all resources have not changed.
  - even that is not enough as the victim may have made changes before the preemption.
5.6 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.
6 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.
- Both of these can be complicated to use and may be incorrectly placed.
- Split-binary semaphores and baton passing are complex.
- Need higher level facilities that perform some of these details automatically.
- Get help from programming-language/compiler.

6.1 Critical Regions

- Declare which variables are to be shared, as in:

  ```
  VAR v : SHARED INTEGER
  ```

- Access to shared variables is restricted to within a `REGION` statement, and within the region, mutual exclusion is guaranteed.

  ```
  semaphore v_lock(1);
  REGION v DO  P(v_lock)
  // critical section  ...  // x = v; (read)  v = y (write)
  END REGION V(v_lock)
  ```

- Nesting can result in deadlock.

  ```
  VAR x, y : SHARED INTEGER
  ```

  ```
  task1 REGION x DO  task3 REGION y DO
  ...  ...
  REGION y DO  REGION x DO
  ...  ...
  END REGION  END REGION
  ```

- 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.
6.2 Conditional Critical Regions

- Introduce a condition that must be true as well as having mutual exclusion.

\[
\text{REGION } v \text{ DO}
\begin{align*}
& \text{AWAIT conditional-expression} \\
& \ldots
\end{align*}
\text{END REGION}
\]

- E.g. The consumer from the producer-consumer problem.

\[
\text{VAR } Q : \text{SHARED QUEUE<INT,10>}
\text{REGION } Q \text{ DO}
\begin{align*}
& \text{AWAIT NOT EMPTY( } Q ) \text{ buffer not empty} \\
& \text{take an item from the front of the queue}
\end{align*}
\text{END REGION}
\]

If the condition is false, the region lock is released and entry is started again (busy waiting).

6.3 Monitor

- A monitor is an abstract data type that combines shared data with serialization of its modification.

\[
\text{\_Monitor name } \{ \\
\text{shared data} \\
\text{members that see and modify the data}
\};
\]

- A mutex member (short for mutual-exclusion member) is one that does NOT begin execution if there is another active mutex member.

  - ⇒ a call to a mutex member may become blocked waiting entry, and queues of waiting tasks may form.
  
  - Public member routines of a monitor are implicitly mutex and other kinds of members can be made explicitly mutex (\_Mutex).

- Basically each monitor has a lock which is Ped on entry to a monitor member and Ved on exit.
6.4 SCHEDULING (SYNCHRONIZATION)

```cpp
class Mon {
    int v;
    uSemaphore MonitorLock(1)

public:
    int x(...) {
        MonitorLock.P()
        ... // int temp = v;
        MonitorLock.V()
        return v; // return temp;
    }
};
```

- Unhandled exceptions raised within a monitor always release the implicit monitor locks so the monitor can continue to function.

- Atomic counter using a monitor:

  ```cpp
  _Monitor AtomicCounter {
      int counter;

public:
    AtomicCounter( int init ) : counter( init ) {}
    int inc() { counter += 1; return counter; }
    int dec() { counter -= 1; return counter; }
  }

  AtomicCounter a, b, c;
  ... a.inc(); ...
  ... b.dec(); ...
  ... c.inc(); ...
  ```

- Recursive entry is allowed (owner mutex lock), i.e., one mutex member can call another or itself.

- Destructor is mutex, so ending a block with a monitor or deleting a dynamically allocated monitor, blocks if thread in monitor.

6.4 Scheduling (Synchronization)

- A monitor may want to schedule tasks in an order different from the order in which they arrive.

- 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.
6.4.1 External Scheduling

- The accept statement controls which mutex members can accept calls.

- By preventing certain members from accepting calls at different times, it is possible to control scheduling of tasks.

- E.g. Bounded Buffer

```cpp
_Monitor BoundedBuffer {  
    int front, back, count;
    int Elements[20];
    public:
        BoundedBuffer() : front(0), back(0), count(0) {}  
        __Nomutex int query() { return count; }  
        __Mutex void insert(int elem);  
        __Mutex int remove();
};
void BoundedBuffer::insert(int elem) {  
    if (count == 20) _Accept(remove); // hold insert calls
    Elements[back] = elem;
    back = (back + 1) % 20;
    count += 1;
}
int BoundedBuffer::remove() {  
    if (count == 0) _Accept(insert); // hold remove calls
    int elem = Elements[front];
    front = (front + 1) % 20;
    count -= 1;
    return elem;
}
```

- Queues of tasks form outside the monitor, waiting to be accepted into either insert or remove.

- An acceptor blocks until a call to the specified mutex member(s) occurs.

- Accepted call is executed like a conventional member call.

- When the accepted task exits the mutex member (or blocks), the acceptor continues.

- If the accepted task does an accept, it blocks, forming a stack of blocked acceptors.

6.4.2 Internal Scheduling

- Scheduling among tasks inside the monitor.

- A condition is a queue of waiting tasks:

```cpp
uCondition x, y, z[5];
```
• A task waits (blocks) by placing itself on a condition:

\[ x.wait(); \quad // \text{wait( mutex, condition )} \]

*Atomically* places the executing task at the back of the condition queue, and allows another task into the monitor by releasing the monitor lock.

• A task on a condition queue is made ready by signalling the condition:

\[ x.signal(); \]

This removes a blocked task at the front of the condition queue and makes it ready.

• Like Ving a semaphore, the signaller does not block, so the signalled task must continue waiting until the signaller exits or waits.

• A signal on an empty condition is lost!

• E.g. Bounded Buffer (like binary semaphore solution):

```java
Monitor BoundedBuffer {
    uCondition NonEmpty, NonFull;
    int front, back, count;
    int Elements[20];

    public:
    BoundedBuffer() : front(0), back(0), count(0) {} // Nomutex
    int query() { return count; }
    void insert(int elem) {
        if (count == 20) NonFull.wait();
        Elements[back] = elem;
        back = (back + 1) % 20;
        count += 1;
        NonEmpty.signal();
    }
    int remove() {
        if (count == 0) NonEmpty.wait();
        int elem = Elements[front];
        front = (front + 1) % 20;
        count -= 1;
        NonFull.signal();
        return elem;
    }
};
```

• *wait()* blocks the current thread, and restarts a signalled task or implicitly releases the monitor lock.

• *signal()* unblocks the thread on the front of the condition queue *after* the signaller thread blocks or exits.
• **signalBlock()** unblocks the thread on the front of the condition queue and blocks the signaller thread.

• **General Model**

```cpp
_Monitor Mon {
    uCondition A, B;
    ...
    public:
    int X(...) {...}
    void Y(...) {...}
};
```

• The **entry queue** is a queue of all calling tasks in the order the calls were made to the monitor.

• **explicit scheduling** occurs when:
  
  – An accept statement blocks the active task on the acceptor stack and makes a task ready from the specified mutex member queue.
  
  – A signal moves a task from the specified condition to the signalled stack.

• **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.

| explicit scheduling | internal scheduling (signal) |
| implicit scheduling | monitor selects (wait/exit) |

• Use external scheduling unless:
  
  – scheduling depends on member parameter value(s), e.g., compatibility code for dating:
  
  – a task might be further blocked while in the monitor (e.g., wait for additional resources)
• Use implicit mutex queues to prevent double (queueing) blocking.

6.5 Readers/Writer

• E.g. Readers and Writer Problem (Solution 3)
Monitor
ReadersWriter {
  int rcnt, wcnt;
  uCondition readers, writers;
public:
  ReadersWriter() : rcnt(0), wcnt(0) {} 
  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();
  }
};

• Can the monitor read/write members perform the reading/writing?

• Has the same protocol problem as P and V.

ReadersWriter rw;
  readers
  rw.startRead()
  rw.endRead()
  // read

writers
  rw.startWrite()
  rw.endWrite()
  // write

• Alternative interface:
Monitor ReadersWriter {
    Mutex void startRead();
    Mutex void endRead();
    Mutex void startWrite();
    Mutex void endWrite();
    public:
    Nomutex void read(...) {
        startRead(); // read
        // read
        endRead();
    }
    Nomutex void write(...) {
        startWrite(); // write
        endWrite();
    };
}

• E.g. Readers and Writer Problem (Solution 4)

Monitor ReadersWriter {
    int rcnt, wcnt;
    uCondition RWers;
    enum RW { READER, WRITER };
    public:
    ReadersWriter() : rcnt(0), wcnt(0) {}
    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();
    };
}
• E.g. Readers and Writer Problem (Solution 5)

```java
_Monitor ReadersWriter {
    int rcnt, wcnt;
    public:
        ReadersWriter() : rcnt(0), wcnt(0) {}
        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?

### 6.6 Condition, Signal, Wait vs. Counting Semaphore, V, P

• There are several important differences between these mechanisms:

  – wait always blocks, P only blocks if semaphore = 0

  – if signal occurs before a wait, it is lost, while a V before a P affects the P

  – multiple Vs may start multiple tasks simultaneously, while multiple signals only start one task at a time because each task must exit serially through the monitor

• It is possible to construct P and V using a monitor:
6.7 Monitor Types

- 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.

- Assigning different priorities to these queues creates different monitors (e.g., C < W < S).

- Many of the possible orderings can be rejected as they do not produce a useful monitor (e.g., W < S < C).

- Implicit Signal

  - Monitors either have an explicit signal (statement) or an implicit signal (automatic signal).
The implicit signal monitor has no condition variables or explicit signal statement.

Instead, there is a `waitUntil` statement, e.g.:

```
waitUntil logical-expression
```

The implicit signal causes a task to wait until the conditional expression is true.

```c
_Monitor BoundedBuffer {
    int front, back, count;
    int Elements[20];

public:
    BoundedBuffer() : front(0), back(0), count(0) {}

_Nomutex int query() { 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;
}
}
```

- There is a restricted monitor type that requires that the signaler exit immediately from the monitor (i.e., `signal ⇒ return`), called immediate-return signal.

- Ten kinds of useful monitor are suggested:

<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>
<tr>
<td>Quasi-blocking</td>
<td>Priority Quasi</td>
<td>No Priority Quasi</td>
</tr>
<tr>
<td></td>
<td>$C &lt; W = S$</td>
<td>$C = W = S$</td>
</tr>
<tr>
<td>Immediate Return</td>
<td>Priority Return</td>
<td>No Priority Return</td>
</tr>
<tr>
<td></td>
<td>$C &lt; W$</td>
<td>$C = W$</td>
</tr>
<tr>
<td>Implicit Signal</td>
<td>Implicit Signal</td>
<td>Implicit Signal</td>
</tr>
<tr>
<td></td>
<td>$C &lt; W$</td>
<td>$C = W$</td>
</tr>
</tbody>
</table>
6.7. MONITOR TYPES

– No-priority monitors require the signalled task to recheck the waiting condition in case of a **barging** task.

⇒ use a **while** loop around a **wait** instead of an **if**

– Implicit (automatic) signal monitors are good for prototyping but have poor performance.

– Immediate-return monitors are not powerful enough to handle all cases but optimize the most common case of signal before return.

– Quasi-blocking monitors makes cooperation too difficult.

– priority-nonblocking monitor has no barging and optimizes signal before return (supply cooperation).

– priority-blocking monitor has no barging and handles internal cooperation within the monitor (wait for cooperation).

• Java monitor

– synchronized (wrong name) ⇒ mutex

– only one condition variable per monitor

  (new Java library has multiple conditions but are incompatible with language condition)

– condition operations: **wait()**, **signal()**, **notifyall()**

– no-priority nonblocking monitor ⇒ **while** (**!C**) **wait();**

• coroutine monitor

– 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.
7 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:

```
Task1          monitor          Task2
  |      ↑                  |      |
  |      ↑                  |      |
  |      ↑                  |      |
  ↑      ↑                  ↑      ↑
  ↑      ↑                  ↑      ↑
  ↑      ↑                  ↑      ↑
  ↑      ↑                  ↑      ↑
  ↑      ↑                  ↑      ↑
  ↑      ↑                  ↑      ↑
  ↑      ↑                  ↑      ↑
  ↑      ↑                  ↑      ↑
  ↑      ↑                  ↑      ↑
```

- Point-to-point with reply direct communication:

```
Task1          Task2
  |      ↑                  |      ↑
  |      ↑                  |      ↑
  |      ↑                  |      ↑
  ↑      ↑                  ↑      ↑
  ↑      ↑                  ↑      ↑
  ↑      ↑                  ↑      ↑
  ↑      ↑                  ↑      ↑
  ↑      ↑                  ↑      ↑
  ↑      ↑                  ↑      ↑
  ↑      ↑                  ↑      ↑
  ↑      ↑                  ↑      ↑
```

- Therefore, tasks need to communicate directly.

7.1 Task

- A task is like a monitor, because it provides mutual exclusion and can also perform synchronization.
  - Public members of a task are implicitly mutex and other kinds of members can be made explicitly mutex.
- A task is also like a coroutine, because it has a distinguished member, (task main), which has its own execution state.
- A task is unique because it has a thread of control, which begins execution in the task main when the task is created.
- In general, basic execution properties produce different abstractions:
<table>
<thead>
<tr>
<th>object properties</th>
<th>member routine properties</th>
</tr>
</thead>
<tbody>
<tr>
<td>thread</td>
<td>stack</td>
</tr>
<tr>
<td>No</td>
<td>No</td>
</tr>
<tr>
<td>No</td>
<td>Yes</td>
</tr>
<tr>
<td>Yes</td>
<td>No</td>
</tr>
<tr>
<td>Yes</td>
<td>Yes</td>
</tr>
</tbody>
</table>

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

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

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

### 7.2 Scheduling

- A task may want to schedule access to itself by other tasks in an order different from the order in which requests arrive.

- As for monitors, there are two techniques: external and internal scheduling.

#### 7.2.1 External Scheduling

- As for a monitor, the accept statement can be used to control which mutex members of a task can accept calls.
7.2. SCHEDULING

_Task BoundedBuffer {
    int front, back, count;
    int Elements[20];

public:
    BoundedBuffer() : front(0), back(0), count(0) {} 
_Nomutex int query() { 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 ( ;; ) {
            _When (count != 20) _Accept(insert) { // after call
                // after call
            } or _When (count != 0) _Accept(remove) { // after call
                // _Accept
            } // for
        }
    }
};

• _Accept( m1, m2 ) S1 ⊑ _Accept( m1 ) S1; or _Accept( m2 ) S1;

if ( C1 | C2 ) S1 ⊑ if ( C1 ) S1; else if ( C2 ) S1;

• Extended version allows different _When and code after call for each accept.

• Equivalence using if statements:

if ( 0 < count && count < 20 ) _Accept( insert, remove );
else if ( count < 20 ) _Accept( insert );
else /* if ( 0 < count ) */ _Accept( remove );

• $2^N - 1$ if statements needed to simulate $N$ accept clauses.

• Why is BoundedBuffer::main defined at the end of the task?

• The _When clause is like the condition of conditional critical region:

  – The condition must be true (or omitted) and a call to the specified member must exist before a member is accepted.
• If all the accepts are conditional and false, the statement does nothing (like `switch` with no matching `case`).

• If some conditionals are true, but there are no outstanding calls, the acceptor is blocked until a call to an appropriate member is made.

• The acceptor is pushed on the top of the A/S stack and normal implicit scheduling occurs (C < W < S).

• If several members are accepted and outstanding calls exist to them, a call is selected based on the order of the `_Accepts`.

  – Hence, the order of the `_Accepts` indicates their relative priority for selection if there are several outstanding calls.

• Once the accepted call has completed, the statement after the accepting `_Accept` clause is executed.

  – To achieve greater concurrency in the bounded buffer, change to:
7.2. SCHEDULING

```c
void insert( int elem ) {
    Elements[back] = elem;
}

int remove() {
    return Elements[front];
}

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
    } // for
}

• 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

– Hence, the terminating else clause allows a conditional attempt to accept a call without the acceptor blocking.

• An exception raised in a task member propagates to the caller, and a special exception is raised at the task’s thread to identify a problem.

_Task T {
    public:
    void mem() {
        ... _Throw E; ... // E goes to caller
    } // uRendezvousFailure goes to “this”

    private:
    void main() {
        try {
            __Enable {
                ... __Accept( mem ); ... // assume call completed
            }
        } catch ( uSerial::RendezvousFailure ) {
            // deal with rendezvous failure
        } // try
    }
};
```
7.2.2 Accepting the Destructor

- Common way to terminate a task is to have a done (join) member:

```cpp
_Task BoundedBuffer {
    public:
        . . .
    void done() { /* empty */ }

    private:
    void main() {
        // start up
        for ( ;; ) {
            _Accept( done ) { // terminate ?
                break;
            } or _When ( count != 20 ) _Accept( insert ) {
                . . .
            } or _When ( count != 0 ) _Accept( remove ) {
                . . .
            } // _Accept
        } // for
        // close down
    }
}
```

- Call done when task is to stop:

```cpp
void uMain::main() {
    BoundedBuffer buf;
    // create producer & consumer tasks
    // delete producer & consumer tasks
    buf.done(); // no outstanding calls to buffer
    // maybe do something else
    // delete buf
}
```

- Alternatively, throw a concurrent exception, but delayed deliver.

- If termination and deallocation follow one another, accept destructor:

```cpp
void main() {
    for ( ;; ) {
        _Accept( ~BoundedBuffer ) {
            break;
        } or _When ( count != 20 ) _Accept( insert ) {
            . . .
        } or _When ( count != 0 ) _Accept( remove ) {
            . . .
        } // _Accept
    } // for
}
```
• However, the semantics for accepting a destructor are different from accepting a normal mutex member.

• When the call to the destructor occurs, the caller blocks immediately if there is thread active in the task because a task’s storage cannot be deallocated while in use.

• When the destructor is accepted, the caller is blocked and pushed onto the A/S stack instead of the acceptor.

• Therefore, control restarts at the accept statement without executing the destructor member.

• This allows a mutex object to clean up before it terminates (monitor or task).

• At this point, the task behaves like a monitor because its thread is halted.

• Only when the caller to the destructor is popped off the A/S stack by the implicit scheduling is the destructor executed.

• The destructor can reactivate any blocked tasks on condition variables and/or the acceptor/signalled stack.

7.2.3 Internal Scheduling

• Scheduling among tasks inside the monitor.

• As for monitors, condition, signal and wait are used.

```cpp
(Task) BoundedBuffer {
    uCondition NonEmpty, NonFull;
    int front, back, count;
    int Elements[20];
    public:
        BoundedBuffer() : front(0), back(0), count(0) {}

    (Nomutex) int query() { return count; }

    void insert( int elem ) {
        if ( count == 20 ) NonFull.wait();
        Elements[back] = elem;
        back = ( back + 1 ) % 20;
        count += 1;
        NonEmpty.signal();
    }
```

int remove() {
    if ( count == 0 ) NonEmpty.wait();
    int elem = Elements[front];
    front = ( front + 1 ) % 20;
    count -= 1;
    NonFull.signal();
    return elem;
}

protected:
void main() {
    for ( ;; ) {
        _Accept( ~BoundedBuffer )
        break;
        or _Accept( insert, remove );
        // do other work
    } // for
}
};

• Is there a potential starvation problem?

7.3 Increasing Concurrency

• 2 task involved in direct communication: client (caller) & server (callee)

• possible to increase concurrency on both the client and server side

7.3.1 Server Side

• Use server thread to do administrative work so client can continue concurrently (assuming no return value).

• E.g., move administrative code from the member to the statement executed after the member is accepted:

  _Task server1 {
     public:
     void xxx(...) { S1 }
     void yyy(...) { S2 }
     void main() {
         ...
         _Accept( xxx );
         or _Accept( yyy );
     }
  }

  _Task server2 {
     public:
     void xxx(...) { S1.copy }
     void yyy(...) { S2.copy }
     void main() {
         ...
         _Accept( xxx ) { S1.admin }
         or _Accept( yyy ) { S2.admin };
     }
  }

• overlap between client and server increases potential for concurrency
7.3. INCREASING CONCURRENCY

7.3.1.1 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?
- I.e., an internal buffer can be used to store the arguments of multiple clients until the server processes them.
- However, there are several issues:
  - Unless the average time for production and consumption is approximately equal with only a small variance, the buffer is either always full or empty.
  - Because of the mutex property of a task, no calls can occur while the server is working, so clients cannot drop off their arguments.
    The server could periodically accept calls while processing requests from the buffer (awkward).
  - 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.

- These problems can be handled by changing the server into an administrator.

7.3.1.2 Administrator

- An administrator is used to manage a complex interaction or complex work or both.
- The key is that an administrator does little or no “real” work; its job is to manage.
- Management means delegating work to others, receiving and checking completed work, and passing completed work on.
- An administrator is called by others; hence, an administrator is always accepting calls.
- An administrator makes no call to another task because such calls may block the administrator.
- 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

7.3.2 **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.

- Asynchronous call requires implicit buffering between client and server to store the client’s arguments from the call.

- µC++ provides only synchronous call, i.e., the caller is delayed from the time the arguments are delivered to the time the result is returned (like a procedure call).

- It is possible to build asynchronous facilities out of the synchronous ones and vice versa.
7.3. INCREASING CONCURRENCY

7.3.2.1 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:
  1. transmit the arguments
  2. retrieve the results

- The time between the two calls allows the calling task to execute asynchronously with the task performing the operation on the caller’s behalf.

- If the result is not ready when the second call is made, the caller blocks or the caller has to call again (poll).

- However, this requires a protocol so that when the client makes the second call, the correct result can be found and returned.

7.3.2.2 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 passes the ticket to retrieve the result.

- The ticket is matched with a result, and the result is returned if available or the caller is blocks or polls until the result is available.

- However, protocols are error prone because the caller may not obey the protocol (e.g., never retrieve a result, use the same ticket twice, forged ticket).

7.3.2.3 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 original client.

- The original client must poll the indicator or block until the indicator is set.

- The advantage is that the server does not have to store the result, but can drop it off immediately.

- Also, the client can write the call-back routine, so they can decide to poll or block or do both.
7.3.2.4 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.
- Further, it removes the difficult problem of when the caller should try to retrieve the result.
- In detail, a future is an object that is a subtype of the result type expected by the caller.
- Instead of two calls as before, a single call is made, passing the appropriate arguments, and a future is returned.
- The future is returned immediately and it is empty.
- The caller “believes” the call completed and continues execution with an empty result value.
- The future is filled in at some time in the “future”, when the result is calculated.
- If the caller tries to use the future before its value is filled in, the caller is implicitly blocked.
- A simple future can be constructed out of a semaphore and link field, as in:

```cpp
class future {
    friend _Task server;       // can access internal state

    uSemaphore resultAvailable;
    future *link;
    ResultType result;

public:
    future() : resultAvailable( 0 ) {}

    ResultType get() {
        resultAvailable.P();       // wait for result
        return result;
    }
};
```

- the semaphore is used to block the caller if the future is empty
- the link field is used to chain the future onto a server work-list.

- 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.
8 Other Approaches

8.1 Languages with Concurrency Constructs

8.1.1 Ada 95

- Like µC++, but not as general.

- E.g., monitor bounded-buffer, restricted implicit (automatic) signal:

```ada
protected type buffer is -- _Monitor
    entry insert( elem : in ElemType ) when count < Size is
    begin
        -- add to buffer
        count := count + 1;
    end insert;
    entry remove( elem : out ElemType ) when count > 0 is
    begin
        -- remove from buffer, return via parameter
        count := count - 1;
    end remove;
private:
    ... // buffer declarations
    count : Integer := 0;
end buffer;
```

- The when clause is external scheduling because it can 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 dating-service.

- E.g., task bounded-buffer:
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 can only appear in a **task**, not any of its subprograms

• Hence, Ada has no direct internal-scheduling mechanism, i.e., no condition variables.

• Instead a requeue statement can send a request to be postponed to another (usually non-public) mutex member of the object.

• The request is re-blocked on that mutex member’s entry queue, which can be subsequently accepted when the request can be restarted.

• However, all requeue techniques suffer the problem of dealing with accumulated temporary results:
  
  – If a request must be postponed, its temporary results must be returned and bundled with the initial request before forwarding to the mutex member handling the next step.
  
  – Alternatively, the temporary results can 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.

8.1.2 Modula-3/Java/C#

• Java’s concurrency constructs are largely derived from Modula-3.
8.1. LANGUAGES WITH CONCURRENCY CONSTRUCTS

```java
class Thread implements Runnable {
    public Thread();
    public Thread(String name);
    public String getName();
    public void setName(String name);
    public void run();
    public synchronized void start();
    public static Thread currentThread();
    public static void yield();
    public final void join();
}
```

- Thread is like uBaseTask in µC++, and all tasks must explicitly inherit from it:

```java
class myTask : 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 the 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.

- Java has synchronized class members (i.e., _Mutex members but incorrectly named), and a synchronized statement.

- Modula-3 has a lock statement that can be used to simulate _Mutex members if certain coding conventions are followed.
• Neither language has very useful external scheduling.

• While it is possible to have public synchronized members of a task, there is no mechanism to manage direct calls, i.e., no accept statement.
  ⇒ no useful direct communication.

• Internal scheduling is no-priority nonblocking ⇒ barging ⇒ wait statements must be in while loops to recheck conditions

• All classes have one implicit condition variable and these routines to manipulate it:
  
  ```java
  public wait();
  public notify();
  public notifyall()
  ```

• 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;
      notify();
    }

    public synchronized int remove() {
      while ( count == 0 ) wait(); // busy-waiting
      // remove from buffer
      count -= 1;
      notify();
      return elem;
    }
  }
  ```

• Because there is only one condition queue ⇒ certain solutions are difficult or impossible.

8.2 Threads & Locks Library

• All concurrency libraries are unsafe or inefficient because the compiler thinks the program is sequential.

• unsafe:
  
  – Valid sequential optimizations can invalidate concurrent programs:
    
    ```java
    P(lock);
    // critical section
    V(lock);
    ```

    compiler moves P or V after/before critical section
- Turn off sequential optimizations to prevent concurrent error but now entire program is unnecessarily slower.

- Several libraries exist for C (pthreads) and C++ (µC++).

- C libraries built around simple model of starting a thread in a routine 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.

```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 ⇒ no direct communication.

- Internal scheduling is no-priority nonblocking ⇒ barging ⇒ wait statements must be in while loops to recheck conditions
typedef struct {
    // buffer declarations
    pthread_mutex_t mutex;     // mutual exclusion
    pthread_cond_t Full, Empty; // synchronization
} buffer;

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 );
    . . .
    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.

**8.3 Threads & Message Passing**

- Message passing is an alternative mechanism to parameter passing.

- In message passing, all information transmitted is grouped into a single data area and (usually) passed by value.

- Hence, all pointers must be dereferenced before being put into the message (i.e., no pointers are passed in a message).

- This makes a message independent of the context of the message sender (i.e., no shared memory is required).

- Hence, the receiver can be on the same or different machines, it makes no difference (distributed systems).

- On shared memory machines, pointers can still be passed.

- Message passing is usually direct communication.

- Messages are directed to a specific task and received from a specific task (receive specific):

  ```
  task_1
  send(tid_2, msg)
  task_2
  receive(tid_1, msg)
  ```

- Or messages are directed to a specific task and received from any task (receive any):

  ```
  task_1
  send(tid_2, msg)
  task_2
  tid = receive(msg)
  ```

**8.3.1 Nonblocking Send**

- Send does not block, and receive gets the messages in the order they are sent (SendNonBlock).

  ⇒ an infinite length buffer between the sender and receiver

- since the buffer is bounded, the sender occasionally blocks until the receiver catches up.

- the receiver blocks if there is no message

```
Producer() {
    for (;;) {
        produce an item
        SendNonBlock(CId, msg);
    }
}
```

```
Consumer() {
    for (;;) {
        Receive(PId, msg);
        consume the item
    }
}
```
CHAPTER 8. OTHER APPROACHES

8.3.2 Blocking Send

- Send or receive blocks until a corresponding receive or send is performed (SendBlock).
- I.e., information is transferred when a rendezvous occurs between the sender and the receiver

8.3.3 Send-Receive-Reply

- Send blocks until a reply is sent from the receiver, and the receiver blocks if there is no message (SendReply).

Producer() {
  for (;;)
    prod = ReceiveReply(msg);
  SendReply(CId, ans, msg);
}

Consumer() {
  for (;;)
    prod = ReceiveReply(msg);
    Reply(prod, ans)
}

- Why use receive any instead of receive specific?
- E.g., Producer/Consumer
  - Producer
    for (i = 1; i <= NoOfItems; i += 1) {
      msg = rand() % 100 + 1;
      cout << "Producer: " << msg << endl;
      SendReply(Cons, rmsg, msg);
    }
    msg = -1;
    SendReply(Cons, rmsg, msg);
  - Consumer
    for (;;) {
      prod = ReceiveReply(msg);
      // check msg
      Reply(prod, rmsg);
      if (msg < 0) break;
      cout << "Consumer: " << msg << endl;
    }

8.4 Message Format

- variable-size messages
  - complex implementation, easy to use
- fixed-size message
8.4. MESSAGE FORMAT

- simple/fast implementation
- requires long messages to be broken up and transmitted in pieces and reconstructed by the receiver

• typed message
  - only messages of a certain type can be sent and received among tasks
  - requires dynamic type checking
Index

_Accept_  80, 93
    destructor, 96
_At_  33
_Coroutine_  9
_Disable_  23
_Enable_  23
_Monitor_  78
_Mutex_  78
_Nomutex_  80
_Task_  29
_Throw_  23, 33
_When_  93

active, 5
Ada 95, 103
adaptive spin-lock, 49
administrator, 99
    worker tasks, 99
allocation graphs, 73
alternation, 37
arbiter, 44
atomic, 34, 41, 45, 79, 81, 109
atomic instruction
    compare/assign, 47
    swap, 47
    test/set, 46
automatic signal, 87

bakery, 43
banker’s algorithm, 72
barging, 89, 107
barrier, 57
baton passing, 64
binary semaphore, 59, 81
blocking send, 110
bottlenecks, 26
bounded buffer, 62, 80, 81, 88, 93, 98
buffer
    bounded, 62
    unbounded, 61
busy wait, 32, 49, 53
busy waiting, 49, 51
call-back, 101
client side, 100
    call-back, 101
    future, 102
    returning values, 101
ticket, 101
COBEGIN, 29, 59
COEND, 29
communication, 32
    direct, 91
compare/assign instruction, 47
concurrency, 25
    difficulty, 25
    increasing, 98
    why, 25
concurrent error
    indefinite postponement, 69
    live lock, 69
    race condition, 69
    starvation, 69
concurrent exception, 33
concurrent execution, 25
concurrent hardware structure, 27
concurrent systems
    explicit, 26
    implicit, 26
    structure, 26
condition, 80, 86
condition lock, 55
    details, 56
conditional critical region, 78
context switch, 9
8.4. MESSAGE FORMAT

internal scheduling, 80
scheduling, 79
signal, 81, 86
simulation, 78
wait, 81, 86
monitor type
  no priority blocking, 88
  no priority immediate return, 88
  no priority implicit signal, 88
  no priority nonblocking, 88
  no priority quasi, 88
  priority blocking, 88
  priority immediate return, 88
  priority implicit signal, 88
  priority nonblocking, 88
  priority quasi, 88
monitor types, 87
multiple acquisition, 52
multiprocessing, 27
multiprocessor, 27
multitasking, 27
mutex lock, 51
mutex member, 78
mutual exclusion, 34, 62
  alternation, 37
  deadlock, 70
declare intent, 37
Dekker, 40
game, 35
lock, 36, 46
N-thread
  arbiter, 44
  bakery, 43
  prioritized entry, 42
tournament, 44
Peterson, 41
prioritize entry, 39
retract intent, 38

no priority blocking, 88
no priority immediate return, 88
no priority implicit signal, 88
no priority nonblocking, 88
no priority quasi, 88
non-preemptive, 50

nonblocking send, 109
ordered resource, 71, 76
ostream
  osacquire, 54
owner lock, 52
details, 54
P, 58, 61, 70, 71, 77, 79, 86
parallel execution, 25
passeren, 58
Peterson, 41
precedence graph, 59
prioritize entry, 39
prioritized entry, 42
priority blocking, 88
priority immediate return, 88
priority implicit signal, 88
priority nonblocking, 88
priority quasi, 88
process, 25
processor
  multi, 27
  uni, 27
prolagen, 58
pthreads, 107
race condition, 69
readers/writer, 83
  freshness, 66
monitor
  solution 3, 83
  solution 4, 85
  solution 5, 86
semaphore
  solution 1, 63
  solution 2, 66
  solution 3, 66
  solution 4, 66
staleness, 66
rendezvous, 110
resume, 9, 18, 23
retract intent, 38

scheduling, 79, 92
  explicit, 82
external, 80, 92
implicit, 82
internal, 80, 97
self testing, 36
semaphore, 58, 70, 71, 79
binary, 59, 81
counting, 60, 86
details, 61
integer, 60
P, 58, 70, 71, 77, 79, 86
programming, 61
split binary, 63
V, 59, 77, 79, 86
semi-coroutine, 6, 18
send/receive/reply, 110
server side
administrator, 99
buffer, 99
signal, 86
automatic, 87
explicit, 87
immediate-return, 88
implicit, 87
signal, 81, 98
single acquisition, 52
spin lock, 49
details, 50
split binary semaphore, 63
staleness, 66
START, 30, 59
starvation, 35, 66, 69
state transition, 28
static variable, 34
stream lock, 54
suspend, 9
swap instruction, 47
synchronization, 62, 79
communication, 32
deadlock, 70
during execution, 32
termination, 31
task, 25
exceptions, 33
external scheduling, 92
internal scheduling, 97
main, 29, 91
scheduling, 92
static variable, 34
terminated, 5
termination synchronization, 31, 58
test/set instruction, 46
thread, 25
communication, 28
creation, 28
synchronization, 28
thread graph, 29
ticket, 101
time-slice, 49
tournament, 44
tryacquire, 50
uCondition, 80, 98
uCondLock, 56
broadcast, 56
empty, 56
signal, 56
wait, 56
uLock
acquire, 50
release, 50
tryacquire, 50
unbounded buffer, 61
uniprocessor, 27
uOwnerLock, 54
acquire, 54
release, 54
tryacquire, 54
uSemaphore, 61, 70, 71, 79
counter, 61
empty, 61
P, 61, 70, 71, 79
TryP, 61
V, 61, 79
uSpinLock
acquire, 50
release, 50
tryacquire, 50
V, 59, 61, 77, 79, 86
WAIT, 30
wait, 86
wait, 81, 98
worker tasks, 99
  complex, 100
courier, 100
notifier, 100
simple, 100
timer, 100